Applying Programming Language Implementation Techniques To Processor Simulation

by

Eric C. Schnarr

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin — Madison

2000

ACKNOWLEDGMENTS

I am grateful to my advisor James R. Larus for his insights that helped define the direction of my work, and his comments that helped me express it more eloquently. Without Jim's support, I would not have been able to pursue my ideas for applying programming language design and optimization techniques to simulation. I am also grateful for Jim's continued support, even after he left academia for a job in industry.

I would like to thank Mark Hill for becoming my advisor in Jim's absence, for his advice, and for keeping me on track in finishing my Ph.D.

This research is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF NYI Award CCR-9337779, NSF Grants CCR-9101035, MIP-9225097, MIP-9625558, EIA-9971256, and CDA-9623632, DOE Grant DE-FG02-93ER25176, an Intel Graduate Fellowship, and donations from Compaq Computer Corporation, Intel Corporation, and Sun Microsystems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I: Introduction

Instruction-level simulators are used for a variety of applications, where the target hardware is either unavailable or lacks features only possible in simulation. But as processor designs have grown more complex and contain more implicit and explicit parallelism, simulators for these designs have become slower and more difficult to implement. For example, while improvements in simulator design may allow a simulator of a simple in-order processor pipeline to run with less than ten times slowdown, typical simulators for out-of-order processors suffer several thousand times slowdown.

I develop several techniques to implement and optimize instruction-level micro-architecture simulators. New variations of programming language optimizations, such as partial evaluation and memoization, improve the performance of out-of-order processor simulation by an order of magnitude over traditional simulation techniques. A new special-purpose programming language—called Facile—simplifies the implementation and optimization of instruction-level simulators. The result is that complex micro-architecture simulators can be written in Facile, then automatically optimized using partial evaluation, memoization, and potentially even run-time code generation to run faster than traditional, hand-coded simulators.

Section 1.1 describes what is meant by instruction-level simulation, what these simulators are used for, and why optimizing their performance is so important. Section 1.2 describes program-

ming language techniques that I have adapted to micro-architecture simulation. Section 1.3 highlights my contributions and outlines the organization of this dissertation.

1.1. Instruction-Level Simulation

Instruction-level simulation focuses on emulating machine instructions rather than modeling individual gates and wires. Instruction-level simulators model an instruction set architecture, but can also include cycle accurate models of some or all of a target micro-architecture. By contrast, logic-level simulators model hardware implementations in terms of logic gates and wires, and circuit-level simulators model circuits in even more detail. An instruction-level simulator can be cycle accurate, but could not be used to determine a hardware implementation's maximum clock speed, for example, the way a detailed, circuit-level simulator can.

Instruction-level simulation is an essential part of research and development of new processors. In processor development, instruction-level simulators at several levels of detail are used to refine processor designs. Multiple levels of simulation are necessary because simulators can execute faster by ignoring details that are not relevant in the early stages of processor design. ISA simulators that only simulate instruction semantics are generally very fast and provide a reference point for correct execution of instructions. They also allow development of compilers and other system software before actual hardware is available.

Performance modeling of individual micro-architecture components—instruction and data caches, the execution pipeline, etc.—slows down simulation, but allows designers to experiment

with various component designs. The most detailed instruction-level simulators model an entire processor micro-architecture, which can show how components interact to affect overall processor performance. Beyond this point, extremely slow logic-level and circuit-level simulators are used to model the detailed processor implementation, and a sufficiently detailed circuit-level description can be automatically synthesized into a circuit layout suitable for etching onto a chip.

In research, simulators allow new technologies to be studied without the time and expense of constructing real hardware. Instruction-level simulators can focus on the specific parts of a computer system being studied (e.g. data cache implementation or branch predictors), or simulate an entire computer system, including the operating system and device hardware [64][45]. Instruction-level simulators are commonly used by architecture researchers to study facets of processor performance, because they are usually accurate enough to validate new architecture ideas and to collect data about the components of existing processor architectures. Hardware counters or logic probes are another way to study processor behavior, but these are limited by the counters implemented in existing hardware and the signals visible to a logic probe. Another motivation for using simulators in research is that their internal workings are more accessible than the internals of actual hardware, and so more data on the internal behavior can be collected.

Architecture simulators are also useful in education. A simulator can provide students with a simplified machine interface for learning about instruction sets and assembly language programming. As with researchers, a simulator provides students with access to the internal working of a

processor's micro-architecture. Students can use simulators to experiment with pipelines, caches, branch predictors, and other basic hardware components.

Another use for instruction-level simulators is to run software on systems with hardware other than that for which the software was compiled. Such simulation allows system software to be developed before a working chip is available. Using this technique, operating systems, device drivers, compilers, and other system tools have been developed concurrently with the development of their target hardware, hence reducing the time to get a new system operational [9]. Moreover, a sufficiently fast emulator—e.g., FX!32 [15]—can allow software products written for one architecture to run on a completely different host architecture.

Instruction-level simulators are used in these applications, rather than more detailed hardware simulators because of their performance. Logic-level simulators are implemented using a fine grained event-driven simulation that is difficult to optimize. Circuit-level simulators solve matrices of equations relating circuit voltages, currents, and resistances, and execute even more slowly. Instruction-level simulators, which model only interesting processor behaviors, are usually much faster than these other simulators for the same micro-architecture. Instruction-level simulators need only model the micro-architecture components of interest, and are usually written in a procedural programming style using languages like C, which makes them inherently faster to execute.

An instruction-level simulator that emulates only the semantic behavior of an ISA can be made to simulate target programs with little or no slowdown, compared to executing target pro-

grams on actual hardware. For example, FX!32 executes 4.4 Alpha instructions to emulate one Intel x86 instruction on average. On a 500 MHz Alpha, FX!32 has performance between that of a 200 MHz Pentium and a 200 MHz Pentium Pro [15]. Adding more detail to the simulated processor model—e.g., simulating instruction and data caches or collecting the cycle count and other statistics for a particular processor pipeline—slows down execution of the simulator and increases the complexity of its implementation. Instruction-level simulation of a detailed, cycle accurate model of a complex out-of-order processor (e.g. MIPS R10000) typically incurs a several thousand times slowdown over execution on actual hardware [13][54][64], and it takes several man-months to implement and debug the simulator.

Although the hardware used to run simulations is becoming faster, simulator complexity and run-time cost is also increasing. Micro-architectures are becoming more complex, and this is reflected in the increased complexity and greater execution cost of the simulators that model them. Researchers also require larger, longer running benchmarks for studies of new hardware or to validate new micro-architecture innovations. These factors combine to make processor simulation more time consuming, and simulators more difficult to implement.

A variety of technologies have been used to increase the performance of instruction-level simulators. Some strategies focus on interpreting target instructions faster by transforming them into more efficient representations. In the extreme, target instructions are re-compiled directly into the native instruction set of the simulation host. Other strategies rely on statistical sampling to reduce the simulation work needed to collect data from large benchmarks. Processor statistics can be col-

lected for a small set of randomly selected samples from the target execution, then results for the entire execution inferred from these samples. Accuracy is a big concern with statistical simulation techniques, since only part of a target program is simulated and the samples may be simulated inaccurately due to cold-start bias[1].

1.2. Programming Language Optimizations

My approach is to apply technologies developed to optimize programming languages to optimize instruction-level processor simulators. Primarily, I have adapted memoization—a well-known technique for optimizing functional programming languages—for use in micro-architecture simulation. Additionally, I use algorithms, commonly used for partial evaluation, to find the parts of a simulator that can be skipped over by memoization. These algorithms simplify the implementation of a memoizing simulator, by adding memoization automatically to a simulator implementation. Run-time code generation can also be used to further optimize a memoizing simulator. These technologies are briefly described below.

MEMOIZATION. Memoization involves caching the results of function calls, indexed by the function being called and the value of its arguments. When a function is called again with the same argument values, then its result is looked up in the memoization cache rather then executing the function a second time. Traditional memoization is most applicable to pure functional languages because function calls in these languages have no side effects.

1. Cold-start bias refers to inaccuracies caused by micro-architecture state that is not maintained between samples—e.g., the set of data in a cache or branch predictor—but is needed to accurately simulate sampled instructions.

My variation—called *fast-forwarding*—detects and skip over repeated work in a simulator. Unlike memoization, fast-forwarding caches simulation code rather than simple return values. Given a subset of the simulated micro-architecture state, some parts of a simulator are guaranteed to produce the same results and can be skipped over. What is cached is the parts of a simulator that may not produce the same results, indexed by the given subset of micro-architecture state. Fast-forwarding greatly accelerates some detailed micro-architecture simulation, because the same instruction sequences are simulated many times, producing nearly the same micro-architecture state and simulation results each time.

PARTIAL EVALUATION. This refers to the partial execution of a program given only part of its input data. Consider a program `p` that accepts input `[in1, in2]`. The effect of partial evaluation is to specialize `p` together with part of its input `in1` to produce a new program $\texttt{p}_{\texttt{in1}}$. The specialized program $\texttt{p}_{\texttt{in1}}$ can then be called with input `in2` to compute the same result as $[\![\texttt{p}]\!]$ `[in1, in2]` (i.e., `p` called with input `[in1, in2]`). The benefit of partial evaluation is that $\texttt{p}_{\texttt{in1}}$ is often faster than the original program `p`.

Algorithms used in partial evaluation to direct how programs are specialized, direct which code to cache in a fast-forwarding simulator. The idea is that a subset of the simulated micro-architecture state is known and partial evaluation can be used to specialize the simulator for this state. The difference between fast-forwarding and partial evaluation is that partial evaluation is a compile time optimization, whereas simulator specialization must occur at run-time.

RUN-TIME CODE GENERATION. Run-time code generation (RTCG) uses run-time data values to produce specialized code at run-time. Because specialization is performed at run-time, more is known about a program's data values and faster specialized code can be produced. This is essentially how fast-forwarding works: Simulator code is specialized with run-time data values at run-time, using annotations computed at compile-time. The difference is that RTCG systems generate machine instructions at run-time. Currently, my implementations of fast-forwarding interprets memoized data rather than memoizing and executing machine instructions.

1.3. My Contributions

My contribution is the adaptation of several programming language techniques to improve the performance of instruction level processor simulators and to simplify their implementation.

- Fast-forwarding is my new variation on memoization that accelerates the simulation of complex micro-architectures.

- Facile is my new special purpose language for writing instruction-level micro-architecture simulators. It is designed to both simplify simulator implementation, and constrain programs to simplify the analyses and optimizations performed by Facile's compiler.

- Partial evaluation algorithms allow the Facile compiler to add memoization to a micro-architecture simulator automatically. Run-time code generation could be used to further optimize these memoizing simulators.

Chapter II discusses related work. There are many instruction-level simulators, and a variety of implementation techniques have been used to make them run fast. Some techniques simulate each instruction more efficiently, other techniques accelerate simulation by only simulating part of a program's execution trace. Chapter II also discusses partial evaluation and run-time code generation: what they are and some examples of their use. Finally, work related to architecture description languages is discussed, including my previous work on SADL—the Spawn Architecture Description Language [39].

Chapter III applies memoization, along with other techniques such as direct execution, to optimize a detailed, cycle accurate simulator for an out-of-order processor. The resulting simulator—FastSim v.1—runs an order of magnitude faster than the SimpleScalar out-of-order simulator [13], while simulating a similarly complex microarchitecture. Most of this performance improvement was the result of memoization. The drawback of this simulator is its complexity and inflexibility. The implementation of memoization was difficult to get right and the resulting simulator is difficult to modify without breaking it.

Chapters IV and V describe the special purpose language Facile and its compiler, which is used to produce memoizing simulators, respectively. Facile is designed to make memoizing simulators more flexible and easier to implement. Special syntax, derived loosely from the New Jersey Machine Code Toolkit specification language [60], allows programmers to easily describe an instruction set architecture. More general programming language constructs allow implementation of the rest of an instruction-level micro-architecture simulator. In the compiler, partial evalua-

tion algorithms automatically identify which parts of a simulator, written in Facile, can be skipped over by memoization, then it generates an optimized simulator that uses memoization.

Chapter VI discusses the performance of memoizing simulators written in Facile. In an out-of-order processor simulator, memoization was very effective at accelerating simulator execution. An out-of-order processor simulator written in Facile executes an order of magnitude faster with memoization than without it. Simple, in-order simulators written in Facile were not improved by memoization. Not enough code could be skipped over by memoizing these simple simulators to overcome the extra cost of memoization. Writing simulators that make effective use of memoization requires an understanding of how memoization works and how to structure the simulator code. By experimenting with several similar simulators that contain only small structural differences, Chapter VI demonstrates strategies for writing more efficient memoizing simulators.

Chapter VII discuses several optimizations that could make memoized code run more efficiently. These include optimizations that were left out of the current compiler because they are too complex to implement quickly or because their usefulness was not realized until after the compiler was written. Run-time code generation is an optimization that was not implemented in the current compiler, because of time constraints. Chapter VII discusses two ways RTCG could be implemented in a memoizing simulator. Other optimization are discussed as well.

Chapter VIII concludes.

CHAPTER II: Related Work

A combination of techniques for (micro-)architecture simulation, compiler optimization, and architecture description form a starting point for the work described in later chapters. This chapter discusses work that precedes my own in these areas. Section 2.1 covers many instruction-level simulators and simulation technologies used to implement instruction-level micro-architecture simulators efficiently. Section 2.2 discusses relevant programming language and compiler techniques. These include partial evaluation, run-time code generation, and existing architecture description languages.

2.1. Accelerating Processor Simulation

There are a large number of processor simulators and a wide variety of techniques for making them run fast. This section describes some techniques that have been used to accelerate these simulations. Section 2.1.1 surveys non-statistical techniques that accelerate the decode-despatch-execute cycle of a typical instruction-level simulator. Section 2.1.2 discusses trace sampling, a statistical technique that accelerates simulation by only simulating samples of a program trace. The result of simulating an entire program trace is extrapolated from the results of simulating these smaller samples. Although I am concerned primarily with instruction-level simulation, much work has gone into improving logic-level simulators (e.g., written in Verilog or VHDL), and I describe some of this work in Section 2.1.3.

2.1.1. Instruction-Level Simulators

The following table is an extended and updated version of a table taken from the 1994 SIGMETRICS paper on Shade [17]. It gives a partial list of instruction-level simulators (and similar tools) covering a variety of simulation technologies and other attributes.

TABLE 2.1: Summary of contemporary instruction level simulators.

Name	Purpose	Input Rep.	Detail	MD	MP	Signals	SMC OK	Technology	Bugs OK
Accelerator[5]	sim	exe	us	Y	N	Y	Y	scc+gi	Y
ATOM[69]	tb_C	exe*	u	N	N	Y	N	aug	N
ATUM[2]	sim/atr	exe	us	Y	$Y_=$	Y	Y	emu	Y
dis+mod+run[28]	sim/atr	asm	u	N	N	N	N	scc	N
Dynascope[68]	db/atr/otr	hll	u	N	N	S	Y	pdi	Y
EEL[39]	tb_C	exe	u	N	N	Y	Y	aug	Y
Executor[17]	sim	exe	u	N	N	Y	Y	pdi	Y
FastSim v.1[65]	sim	exe	u	N	N	Y	N	aug+ffw	Y
FastSim v.2	tb_C	exe	u	N	N	Y	Y	ddi+ffw	Y
FX32[15]	sim	exe	u	N	N	Y	Y	ddi+scc	Y
g88[9]	sim/db	exe	usd	Y	N	Y	Y	tci	Y
gsim[43][44]	sim/db/atr/otr/tb_C	exe	usd	Y	Y_1	Y	Y	tci+dcc	Y
Mable[23]	sim/db/atr	exe	u	N	Y_1	N	Y	ddi	N
Migrant[66]	sim	exe	u	Y	N	Y	Y	scc+emu	Y
Mimic[46]	sim	exe	u	N	N	N	N	dcc	N
MINT[73]	atr	exe	u	N	Y_1	Y	N	pdi+dcc	Y*
Moxie[16]	sim	exe	u	N	N	Y	N	scc	N
MPtrace[26]	atr	asm	u	N	$Y_=$	S	N	aug	N
MX/Vest[67]	sim	exe	u	N	$Y_=$	Y	Y	scc+gi	Y
Pixie[48]	atr	exe*	u	Y	N	Y	N	aug	N
Pixie-II[17]	atr/otr/db	exe*	us	Y	N	Y	S	scc	N
Proteus[12]	atr	hll	u	N	Y_1	N	S	aug	N
Purify[32]	db	exe*	u	N	N	Y	N	aug	Y
qp/qpt[38]	atr/otr	exe	u	N	N	N	N	aug	N
RPPT[21]	atr	hll	u	N	Y_1	N	N	aug	N
RSIM[54]	sim/atr/otr	exe	u	Y	N	N	N	emu	Y

TABLE 2.1: Summary of contemporary instruction level simulators.

Name	Purpose	Input Rep.	Detail	MD	MP	Signals	SMC OK	Technology	Bugs OK
SELF[14]	sim	exe	u	N	N	Y	Y	dcc	Y
Shade[17]	sim/atr/otr/tb_C	exe	u	N	N	Y	Y	dcc	N
SimICS[45]	sim/atr/otr	exe	usd	Y	Y_1	Y	Y	tci	Y
SimOS[34]	sim/atr/otr	exe	usd	Y	Y_+	Y	Y	ddi+dcc	Y
SimpleScalar[13]	sim/atr/otr	exe	u	N	N	N	N	pdi	Y
SoftPC[52]	sim	exe	u(s)d	N	N	Y	Y	dcc	Y
Spa[17]	atr	exe	u	N	N	S	Y	ddi	N
SPIM[33]	sim/atr	exe	u	N	N	Y	N	pdi	Y
ST-80[24]	sim	exe	u	N	N	Y	Y	dcc	Y
Talisman[10]	sim/db/atr/otr/tb_C	exe	usd	Y	Y_1	Y	Y	tci	Y
Tango Lite[29]	atr	asm	u	N	Y_1	N	S	aug	N
Titan[11]	atr	exe	us	Y	N	Y	N	aug	N
TRAPEDS[71]	atr	asm	us	Y	$Y_=$	S	N	aug	N
WWT[62]	atr/otr	exe	u	Y	Y_+	Y	N	emu+aug+ddi	Y
Z80MU[8]	sim	exe	u(s)	N	N	Y	Y	ddi	Y

Purpose indicates the use for which the tool was intended: cross-architecture simulation (sim), debugging (db), address tracing or memory hierarchy analysis (atr), or more detailed kinds of tracing (otr). Tools marked tb_C are tool-building tools and usually use C as the extension language. *Input* describes the language of the target programs read by the simulator: high-level language (hll), assembly code (asm), or executable code (exe). Some tools read executables but also require symbol table information to work properly (exe*). *Detail*: most tools work with only user-level code (u), some also run system-level code (s), and system mode simulation generally requires device emulation (d). Some target machines have no system mode, so simulation can avoid the costs of address translation and protection checks; these machines have the system mode marked in parenthesis.

MD reports whether the tool supports multiple protection domains and multi-tasking (i.e., multiple processes per target processor). *MP* tells whether the tool supports multiple processor execution: only a single host processor (Y_1), one target processor per host processor ($Y_=$), or several target processors per host processor (Y_+). *Signals* indicates whether the tool can handle asynchronous events like signals (Y or N) or only some of the possible events (S). *SMC OK* describes whether the tool is able to operate on programs in which the instruction space changes dynamically (e.g., dynamic linking). *Bugs OK* describes whether the tool is robust in the face of application errors such as memory addressing errors or divide-by-zero. Y* indicates that checking can be turned on but this degrades performance.

Technology describes the general implementation techniques used in the tool. The implementations appearing in this table are:

- Hardware emulation including both dedicated hardware and microcode (emu).

- The “obvious” implementation, a decode and dispatch interpreter (ddi).

- Pre-decode interpreters (pdi) that pre-convert to a quick-to-decode intermediate representation. The IR can be many forms; a particularly fast, simple, and common form is threaded code (tci).

- Static cross-compilation (scc) decodes and dispatches instructions during cross-compilation, essentially avoiding all run-time dispatch costs. As a special case, where the host and target are the same, the static compiler merely annotates or augments (aug) the original program.

- Dynamic cross-compilation (dcc) is performed at run-time and can work with any code including dynamically-linked libraries.

- Fast-forwarding (ffw) is a variation on memoization, a well known optimization technique commonly applied to functional programming languages. Fast-forwarding caches simulator results and reuses them to accelerate subsequent simulator execution.

- Where interpreter specifics are unavailable the tool is listed as using a general interpreter (gi).

As the above table shows, instruction-level simulators have used a number of different technologies. Logically, all simulators must decode, dispatch, and execute each instruction in turn, but few of them are written using this straight-forward implementation. Instead, most tools use alternate strategies to perform these operations more quickly. One of the fastest alternatives is to use hardware or microcode emulation of the target machine. For example, ATUM uses special microcoded versions of load/store instructions to perform cache simulation without modifying the target executable [2]. WWT uses existing hardware features (e.g., ECC bit on memory) or special purpose hardware (i.e., the Typhoon-0 hardware [63]) to simulate cache coherent shared memory.

This approach can execute programs quickly, but building hardware is expensive and time consuming, and existing hardware lacks the flexibility to handle many kinds of simulation.

A common software approach to accelerating simulation is to translate the target program into a faster form, perhaps even a form that can run directly on host hardware. Several of the above tools pre-decode target instructions—either when a target program is loaded, while it executes, or after execution (e.g., FX!32) for use the next time it is run—to speedup the decode and dispatch loop. This commonly involves converting an instruction opcode into a pointer to a function that simulates that opcode, and listing the register and literal operands in byte or word aligned fields. Cross-compilation and, in the case where the host and target instruction sets are nearly identical, augmentation, go a step further by compiling the target and simulator into a single executable. This technique executes target instructions using equivalent host instructions and performs simulator actions (e.g., counting cache misses) or other behavior (e.g., run-time error checking) with extra code inserted among the translated target instructions. WWT[62], Purify[32], and qpt[38] directly execute code from target executables augmented with code snippets that simulate distributed shared memory, detect various run-time errors, and profile/trace execution respectively.

Tools like Shade[17] use a more dynamic approach, in which target instructions are translated into host code as they are first encountered in the dynamic instruction stream. Dynamic cross-compilation allows a tool to only translate instructions that actually execute, thereby reducing the possible code explosion caused by translating an entire executable. This approach also allows a cross-compilation based system to change its behavior, while a simulation is running. For exam-

ple, the simulator can collect minimal statistics for the first 10,000,000 instructions and then collect more detailed information, once the program enters its main phase of computation.

SimpleScalar [13], RSIM [54], and MXS [34] are contemporary simulators for out-of-order processors. They all execute thousands of host cycles per simulated cycle. SimpleScalar, one of the fastest out-of-order simulators using traditional technology, simulates a MIPS-like architecture and runs target programs with a 4,000 times slowdown [13]. RSIM emulates a multi-processor with a SPARC-like architecture and typically simulates 10,000–15,000 instructions per second on a SUN Ultra 1/140 workstation [54]. MXS is the detailed, dynamic execution processor simulator from SimOS. It executes approximately 20,000 instructions per second, with a "several thousand times slowdown [34]."

Cross-compilation has not previously been applied to simulators of out-of-order micro-architectures, because it is difficult to reconcile the dynamic behavior of an out-of-order execution engine with statically scheduled cross-compiled instructions. These simulators are slow. Often suffering thousands of times slowdown relative to program execution on native hardware. But even with cross-compilation, out-of-order simulators would still be slow because most of their time is spent simulating the complex micro-architecture, not the functional behavior of instructions. Fast-forwarding is my new technique for accelerating the simulation of complex micro-architecture, and combined with cross-compilation produced an out-of-order simulator with only a 190–360 times slowdown.

2.1.2. Trace Sampling

My simulator memoization technique improves simulator performance by trading memory for speed. Another common approach is to trade accuracy for speed. This is the idea behind trace sampling. A simulator that uses trace sampling only simulates a fraction of a program's entire execution trace. The behavior of the entire program execution is extrapolated from the data collected on the much smaller sample. Just as a memoizing simulator must be designed carefully to get the best performance for the least increase in memory, trace sampling simulators are carefully designed to get both good performance and reasonably accurate results.

Laha, Patel, and Iyer first applied statistical sampling techniques to program address traces for the purpose of cache simulation in 1988 [37]. They showed at that time that 35 randomly selected samples containing just tens-of-thousands of instructions each were sufficient to approximate the distribution of cache misses for an entire target program. This approach avoided the need for traces containing data from millions of memory references, thereby saving disk space to store the traces and allowing simulation results to be computed in much less time. By sampling throughout the execution of a program, cache simulation results were significantly more accurate than naive measurements taken only at the beginning of a program trace. Trace sampling has since been the subject of much research and further refinements, including its application to the simulation of other processor components [20][56][40]. The next few paragraphs describe how trace sampling has evolved to handle more modern micro-architecture constraints.

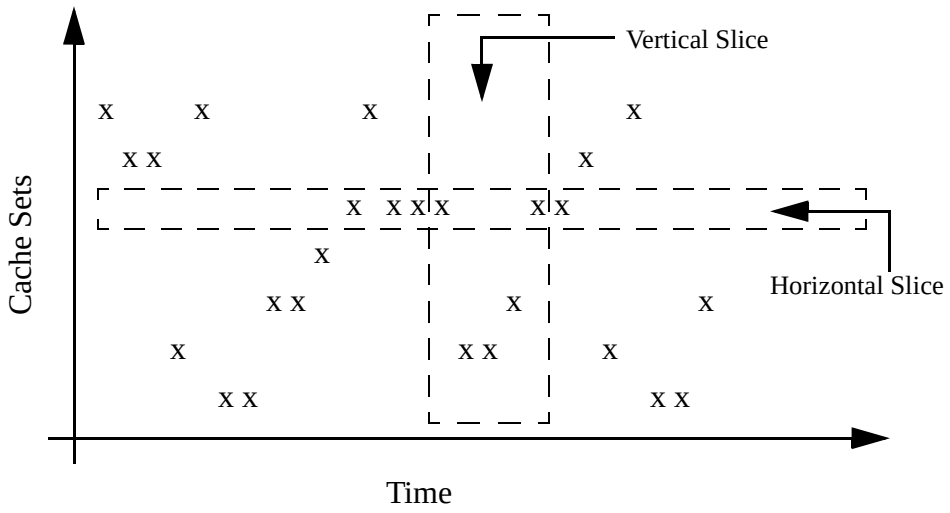

FIGURE 2.1: Sampling as vertical and horizontal time-space slices. This figure (taken from [36]) shows a time-space diagram of a simulation with a very short trace. An observation in set sampling is the cache performance of references that occur in a horizontal slice of this figure. An observation in time sampling is the cache performance of references in a vertical slice.

Since trace sampling was introduced, cache sizes and hardware complexity have increased. The original techniques described by Laha, Patel, and Iyer [37] are no longer sufficient to efficiently approximate cache and processor behavior. Kessler, Hill, and Wood compared several trace sampling techniques for their ability to meet a "10% sampling goal"—i.e., could they estimate a trace's true misses per instruction with ≤10% relative error, using ≤10% of the trace, at least 90% of the time—in the presence of multi-megabyte caches [36]. They focused on two basic strategies—*set sampling* and *time sampling*—illustrated in figure 2.1. Their results showed trace sampling using a set sampling strategy met the 10% sampling goal. However, set sampling is not applicable in all situations, such as for caches that have time-dependent behavior (e.g., prefetching) or when structures are shared by many sets (e.g., write buffers).

Time sampling can be used where set sampling cannot, but the results in [36] showed that time sampling techniques did not meet the 10% sampling goal when applied to multi-megabyte caches in the multi-billion instruction traces studied. The main problem is simulation error caused by cold cache misses—called the *cold-start bias*—because changes to the cache state could not be determined between samples. Several existing techniques, each with different strategies for reducing cold-start bias were compared: COLD—each sample starts with a cold cache, HALF—initialize the cache during the first half of each sample and only collect data from the second half, PRIME—only simulate the cache on accesses to sets that are initialized by earlier references in the sample [37][70], STITCH—reuse the cache state from the end of the previous sample [1], and INITMR—estimate the fraction of cold-start misses that would have missed even if the cache state were known [80]. The results showed that, for the given traces, INITMR was the most effective at reducing cold-start bias, although HALF also performed well. But large observation intervals were still needed, in the range of tens-of-millions to hundreds-of-millions of instructions for each observation. The result was that time sampling did not meet the 10% sampling goal.

Partial sampling is another trace sampling technique, but uses non-clustered sampling, and limits the effect of unknown states—i.e., the cold-start bias. Vengroff, Simpson, and Gao described a technique for non-clustered trace sampling for simulating multiple cache designs [74]. They eliminated cold-start bias and allowed individual cache references to be sampled in isolation by constructing a compact record for each sample observation, with enough information to determine, with absolute certainty, whether the reference would hit or miss in a simulated cache. The same records could be used simultaneously to simulate many different cache designs. Since their

only results were for relatively small cache sizes, it is uncertain how well their technique scales to multi-megabyte caches and other sources of cold-start bias. Also, their technique does not work for out-of-order processors, because loads and stores can be re-ordered by these processors in response to changes in cache behavior.

2.1.3. Compiler Optimization for Verilog and VHDL

Willis and Siewiorek described a variety of compiler optimizations used in their experimental simulator—Auriga—to reduce run-time complexity and promote concurrency in the simulation of VHDL models [79]. Their *temporal analysis* used a data-flow analysis adapted to the event driven nature of VHDL to find patterns in the sequence of values assumed by signals (e.g., the regular on/off pattern of a clock signal) and the set of values that could get bound to signals and variables during simulation. Although the authors don't make the connection, this analysis and the grammars they use to encode the sets of possible signal and variable values is similar to techniques used for off-line partial evaluation. Off-line partial evaluators use data-flow analysis to find the set of values that can be passed as arguments to each function, and these sets of values are sometimes encoded as grammars [76].

Willis and Siewiorek used other analyses as well: *waveform propagation* propagates the information about signal patterns found by temporal analysis from output ports to all input ports that may receive those values, and finds the output ports whose data is never used; *concurrent evaluation* determined which parts of the model could be simulated in parallel; *input desensitization* finds events that can be ignored during simulation. Input desensitization is an extension of earlier

clock suppression techniques [61], but not as selective as later techniques in [55]. With these analyses and several common compiler optimizations—e.g., function-inlining, loop unrolling, and constant propagation—optimized simulators executed up to 35 times fewer machine instructions, compared to an early, un-optimized VHDL simulator.

Park and Park optimized VHDL simulation by suppressing *insensitive events* in the simulation of both sequential and combinational circuits [55]. Typical VHDL simulation uses events, sorted by time, to trigger the evaluation of parts of a circuit model. Due to programming styles, the VHDL language, and synthesis tools, many parts of a VHDL program are triggered by events that cause no visible effect. By analyzing VHDL programs to find the set of insensitive events for each part of the program, and applying source-to-source translations to suppress these events, the authors achieved an almost two times speedup in simulation. Earlier work by Devadas et al., suppressed events in synchronous circuits only [25], and Razden et al., eliminated unnecessary activity caused by clocks [61]. But the analysis and optimization in this later work on insensitive event suppression subsumed the benefits uncovered by these earlier techniques.

2.2. Relevant Programming Language Techniques

This section gives an overview of the programming language techniques I apply to instruction-level simulation. Partial evaluation (PE) and run-time code generation (RTCG) are defined in Sections 2.2.1 and 2.2.2 respectively, along with several examples of their use and implementation. Later chapters describe how I apply PE (and potently RTCG) to processor simulation. Section 2.2.3 discusses special purpose languages for describing instruction set architectures.

2.2.1. Partial Evaluation

My fast-forwarding optimization is a variation of memoization, but instead of only caching function return values it caches partially evaluated simulator code. The idea is that cached code skips operations that behave the same as when the code was cached, and only executes operations that do something different. The operations that do something different are called *dynamic*. Techniques from partial evaluation are used to select the dynamic operations that are memoized by fast-forwarding.

2.2.1.1. WHAT IS PARTIAL EVALUATION?

Consider the program `p` that accepts input `[in1, in2]`. The effect of partial evaluation is to specialize `p` together with part of its input `in1` to produce a new program p_{in1} that can then be called with input `in2` to compute the same result as ⟦p⟧ `[in1, in2]` (i.e., p called on input `[in1, in2]`). The correctness of p_{in1} can be described for all `in2` by the equation ⟦p⟧ `[in1, in2]` = ⟦p_{in1}⟧ `in2`. The technique is to pre-compute all the expressions in `p` that rely only on values in `in1`, unfolding recursive calls, unrolling loops, etc. The chief motivation for partial evaluation is speed: the specialized program p_{in1} is often faster that `p`, and this can be advantageous when `in2` changes more frequently than `in1`.

ON-LINE AND OFF-LINE PARTIAL EVALUATION. On-line partial evaluation works in one pass. It specializes program parts as they are encountered, making decisions “on the fly” using only (and all) the data available. On-line partial evaluation can often produce more efficient code than off-line PE, because it uses the values of static data to help drive specialization, whereas off-line PE is

```
/* Select between static value S and dynamic value D
** using the static value given in x. */

int F(int x, int S, int D)
{
    if(x == 0) return S;
    else return D;
}
```

FIGURE 2.2: On-line vs. Off-line partial evaluation. Function F tests the value of its first argument to decide which of its next two argument values to return. Suppose that for some call to F, the arguments x and S are static (i.e., known at specialization time) and D is dynamic. An on-line partial evaluator would know the value of x and, if its value was 0, could reduce the call to its result—S—and remove the call to F at specialization time. An off-line partial evaluator would not be aware of the value of x when deciding how to specialize this code. So the call to F would not be eliminated with off-line PE.

driven by annotations derived without using static data values. Although on-line partial evaluators produce more specialized code, it is more difficult to guarantee their termination. Figure 2.2 demonstrates how more specialization can be performed with static data values than without them.

Off-line partial evaluation uses two passes: an initial pass to annotate the subject program, then another pass to specialize the program using these annotations. The first pass performs *binding-time analysis*, described below, and generates annotations for the later specialization pass. A typical set of annotations are: *evaluate* statically computable code, *unfold at specialization time* static function calls, *generate residual code* for expressions containing dynamic components, and *generate residual function calls* for functions containing some dynamic code. The specialization pass then uses these annotation to produce a specialized program.

BINDING-TIME ANALYSIS (BTA). Binding-time analysis is used by off-line partial evaluators to divide a subject program into expressions that are *static*, meaning they depend only on information known at specialization time (e.g., the input data `in1` that `p` is specialized for), and expressions that are *dynamic*, because they depend on some information that is only available at run time. In some versions of BTA, the binding-time information for each function can have different divisions of a function's arguments for different call instances—called *poly-variant division*—or BTA may require all calls to the same function to have the same division—*mono-variant division*.

The division of a program into static and dynamic parts can be computed, like other abstract interpretation problems, using fixed point iteration. The idea is to interpret the program, but use binding times instead of actual data values. For example, if a variable `x` is set to the static literal value `5`, then the abstract interpreter only records that `x` is static and ignores the value `5`. When a program point is reached from two different control flow paths, binding time data from the two paths is merged. This abstract interpretation continues, perhaps iterating multiple times over code contained in loops, until the binding times of all variables at every program point stops changing—i.e., it reaches a fixed-point. These final binding-times represent a division of the program into static and dynamic components. Fixed-point iteration has potential problems with termination—especially when using poly-variant divisions—but these problems can be overcome [35].

2.2.1.2. EXAMPLES OF PARTIAL EVALUATION

The potential benefits of partial evaluation have been studied for some real world applications that demonstrate the ability of partial evaluation technology to handle more than simple toy

benchmarks. Ray tracing is an algorithm in computer graphics that calculates every pixel in a rendered image by back-tracing the paths imagined light rays would take from each image pixel back to any light source, reflecting off objects in the scene. Partial evaluation was applied to ray-tracing by Mogensen [49], and again by Andersen who optimized an already efficient ray tracing algorithm to achieve a 1.3 to 3.3 times speedup depending on the scene and degree of specialization [4]. Baier, Glück, and Zöchling describe the effect of partial evaluation applied to numerical programs in Fortran [7]. By taking advantage of the statically determinable control flow common in these programs, their off-line partial evaluator for Fortran 77 achieved speedups on three numeric algorithms: Fast fourier transform (3.1-4.0x speedup), N-body problem (1.3-1.4x), and cubic splines interpolation (4.0-6.0x).

Lars Ole Anderson developed the first partial evaluator for a significant subset of the C programming language [3]. Anderson's partial evaluator—called C-Mix—uses off-line partial evaluation with poly-variant divisions. C-Mix handles such issues as structured data stored in C structures and unions, C pointers (using an inter-procedural alias analysis to determine the set of objects that may be pointed to by each pointer object), and separate compilation by dealing with the effects of external functions and data on C-Mix's analyses. C-Mix is self-applicable (i.e., can specialize itself) and achieved significant speedups in several toy benchmarks.

Tempo [18] is a modern partial evaluator for the C programming language, primarily targeted at the specialization of system software. It uses more mature analyses than C-Mix, and has been applied to some real applications. Tempo has been shown effective at optimizing file system

access in the Synthetix kernel [58], and in optimizing the Sun RPC protocol [50]. A key advantage of this partial evaluation system is its ability to both specialize programs at compile time and at run-time. Run-time specialization (discussed in section 2.2.2) can make use information that is only available at run-time, to produce more highly specialized code.

Muller, Volanschi, and Marlet applied partial evaluation to the Sun commercial RPC protocol [50]. Remote Procedure Call (RPC) is a protocol that makes a remote procedure look like a local one, and primarily involves the marshaling/unmarshaling (i.e., encoding/decoding) of call arguments and managing the exchange of messages through the network. Because the Sun RPC protocol is implemented in a highly generic way, it offered multiple opportunities for specialization. Using Tempo's partial evaluation and making small changes in the RPC code, they achieved a 3.75 times speedup in client encoding procedures, which translates to a 1.35 times speedup on complete remote procedural calls. In their conclusions, the authors stated that partial evaluation can be applied to realistic programs with non-trivial results, but recognized that good knowledge of the application domain is needed to find the opportunities for specialization.

Pu, Massalin, and Ioannidis introduced the Synthesis operating system kernel in 1988 [59]. One of the important contributions of Synthesis was its generation of specialized—thus short and fast—versions of kernel routines for specific situations. For example, the `open` system call synthesized new versions of the `read` and `write` system calls, specialized for accessing a particular file or device. These specialized routines were optimized by *factoring invariants* to bypass redundant computations, *collapsing layers* to eliminate unnecessary procedure calls and context

switches, and *executable data structures* that shorten data structure traversal time. Synthesis' optimizations, especially collapsing layers, allowed a more layered operating system implementation, with a high-level interface, to run efficiently. Their results showed that specialized kernel routines could significantly improve system call performance, while simultaneously allowing a high-level operating system interface.

Synthesis is an early example of program specialization being applied to an operating system kernel, although it did not use actual partial evaluation. Instead, kernel routines that had been specialized by hand were strung together using threaded code. Subsequent work on Synthesis (renamed Synthetix) used partial evaluation to specialize the operating system interface. In [57], domain-specific microlanguages allowed clients to tell Synthetix how base operating system functionality would be used. Then partial evaluation was used to construct specialized kernel routines from code written in a microlanguage, which optimized for the predicted sequences of operations. These specialized kernel routines significantly improved system performance when the actual program behavior matched the behavior predicted by mircolanguage code.

Partial evaluation has also been applied to hardware design. Wang and Lewis described their compiler—PECompiler—which used partial evaluation to automatically design field-programmable custom compute machines (FCCMs) to accelerate the execution of the program being compiled [77]. PECompiler optimizes programs written in a subset of C using partial evaluation (and several other techniques) to: 1) generate a residual program given part of the program's input data, then 2) collect the remaining computations into a data dependence graph (DDG), from which a

VHDL description of hardware functional units can be generated. The result is a compiled program that uses a special purpose VLIW co-processor, implemented on FPGAs, to accelerate the computationally intensive core of a source program. This technique was demonstrated by compiling a program that implemented a timing simulation of digital circuits. In experiments specializing this program for several different input circuits, the automatically generated FCCMs were comparable to hand designed FCCMs that accelerated the simulation of the same digital circuits.

2.2.2. Run-Time Code Generation

With more information about a program's data values, program code can be more highly specialized and run faster. Run-time code generation (RTCG) uses run-time data values to produce specialized code at run-time. This is essentially how fast-forwarding works: Simulator code is specialized with run-time data values at run-time, using annotations computed at compile-time. The difference is that RTCG systems generate machine instructions at run-time. So far, my implementations of fast-forwarding interpret memoized data rather than memoizing and executing native machine instructions.

A key issue is the efficiency of a run-time specializer, since spending too much time in the specializer can negate any benefit from executing the specialized code. This efficiency must be balanced with the need to generate good specialized code. A common technique, used in the examples below, is to compute code templates at compile time, then instantiate them at run-time by making a copy of the needed templates and filling in the holes with run-time static data. Other

issues include managing run-time generated code, and specifying which parts of the subject program should be run-time generated.

Work done at the University of Washington includes a technique for generating code templates at compile time that are later used in dynamic code generation, based on annotations provided by the programmer (Auslander et al. [6]). Annotations added to C source code identify exactly which regions of code should be generated at run-time, and which inputs to consider as run-time static when specializing the code. Although other systems provide more automatic approaches to creating dynamic code templates, this work had the advantage of handling the full functionality of C. Subsequent work at the University of Washington continued to use programer annotation in C programs to drive specialization, but automated more of the specialization process [30]. Programmer annotations still identified the run-time data to use as static input for run-time specialization, but the choice of regions to specialize and construction of dynamic code templates for those regions was handled automatically by the compiler. Run-time generated code was then cached automatically by the run-time specializer.

At the University of Rennes / IRISA, Consel and Nöel used off-line partial evaluation to automatically generate efficient templates for run-time generated code from C programs [19]. Binding-time analysis determined the static and dynamic parts of a program, then action analysis determined how each piece of code should be specialized (e.g., *reduced* (removed) from specialized code or *rebuilt* to contain only the dynamic components of computation). After BTA, performed at compile time, the values of run-time static data are still unknown, hence the effects of

loop unrolling or specializing static branches is still unknown. This means the structure of the dynamic code templates cannot be fully determined. To solve this problem the authors introduced *specialization grammars* that represent a safe approximation of the set of possible code layouts for specialized code, represented by a grammar. C code templates were then generated for the specializations described by the specialization grammar. The templates generated for a given source function were all generated as C code within a single C function that was then compiled and optimized by gcc. The compiled templates were extracted from the gcc output, and hence benefit from the optimizations in a real compiler, including optimizations between templates. Finally run-time specializers were generated in C code that emitted sequences of binary code templates, using run-time static data to control which templates were generated, and to fill in the template holes with run-time static values. The advantages of this technique over previous RTCG systems were its use of partial evaluation to generate templates automatically using program analysis, and its ability to efficiently optimize the run-time generated code across template boundaries.

Lee and Leone described a low cost RTCG system for a pure subset of ML that did not use templates [41]. Their system—Fabius—used currying to naturally annotate run-time static data without adding any new syntax to the language for the purpose of annotation. For example, a function $\texttt{f}(S_1, D_1, S_2, D_2)$, with infrequently changing arguments S_1, S_2 and frequently changing arguments D_1, D_2, could be redefined as $\texttt{f}(S_1, S_2)(D_1, D_2)$. The call $\texttt{f}(S_1, S_2)$ would then return the function `f`, specialized for the run-time static data S_1 and S_2. Fabius did not use code templates to generate specialized code. Instead, it used a technique the authors call *deferred compilation* [42] that works by specializing a program specializer to the given ML program. These

specialized run-time specializers were able to emit dynamic instructions at an average cost of 4.8-6.2 run-time specializer instructions for each dynamic instruction emitted, when used in simple benchmarks.

Some important issues in controlling RTCG are: specifying the parts of a program to specialize, when to use specialized code, and which data values to consider run-time static. Tempo [19]—a partial evaluator for C programs capable of both compile-time and run-time specialization—uses a separate description of the specialization context, and the run-time specialized code Tempo generates must be managed entirely by the user. In the C dynamic compiler developed at the University of Washington [30], the programmer marks replacable components by directly annotating a program with syntactic extensions to C. The management of run-time specialized blocks is done automatically. In Fabius [41], run-time static data is specified by currying in ML—i.e., run-time static function arguments are identified by passing them as the first set of arguments to a curried function. The user generates specialized code by calling a function with only its first (i.e., run-time static) arguments, and manages specialized functions by explicitly storing and later using them as needed. Volanschi et al. [75] describe *specialization classes* as a way to control compile-time and run-time specialization of programs written in an extension of Java. Specialized versions of object methods are declared using specialization classes that use inheritance to extend existing classes in a program and identify the (run-time) static data used in specialization. Specialized code is automatically generated, cached, and selected for use, with only minor hints provided by a programmer as part of the specialization class definitions.

Fast-forwarding is similar to RTCG in that it automatically generates, caches, and selects specialized code, although current implementations generate interpreted actions rather machine instructions. Fast-forwarding simulators generate many different specialized copies of the simulator's code, so the size of specialized code is critical. Special techniques are used to save space and allow fast lookup of specialized code. To save space, only control flow paths that are actually executed are specialized. No code is stored for control flow paths that are not executed. Code lookup is optimized by linking each instance of specialized code to the specialized code that executes next, eliminating a hash table lookup to find the next specialized code sequence.

2.2.3. Architecture Description Languages

One of my contributions is a new special purpose programming language—called Facile—for writing instruction-level micro-architecture simulators optimized with fast-forwarding. An important component of this language is syntax that concisely describes an instruction set architecture. Several previous architecture description languages have been developed to describe instruction set architectures. Facile is loosely based on the New Jersey Machine Code Toolkit language.

The New Jersey Machine-Code Toolkit [60] is designed to help programmers write applications that manipulate machine code—e.g., assemblers, disassemblers, code generators, tracers, profilers, and debuggers. Its specification language is very general, and is suitable for describing both CISC and RISC ISAs; The authors have written specifications for the MIPS R3000, SPARC, and Intel 486 instruction sets. Specifications are built using sequences of fixed width *tokens* to form *patterns* that describe the binary encodings of instructions. RISC instruction sets, such as

MIPS R3000, and SPARC are specified with one token per instruction, but CISC ISAs need multiple tokens to encode variable width instructions. Patterns for several instructions can be specified simultaneously in tabular format, similar to the tables found in many architecture manuals [72], compacting the descriptions and reducing the potential for programmer error. Finally, *constructors* map between assembly language and machine language representations. This toolkit produces encoding procedures for each assembly instruction that represents a corresponding machine code instruction. To decode binary instructions, *matching* statements, embedded into C or Modula-3 programs, execute C or Modula-3 code for instructions that match constraints on their binary encoding. The ISA descriptions used by this toolkit are compact, needing only 127, 193, and 460 lines to describe the MIPS, SPARC, and Intel 486 instruction sets respectively.

Önder and Gupta from the University of Pittsburgh described the UPFAST system that automatically generates instruction level simulators from a special purpose architecture description language [53]. Their language—called ADL—can represent simulators ranging from a five stage in-order pipeline to an implementation of Tomosulo's algorithm (one algorithm used to implement out-of-order execution) for the MIPS ISA. From a machine description in ADL, UPFAST automatically generates an assembler, dissassembler, a cycle level simulator, and a debugger. Compared to some other architecture description languages, UPFAST's ADL is not concise, requiring over 4,500 lines to represent the relatively simple MIPS ISA. Because all the UPFAST specifications discussed are for MIPS architectures, I am uncertain whether ADL is general enough to represent other ISAs, including other RISC architectures such as recent specifications of SPARC.

I designed a language called SADL—short for the Spawn Architecture Description Language—which is the immediate predecessor of my new language, Facile. SADL descriptions were processed by the Spawn tool that automatically generated the back-end machine manipulation routines used by EEL [39]. SADL descriptions included both instruction encodings and a register-transfer level (RTL) semantic description of most instructions in an ISA. Spawn analyzed instruction encodings and semantic information written in SADL, then replaced various annotations embedded in C code with ISA specific code that performed the actions required by EEL. Operations that could be specified by these annotations include decoding a binary instruction to determine what kind of instruction it is (e.g., branch, load, etc.), determining which registers are read from and written to, and computing branch targets or retargeting existing branch instructions.

My subsequent work on Spawn and SADL incorporated information about in-order pipeline hazards into the language, and used this information to generate a micro-architecture specific instruction scheduler for EEL. The purpose of an instruction scheduler is to arrange the instructions in a program to minimize its execution time by minimizing the number of pipeline stalls. By scheduling instructions in EEL, the cost of executing added instrumentation could be reduced. Experiments with basic-block profiling instrumentation on two superscalar SPARC processors showed that instruction scheduling hid an average of 13% and 33% of instrumentation overhead in the Spec95 integer and floating-point benchmarks respectively. One of the lessons learned during my work on SADL, was that structural hazards for in-order pipelines are easy to encode in a specification language and to use for timing simulation. But specifications for more dynamic out-of-order pipelines could not be so easily encoded.

Several techniques exist to accelerate the detection of structural hazards in “in-order” pipelines and are used in instruction schedulers. *Reservation tables* are a common way to represent instruction resource usage for in-order pipelines [22]. Reservation tables list the pipeline resources on one axis and time (in cycles) on the other axis, and the table cells are filled to indicate which resources are used by an instruction in each cycle of execution. Pipeline hazards are detected by overlaying the reservation table for each instruction onto the combined tables of all previous instructions such that no resource is simultaneously in use by more than one instruction. One strategy for optimizing hazard detection—proposed by Eichenberger and Davidson in [27]—is to compress these reservation tables into smaller, but equivalent, tables that take less time to check for hazards. Gyllenhaal, Hwu, and Rau proposed another technique, where reservation tables are transformed into AND/OR-trees to more efficiently check resource constraints [31].

Müller proposed another pipeline hazard detection technique, where reservation tables were transformed into a deterministic finite automata (DFA) [51]. Starting with the reservation tables for a given in-order micro-architecture, a DFA was generated that takes as input the next instruction in an instruction sequence and returns the number of pipeline stalls that result. Using this technique, Müller reported an 18 times speedup in pipeline simulation, over simulation using unoptimized reservation tables.

I considered using a statically generated DFA to accelerate the simulation of an out-of-order pipeline, but a quick calculation shows that this is impractical. Consider an out-of-order pipeline with N patterns of instruction resource usage (i.e., every instruction exhibits one of these pat-

terns), and an instruction window that is M instructions long. The automata to statically encode all possible instruction sequences that could fit in the instruction window could need $O(N^M)$ states. Such an automaton would be extremely large, and unlikely to fit in any simulator's virtual address space. Despite this exponential size explosion, using finite state automata to accelerate processor simulation is a good idea. The solution for out-of-order processor simulation is to only generate the parts of an automata that are needed by a simulator at run-time. Fast-forwarding is my technique to do this.

CHAPTER III: Memoization Of An Out-Of-Order Processor Simulator[1]

Most simulators of out-of-order processors run programs thousands of times slower than actual hardware. But most of this time is spent repeating work performed earlier in the simulation. By applying a technique often used to optimize functional programming languages—memoization—the cost of out-of-order processor simulation can be reduced by up to an order of magnitude, with no effect on the simulator's accuracy.

FastSim Version 1 is a direct-execution simulator of a speculative, out-of-order uniprocessor with non-blocking caches. Its two primary contributions are *speculative direct-execution*, which efficiently performs the functional simulation of a program, and a variation of *memoization*, which dramatically accelerates the time-consuming simulation of an out-of-order micro-architecture.

Direct-execution simulators run machine code from a target program directly on a host processor, and use a variety of methods to interleave simulation code. This widely used technique allows functional simulation to run at near-hardware speed. Direct-execution, however, has not been previously used to simulate out-of-order processors, because of the difficulty of reconciling the fixed behavior of an executing program with the fluid behavior of a speculative out-of-order micro-architecture. FastSim solves this problem by decoupling the simulation of out-of-order exe-

1. This chapter is mostly taken from my conference paper, "Fast Out-Of-Order Processor Simulation Using Memoization" appearing in ASPLOS-VIII [65].

cution from the functional execution of instructions. With a new technique called speculative direct-execution, FastSim allows mispredicted branch paths to be executed directly, then rolled back. Without further optimization (e.g., memoization), FastSim runs 1.1–2.1 times faster then the well-known SimpleScalar out-of-order simulator, which does not use direct-execution.[1]

FastSim's primary contribution is the application of memoization—result caching—to the expensive process of simulating an out-of-order micro-architecture. Traditionally, memoization is used to optimize functional programming languages by caching function return values. Expensive computation can be avoided by returning a previously cached value, when available.

FastSim records micro-architecture configurations and the simulator actions that result from them. When a previously recorded configuration is encountered, the associated actions can be replayed at high speed until a previously unseen configuration is encountered. This memoization makes the simulator run 5–12 times faster, with no change in simulation results (e.g., cycle count). Combining direct-execution and memoization, FastSim simulates a MIPS R10000-like micro-architecture with a 190–360 times slowdown (i.e., simulation time over native benchmark execution time on the host), which is an order of magnitude faster than SimpleScalar.

1. There is no version of FastSim without direct-execution. Instead, SimpleScalar is used as a surrogate, as it simulates a comparable processor at an equivalent level of detail.

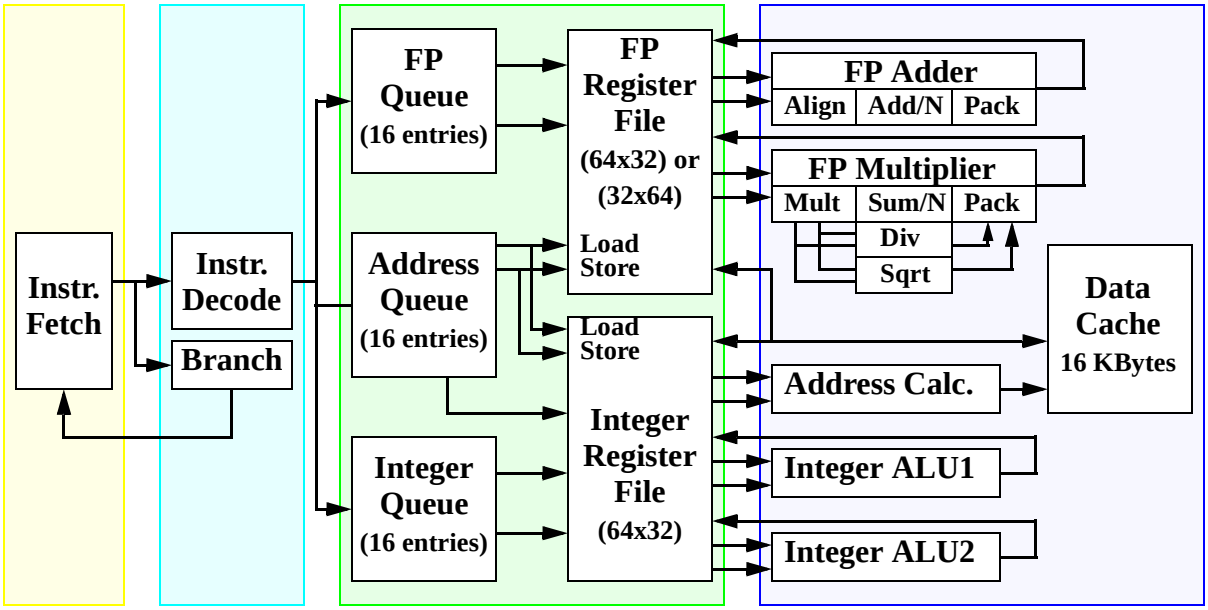

FIGURE 3.1: The dynamic execution micro-architecture modeled by FastSim.

3.1. The Structure of FastSim v.1

FastSim v.1 is a cycle-accurate, direct-execution simulator of an out-of-order uniprocessor. Like RSIM, it models a SPARC v.8 [72] instruction set running on a MIPS R10000-like [81] micro-architecture—Figure 3.1—although, unlike RSIM, FastSim only simulates a single processor. FastSim's processor model supports out-of-order instruction execution, speculative execution, and an aggressive non-blocking cache. Table 3.1 lists the processor parameters used in this simulator.

TABLE 3.1: FastSim's processor model parameters.

Decode 4 instructions per cycle.
2 integer ALUs, 2 FPUs, and 1 load/store address adder.
64 physical 32-bit integer registers, and 64 32-bit (or 32 64-bit) floating point registers.
2-bit/512-entry branch history table for branch prediction.
Speculatively execute instructions through up to 4 conditional branches.
Non-blocking L1 and L2 data caches, 8 MSHRs each.
16 KByte 2-way set associative write through L1 data cache.
1 MByte 2-way set associative write back L2 data cache.
8 byte wide, split transaction bus

Direct-execution is not easily applicable to speculative, out-of-order processor simulation. The first problem is simulating out-of-order execution using direct-execution, which is inherently in-order. As discussed in Section 3.1.1, FastSim directly executes groups of instructions in program order, then subsequently simulates their timing with respect to the out-of-order pipeline model. This is possible in FastSim, because loads, stores and other instructions do not require precise timing information to execute correctly on a uniprocessor machine.

Section 3.1.2 discusses FastSim's speculative direct-execution. Briefly, FastSim saves register and memory state at branches, then allows mispredicted branches and consequent execution paths to directly execute. Feedback from the μ-architecture simulator tells direct-execution when to restore register and memory state and restart execution at the corrected branch target. Hence mispredicted execution paths are directly executed, and data is collected for use in FastSim's micro-architecture simulator.

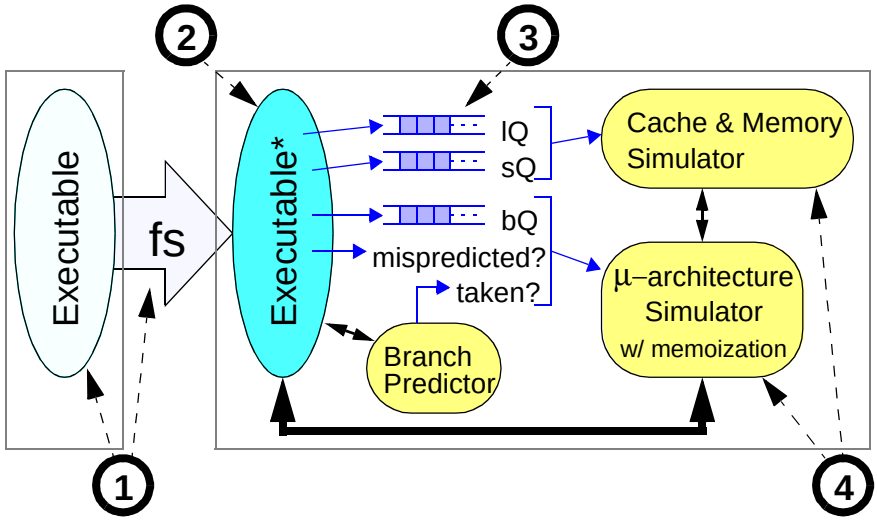

FIGURE 3.2: Overview of the FastSim simulator. 1) The tool **fs** rewrites a target executable. **2)** The edited executable directly executes target instructions. **3)** Instrumentation records information about loads, stores, and branches that is later used for μ-architecture and cache simulation. **4)** The μ-architecture and cache simulators are called periodically to simulate FastSim's processor model.

3.1.1. Direct-execution & OOO Simulation

Figure 3.2 shows the major components of the FastSim simulator. FastSim uses a binary rewriting tool (**fs**) based on the Executable Editing Library (EEL) [39] to instrument a statically linked SPARC program executable and link it with FastSim's μ-architecture and cache simulators.

The key to using direct-execution in out-or-order processor simulation is to separate functional—in order—execution of target instructions from simulation of the out-of-order pipeline. This is possible for two reasons. First, FastSim simulates a uniprocessor, hence loads and stores can be executed before their precise timing is known without affecting their result. Second, out-

of-order pipelines preserve the appearance of executing instructions in program order. FastSim exploits these properties by directly executing groups of instructions in program order, then simulating their behavior with respect to FastSim's out-of-order pipeline model.

A target executable is instrumented to record the addresses accessed by every load and store, and the target of every conditional branch and indirect jump. Load and store addresses are put in queues, called lQ and sQ respectively, for FastSim's cache simulator. Instrumentation also calls FastSim's μ-architecture simulator at every conditional branch and indirect jump (including return instructions). Since FastSim's μ-architecture simulator is invoked at every control transfer instruction having more than one possible target, a single variable records whether a branch is taken or not-taken or records the target of an indirect jump.

FastSim's μ-architecture simulator decides when the processor being modeled would have fetched, decoded, executed, and retired instructions previously executed via direct-execution. This simulator does not manipulate program data values or compute any functional results of the target program. These tasks are handled by direct-execution. When invoked, the μ-architecture simulator advances the out-of-order pipeline simulation up to fetching the current branch or indirect jump. Control flow information previously recorded for the last conditional branch or indirect jump is used to fetch instructions along the same execution path as direct-execution. When μ-architecture simulation catches up with direct-execution, the simulation is suspended and direct-execution continues to the next branch or indirect jump.

The μ-architecture simulator in turn calls FastSim's cache simulator. Queued load and store addresses along with timing information provided by the μ-architecture simulator permit accurate simulation of an aggressive non-blocking cache. The μ-architecture simulator computes the cycle at which load and store instructions are issued to the cache simulator. The cache simulator then models the cache's behavior for loads and stores, and informs the μ-architecture simulator how long each load will take to produce the requested data. Note that no program data is returned by the cache simulator, only the time taken to obtain the data.

3.1.2. Simulating Speculative Execution

In the behavior described so far, direct-execution drives FastSim's μ-architecture and cache simulators and no information flows in the other direction. Speculative execution, however, requires feedback from the μ-architecture simulator. The decision when to roll-back, following a mispredicted branch, is made by the μ-architecture simulator and must control direct-execution. On the other hand, the μ-architecture and cache simulators require data collected by direct-execution before they can run. *Speculative direct-execution* is FastSim's new technique to solve this problem. The idea is to directly execute mispredicted execution paths, while recording enough information to restore processor and memory state after a misprediction is detected by the μ-architecture. Figure 3.3 shows where instrumentation is inserted into a target executable to perform speculative direct-execution.

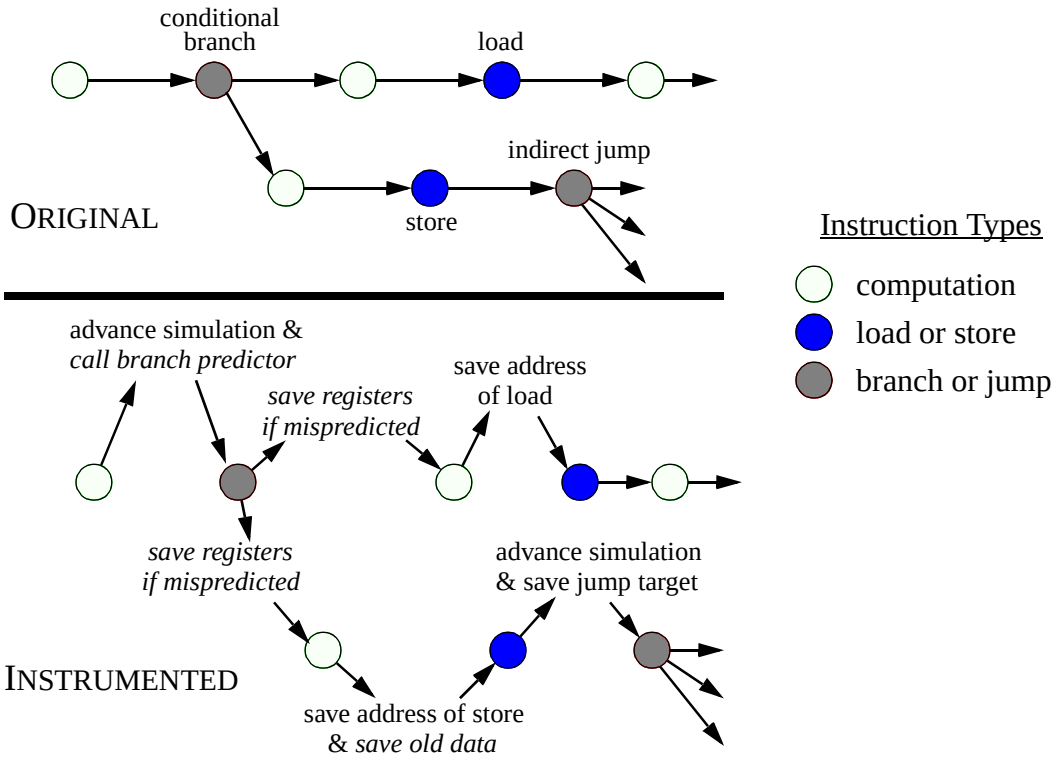

FIGURE 3.3: Instrumentation for speculative direct execution. Instrumentation is inserted among the original program instructions by **fs** to maintain FastSim's data structures, call the μ-architecture simulator, and support speculative execution. The instrumentation needed for speculative execution is shown in *italics*.

All conditional branches in a target executable are replaced with instrumentation that first calls the μ-architecture simulator, then consults FastSim's branch predictor and branches in the predicted direction. Mispredictions are detected immediately by comparing the original branch condition to the predicted branch direction. Instrumentation along the two arcs out of each branch detects mispredictions—the original branch instruction is used as part of this instrumentation. If mispredicted, all register values—integer, floating point and control registers—are saved in FastSim's bQ data structure. The bQ can hold register data for up to four mispredicted branches, which is all that is required by FastSim's simulated processor model. In the common case, where

a branch is predicted correctly, no state is saved, as the simulation never rolls-back a correctly predicted branch.

The bQ allows FastSim to restore register values when the μ-architecture simulator detects a misprediction. Other techniques are used to restore memory. To restore memory to its state before the mispredicted branch, instrumentation is added before every store instruction that records the value in memory before the store is executed (its pre-store value) and puts this data in the same sQ entry as the store's effective address. When a misprediction is detected, all pre-store memory values following the mispredicted branch are restored, in reverse order.

Using these techniques, mispredicted execution paths directly execute on a host processor, thereby collecting information needed by FastSim's μ-architecture and cache simulators. Conditional branches are executed based on the results of prediction rather than using target program values, and sufficient information is recorded to roll-back execution of a mispredicted branch. When the μ-architecture simulator discovers a misprediction, FastSim rolls-back execution of the target program by restoring host memory and registers, then continues direct-execution of the target program from the corrected target of the mispredicted branch.

3.2. Fast-Forwarding

FastSim's primary contribution is the application of memoization to micro-architecture simulation. FastSim uses a new variation on memoization—called fast-forwarding—that caches μ-

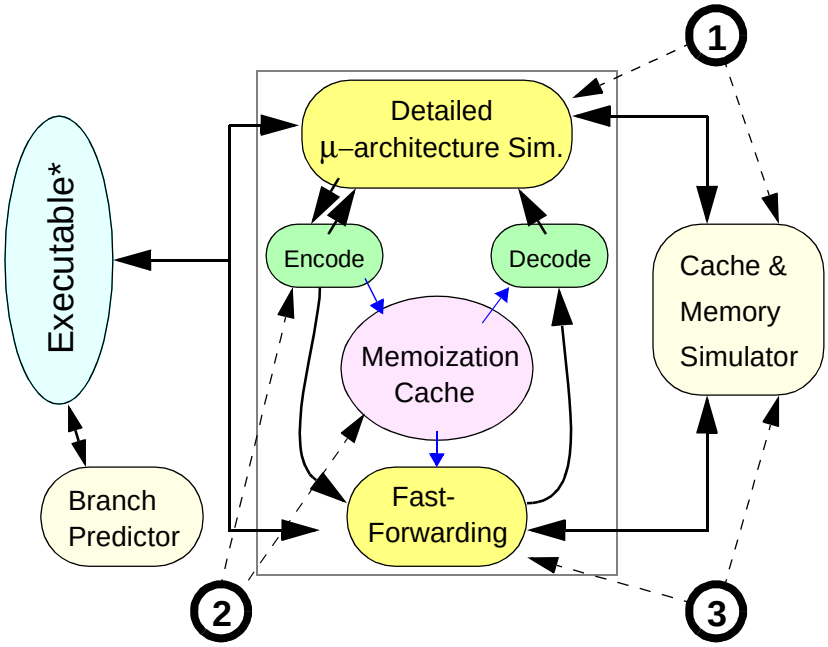

FIGURE 3.4: Memoization of FastSim's μ-architecture simulator. 1) A detailed simulator models the out-of-order micro-architecture. **2)** Actions of the detailed simulator are recorded in the memoization cache, indexed by μ-architectural configurations. **3)** Cached simulator actions are replayed for previously simulated configurations, thereby avoiding expensive detailed simulation.

architecture configurations and the resulting simulator actions for use in subsequent simulation. Figure 3.4 shows the structure of FastSim's fast-forwarding μ-architecture simulator.

The next section (3.2.1) describes the construction of FastSim's μ-architecture simulator, focusing on the techniques used to centralize simulator state and reduce the space requirements for encoding this state—necessary first steps for implementing fast-forwarding. Section 3.2.2 describes how simulator configurations (i.e., μ-architecture state) and the resulting simulator actions are further compressed and cached into FastSim's memoization cache. This memoization cache is subsequently used to fast-forward simulation. Fast-forwarding produces the same result

as detailed simulation, since μ-architecture simulator state stored in the memoization cache completely determines consequent actions of the detailed simulator. Finally, Section 3.2.3 discusses strategies for further reducing the size of the memoization cache.

3.2.1. μ-architecture Simulator

FastSim's μ-architecture simulator has been carefully designed to minimize the space needed to represent the state of its out-of-order pipeline—approximately 16 bytes plus 2 bytes per instruction in the pipeline—without reducing the complexity of its processor model. At the same time, it minimizes the amount of interaction between the μ-architecture simulator and other FastSim components (e.g., direct execution and cache simulation). These are necessary first steps to perform fast-forwarding simulation. Larger state encodings consume more space in the memoization cache, while interactions between the μ-architecture simulator and other simulator components result in more states needing to be cached.

FastSim's μ-architecture simulator is simplified by only simulating the timing of instructions, not their functional behavior. For example, values in registers and memory are not considered by the μ-architecture simulator, although the cache simulator does use addresses recorded in the lQ and sQ for load and store instructions respectively.

Another simplification is that FastSim's cache simulator is not memoized. The cache simulator is called by the μ-architecture simulator as infrequently as possible through a simple interface. The cache simulator is invoked each time a load or store is chosen from FastSim's R10000-like

address queue and begins its simulated execution. For loads, the cache simulator immediately returns the shortest interval (in cycles) before the requested data could become available, considering all other loads and stores already executing. The μ-architecture waits for this interval before again invoking the cache simulator for this load, although the cache simulator may be called in the meantime to handle other loads and stores. These calls to the cache simulator either return that data is now available or return a new interval for the μ-architecture to wait. A common example is a load that first misses in the L1 cache (usually a 6 cycle delay), then misses in the L2 cache resulting in an additional delay depending on the current state of the cache and memory bus. With this interface, the μ-architecture simulator is oblivious to the internal workings of its associated non-blocking cache simulator.

FastSim's μ-architecture simulator is built around one central data structure, the iQ, which contains one entry for every instruction currently in the out-of-order pipeline. Between simulated cycles, the iQ contains the entire configuration of the μ-architecture simulator, which can be used to index into FastSim's cache of memoized actions. The iQ is only an abstraction in FastSim's μ-architecture simulator used to centralize simulator state. It can be easily adapted to model a variety of pipeline designs.

Entries remain in the iQ from the time an instruction is fetched until it is retired. The iQ records an instruction's address—from which the instruction itself can be looked up—and a small amount of additional state information. This per-instruction state information identifies in which pipeline stage an instruction resides and the minimum number of cycles before this stage might

change. For example, an integer divide instruction may be executing—in the **execute** stage—with up to 34 cycles before it finishes executing and can be retired.

At every simulated cycle, FastSim's μ-architecture simulator makes a complete pass over instructions in the iQ, in program order, from oldest to newest. Retired instructions are removed, state information for each instruction is updated for one cycle of execution, and new instructions are fetched into the queue. Most implementation constraints in FastSim's μ-architecture model can be implemented with simple counters. One constraint is that R10000's integer instruction queue (see Figure 3.1) holds at most 16 instructions. FastSim counts the number of integer instructions already in the **queue** stage before allowing later integer instructions to move into this stage. Similarly, a simple counter limits the pipeline to at most four speculative branches. Since these kinds of constraints are recomputed every cycle, they are not part of the μ-architecture state carried between cycles.

Other constraints are more complex, but can still be implemented without explicit state information. Consider the R10000 register renaming scheme. FastSim recomputes register renaming information every cycle. This is possible, since the actual map of logical to physical registers does not affect the simulated time. The only consideration is the number of physical registers required to hold all output values of enqueued and executing instructions. FastSim builds up a new logical to physical register map every cycle, which models the physical register limitation of an R10000 and finds all true data dependencies between instructions.

3.2.2. Memoization Cache and Fast-Forwarding

FastSim's memoization cache stores a map from μ-architecture configurations to simulator actions that result from those configurations. A μ-architecture configuration is simply a snapshot of the iQ taken between cycles. Simulator actions are events, such as calling the cache simulator for a load or store, returning to direct-execution, or updating the simulation cycle counter. In general, actions stored in the memoization cache represent the ways in which FastSim's μ-architecture simulator interacts with direct-execution and cache simulation, or updates counters, such as the simulation cycle counter. Figure 3.5 shows one possible μ-architecture configuration and some of the actions resulting from this configuration.

At the start of simulation, FastSim's memoization cache is empty. μ-architecture simulation starts by running FastSim's detailed μ-architecture simulator. Whenever the detailed simulator interacts with either direct-execution or FastSim's cache simulator, it allocates a new action, describing the interaction, in the memoization cache. These actions are linked to the most recent μ-architecture configuration, which captures the simulator state before these actions executed.

When FastSim encounters a configuration already stored in the memoization cache, it looks up and replays the associated actions rather than using the detailed (slow) μ-architecture simulator to recompute them. This process is called fast-forwarding, and it produces exactly the same results as detailed μ-architecture simulation. Actions are replayed in the same order—calling the

Addr.	Instruction	Tag1	Tag2
0x10074	**clr** %fp	done	
0x10078	**ld** [%sp + 0x40], %l0	cache	6
0x1007c	**add** %sp, 0x44, %l1	exec	1
0x10080	**sub** %sp, 0x20, %sp	queue	
0x10084	**tst** %g1	queue	
0x10088	**be** 0x10098	queue	
0x1008c	**mov** %g1, %o0	queue	
0x10098	**sethi** %hi(0x5b000), %o0	fetch	
0x1009c	**or** %o0, 0x148, %o0	fetch	
0x100a0	**call** 0x3f378	fetch	
0x100a4	**nop**	fetch	
0x3f378	**save** %sp, -96, %sp		
0x3f380	**sethi** %hi(0x75c00), %o0		
0x3f384	**call** 0x56ce8		
0x3f388	**or** %o0, 0x2e0, %o0		
0x56ce8	**save** %sp, -96, %sp		
0x56cec	**sethi** %hi(0x77000), %g1		
0x56cf0	**ld** [%g1 + 0x17c], %g1		
0x56cf4	**call** %g1		
0x56cf8	**restore**		

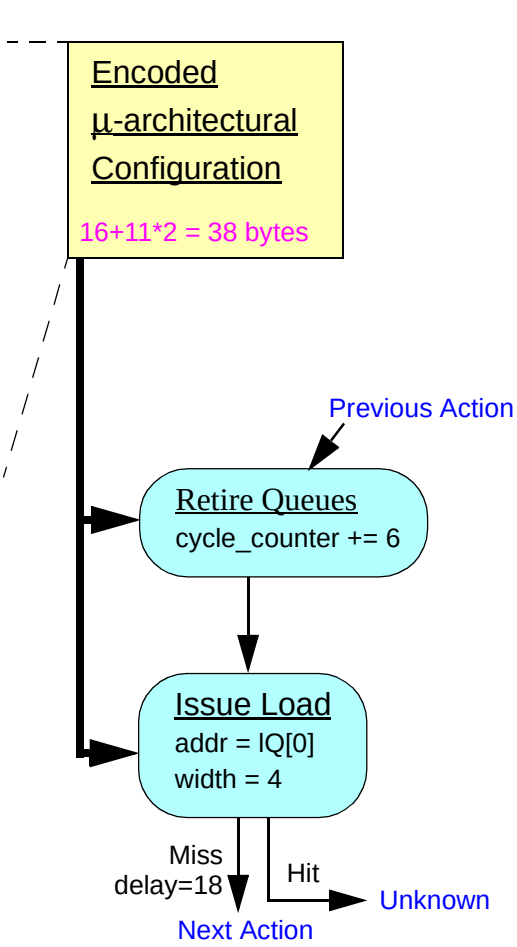

FIGURE 3.5: A μ-architecture configuration and associated actions. The instructions on the left are taken from a dynamic instruction stream. The top 11 instructions are currently in the simulated out-of-order pipeline and are considered part of this μ-architecture configuration. This configuration can be packed into 38 bytes in the memoization cache, since only the underlined data is needed to uniquely encode it. In subsequent simulation, as a result of this configuration, the μ-architecture will execute for 6 cycles, then call the cache simulator for the load at 0x10078. These actions are encoded in the memoization cache, indexed by the compressed μ-architecture configuration.

cache simulator, returning to direct-execution, and updating simulation statistics—as when they were first generated.

The only variations in μ-architecture behavior arises from different cache behavior (caused by the unpredictable internal state of the cache simulator or different values in lQ and sQ) and from different control flow in the direct-execution. These variations are checked when the actions are replayed, and previously unseen behaviors terminate fast-forwarding, so that the detailed simulator can simulate the new scenario.

Configurations stored in the memoization cache are a compressed representation of data in the iQ. This compression takes advantage of having instructions listed in program order. To encode the sequence of instruction in the iQ, FastSim only saves the starting addresses (PC and nPC) of the oldest instructions in the iQ, plus one bit per conditional branch (taken/not-taken), plus the target address of any indirect jumps. The iQ's per instruction state information can be compressed into 1.5 bytes per instruction, which subsumes the 1 bit of taken/not-taken information needed for conditional branches. Including some additional header information, this compresses a configuration to 16 bytes plus 4 bytes per indirect jump plus 1.5 bytes per instruction. New configurations are allocated at the end of a cycle in which an action was allocated. Hence at most one configuration is stored per simulated cycle, but several simulated cycles are often associated with a single configuration. Note that all interactions between the μ-architecture simulator and other FastSim components take place in the last cycle associated with a configuration because of the way configurations are allocated.

Multiple actions can be associated with a single configuration. FastSim allocated 2.9–5.7 actions per configuration while simulating the SPEC95 benchmarks. The first action following a configuration identifies the number of simulated cycles associated with that configuration. Other actions, such as calling the cache simulator or returning to direct-execution, are linked in the order in which they were produced by the detailed simulator. The last action in a chain of actions associated with a configuration is linked to the first action of the following configuration, forming an unbroken chain of actions.

Variations in behavior, caused by different values from the cache simulator or changes in control flow following a branch or indirect jump, cause the fast-forwarding simulator to choose one of several possible successor actions in the action chain. For example, there are four possible outcomes following a conditional branch in direct-execution (i.e., taken/predicted, taken/mispredicted, not-taken/predicted, and not-taken/mispredicted) and arbitrarily many return values for a load event sent to the cache simulator (i.e., possible intervals before data becomes available). If the action for a particular outcome is not in the memoization cache (e.g., the outcome has not yet occurred for the current configuration), fast-forwarding stops and detailed simulation resumes. Subsequent detailed simulation computes the μ-architecture behavior for this new outcome, and generates actions along a new branch of the action chain to handle this outcome in the future. Figure 3.6 illustrates the graph structure of the memoization cache in terms of configurations and action chains, and shows how new configurations and actions are linked into the existing graph structure to handle new outcomes.

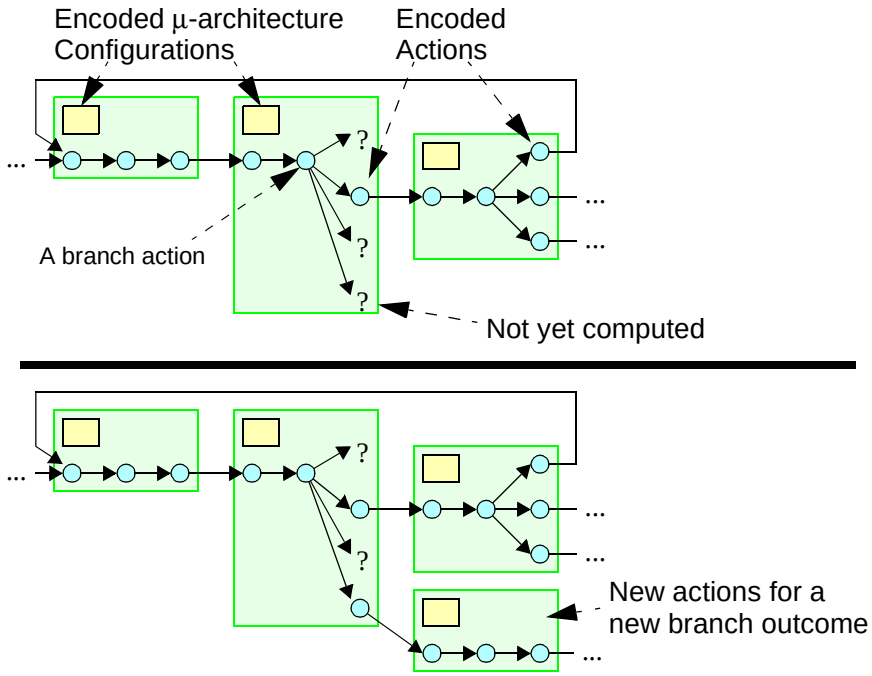

FIGURE 3.6: Action chains in the memoization cache. The upper diagram is a sample graph of configurations and actions in the memoization cache. The lower diagram shows how new configurations and actions might get linked into the existing graph when the detailed simulator is invoked to handle an alternate branch outcome.

3.2.3. Limiting Memoization Cache Size

Fast-forwarding accelerates μ-architecture simulation at the cost of increased memory consumption. Without limitation, the memoization cache can grow to hundreds of megabytes for the more complex SPEC95 benchmarks (e.g., 889MB for go, 296MB for gcc). Test results presented in the next section were collected on a host machine with 2GB of physical memory, but few people will have machines this large. Consequently several techniques are investigated for handling FastSim's memory consumption.

My first trivial memoization cache replacement policy allows unbounded growth of the memoization cache. This policy produces fast simulation times, providing the memoization cache fits in physical memory. If it does not fit in physical memory, then the OS will page (and likely thrash). A better replacement policy is to flush the memoization cache when full. This cache-flush policy is easy to implement and can limit the memoization cache to any size, but there is a performance trade-off. Whenever the cache is flushed, FastSim must use detailed (slow) μ-architecture simulation to recompute actions and configurations.

A drawback of the cache-flush policy is that useful actions are flushed along with never to be uses ones. An alternative policy, which also maintains pointers and avoids fragmentation, is to use a copying garbage collector. Only actions that were accessed since the last garbage collection are copied. This policy incurs extra overhead—the cost of copying—which one would hope could be offset by increased reuse of cached actions. A further refinement is to use a generational garbage collector, so frequently replayed actions will not be copied by the garbage collector as often—hopefully reducing garbage collection overhead.

3.3. FastSim Performance

This section describes some performance measurements of FastSim v.1 running the SPEC95 benchmarks. Experiments were run on a Sun Microsystems Ultra Enterprise E5000 with 167MHz UltraSPARC processors and 2 GBytes of physical memory. All programs, except compress, were

Benchmark	Program	SlowSim /	FastSim /	Slow / Fast
099.go	138.2	1,554.2	248.4	6.3
124.m88ksim	2.9	1,363.3	249.5	5.5
126.gcc	12.3	1,122.7	215.1	5.2
129.compress	0.3	1,304.4	218.2	6.0
130.li	8.6	1,435.6	293.5	4.9
132.ijpeg	3.3	1,837.5	199.4	9.2
134.perl	18.0	1,115.9	177.8	6.3
147.vortex	82.2	1,310.7	221.8	5.9
101.tomcatv	12.6	1,322.3	199.8	6.6
102.swim	4.5	1,460.4	191.3	7.6
103.su2cor	6.9	1,934.6	251.4	7.7
104.hydro2d	9.1	2,174.1	232.8	9.3
107.mgrid	33.3	2,569.6	215.9	11.9
110.applu	122.7	1,982.8	292.5	6.8
125.turb3d	114.1	1,992.9	254.5	7.8
141.apsi	66.8	2,758.1	357.7	7.7
145.fpppp	14.9	2,423.7	322.9	7.5
146.wave5	36.6	2,169.4	303.8	7.1

TABLE 3.2: Performance of FastSim on the SPEC95 benchmarks. "**Program**" is time (in seconds) to execute the original, un-instrumented executables. The two simulator slowdowns show how many times slower the benchmarks ran in FastSim without memoization (**SlowSim)** and with memoization (**FastSim**). The final column is the factor by which memoization improved the simulation.

run using their "test" input sets to reduce simulation time. Compress, which requires less time, used its "train" data set.

Table 3.2 shows the performance of FastSim, as compared against the original benchmarks (before they were instrumented) and against a direct-execution simulator without memoization. SlowSim is FastSim with memoization disabled—the fast-forwarding simulator was turned off and no configurations were encoded or put in the memoization cache. The table shows that memoization improves overall simulation performance by a factor of 4.9–11.9 times. Despite this dramatic speedup, the cycle counts—and all other processor statistics—generated by FastSim are identical.

Benchmark	Program cycles	Program insts.	SimpleScalar	SlowSim	FastSim Kinsts/sec.	FastSim / SimpleScalar
099.go	1.14E+10	1.64E+10		76.2	477.0	
124.m88ksim	2.78E+08	4.81E+08	58.4	121.8	665.5	11.4
126.gcc	9.27E+08	1.41E+09	47.2	102.8	536.6	11.4
129.compress	2.74E+07	4.43E+07	51.9	104.5	624.6	12.0
130.li	8.87E+08	1.24E+09		100.4	491.0	
132.ijpeg	2.61E+08	4.81E+08	50.2	80.4	740.6	14.7
134.perl	1.34E+09	1.93E+09	48.1	96.5	605.6	12.6
147.vortex	5.76E+09	1.09E+10		101.1	597.3	
101.tomcatv	9.83E+08	1.55E+09	57.9	93.0	615.7	10.6
102.swim	2.35E+08	4.23E+08	55.3	64.5	492.3	8.9
103.su2cor	5.48E+08	9.14E+08	56.1	68.7	528.8	9.4
104.hydro2d	6.28E+08	8.46E+08		42.7	399.0	
107.mgrid	2.96E+09	5.26E+09	56.5	61.4	731.1	12.9
110.applu	8.53E+09	1.51E+10		61.9	419.8	
125.turb3d	8.87E+09	1.59E+10		70.0	547.8	
141.apsi	6.28E+09	8.57E+09		46.5	358.7	
145.fpppp	1.20E+09	1.99E+09	48.9	55.1	413.6	8.5
146.wave5	2.59E+09	4.64E+09		58.4	417.3	

TABLE 3.3: FastSim vs. SimpleScalar. Program cycles and insts. are the total number of cycles and retired instructions resulting from out-of-order simulation in FastSim. Next are the average instructions retired per second by the SimpleScalar simulator, FastSim without memoization (**SlowSim**), and FastSim with memoization (**FastSim**). The last column shows FastSim's performance improvement relative to SimpleScalar.

Table 3.3 compares FastSim against the SimpleScalar out-of-order simulator [13] modeling similar processor and cache parameters. Despite their differences—e.g., SimpleScalar models a different instruction set—SimpleScalar provides a good baseline for measuring FastSim's performance and demonstrating the benefit of its techniques. With only direct-execution, FastSim runs 1.1–2.1 (mgrid–gcc) times faster than SimpleScalar. With fast-forwarding, FastSim runs 8.5–14.7 (fpppp–ijpeg) times faster than SimpleScalar.

One reason for memoization's large benefit is that FastSim was able to replay simulator actions for almost all instructions. Table 3.4 shows the fraction of instructions simulated in detail

Benchmark	Detailed (insts.)	Replay (insts.)	Detailed / Total
099.go	1.61E+07	1.64E+10	0.099%
124.m88ksim	6.49E+04	4.81E+08	0.013%
126.gcc	4.40E+06	1.41E+09	0.311%
129.compress	3.41E+04	4.42E+07	0.077%
130.li	4.17E+04	1.24E+09	0.003%
132.ijpeg	9.78E+05	4.80E+08	0.203%
134.perl	4.34E+05	1.93E+09	0.022%
147.vortex	8.37E+05	1.09E+10	0.008%
101.tomcatv	4.02E+04	1.55E+09	0.003%
102.swim	9.93E+04	4.23E+08	0.023%
103.su2cor	2.35E+05	9.14E+08	0.026%
104.hydro2d	2.41E+05	8.46E+08	0.028%
107.mgrid	6.72E+04	5.26E+09	0.001%
110.applu	1.40E+05	1.51E+10	0.001%
125.turb3d	8.75E+04	1.59E+10	0.001%
141.apsi	1.52E+05	8.57E+09	0.002%
145.fpppp	2.53E+05	1.99E+09	0.013%
146.wave5	2.39E+05	4.64E+09	0.005%

TABLE 3.4: Simulation skipped over by memoization. Instructions that FastSim simulated by fast-forwarding (Replay) and by detailed simulation (Detailed). The last column is the fraction of instructions that FastSim simulated in detail.

compared against the much larger proportion of instructions for which actions were replayed. For all benchmarks except gcc and ijpeg, FastSim used its detailed μ-architecture simulator for fewer than 0.1% of target instructions. However, the performance improvement does not appear to be directly attributed to this fraction (compare ijpeg).

Table 3.5 reports measurements of the memoization process. The first column reports size of the memoization cache. In many programs, it was manageably small. However, in five applications it grew to over one hundred megabytes. The go benchmark generated nearly 900MB of memoized data, by far the most. Fortunately, a simple cache replacement policy, discussed later, was sufficient to greatly reduce the memory requirements for simulating most benchmarks.

Benchmark	Memoization Cache (MB)	# Static Configs.	# Static Actions	Actions / Config.	Cycles / Config.	Dyn. Chain Length Avg.	Dyn. Chain Length Max.
099.go	889.4	5,096,560	14,764,742	3.5	1.5	17,300	1,882,101
124.m88ksim	4.6	26,660	89,180	3.6	1.5	190,974	592,750,035
126.gcc	296.0	1,774,016	5,353,318	3.5	1.5	5,354	1,618,693
129.compress	2.8	13,475	57,429	3.5	1.4	35,711	5,231,549
130.li	3.2	18,944	60,581	3.4	1.4	645,873	49,204,501
132.ijpeg	199.5	816,075	3,343,805	3.7	1.5	19,142	2,679,671
134.perl	142.9	559,449	3,205,519	3.6	1.6	51,189	13,495,080
147.vortex	108.6	557,362	2,037,172	3.7	1.3	259,160	32,527,035
101.tomcatv	5.6	27,191	114,445	3.9	1.4	1,934,565	619,213,774
102.swim	16.8	79,002	262,422	4.5	1.2	426,471	491,018,150
103.su2cor	32.8	156,603	642,213	4.1	1.1	178,467	182,556,421
104.hydro2d	35.5	174,422	679,767	4.5	1.2	244,809	194,389,159
107.mgrid	9.5	47,035	192,098	3.4	1.0	3,788,172	322,900,913
110.applu	19.5	94,893	375,606	4.7	1.0	7,414,106	38,010,020,845
125.turb3d	10.4	50,275	205,181	4.1	1.2	10,490,459	2,555,810,836
141.apsi	20.3	98,550	409,502	4.7	1.0	5,122,367	784,023,417
145.fpppp	25.4	127,051	460,440	3.8	1.0	272,104	27,784,740
146.wave5	38.3	180,398	752,237	4.9	1.0	1,049,836	458,444,554

TABLE 3.5: Measurement of memoization details. "**Memoization Cache"** is the total memory used to record configurations and actions. The next two columns report the static number of configurations and actions allocated. **"Actions/Config."** is the average dynamic number of actions replayed per configuration, and **"Cycles/Config."** is the dynamic number of simulation cycles simulated per configuration. The final two columns report average and maximum lengths of action chains played back without stopping to perform detailed simulation.

Table 3.5 also reports the number of actions and configurations statically generated for each program. Although the number of actions and configurations varied greatly between programs, the dynamic number of actions per configuration remains relatively consistent—between 3.4 and 4.9—for all benchmarks. This number is a measure of how much work can be directly replayed at a memoized configuration. By dividing the actions per configuration by the dynamic cycles per configuration, we get an indication of how much simulated work is performed by the μ-architecture each cycle. The average actions per cycle over all the integer benchmarks is 2.4, compared to 3.9 for floating-point benchmarks, which corresponds to this pipeline's ability to execute more instructions in parallel if there is a mix of integer and floating point operations. The final two col-

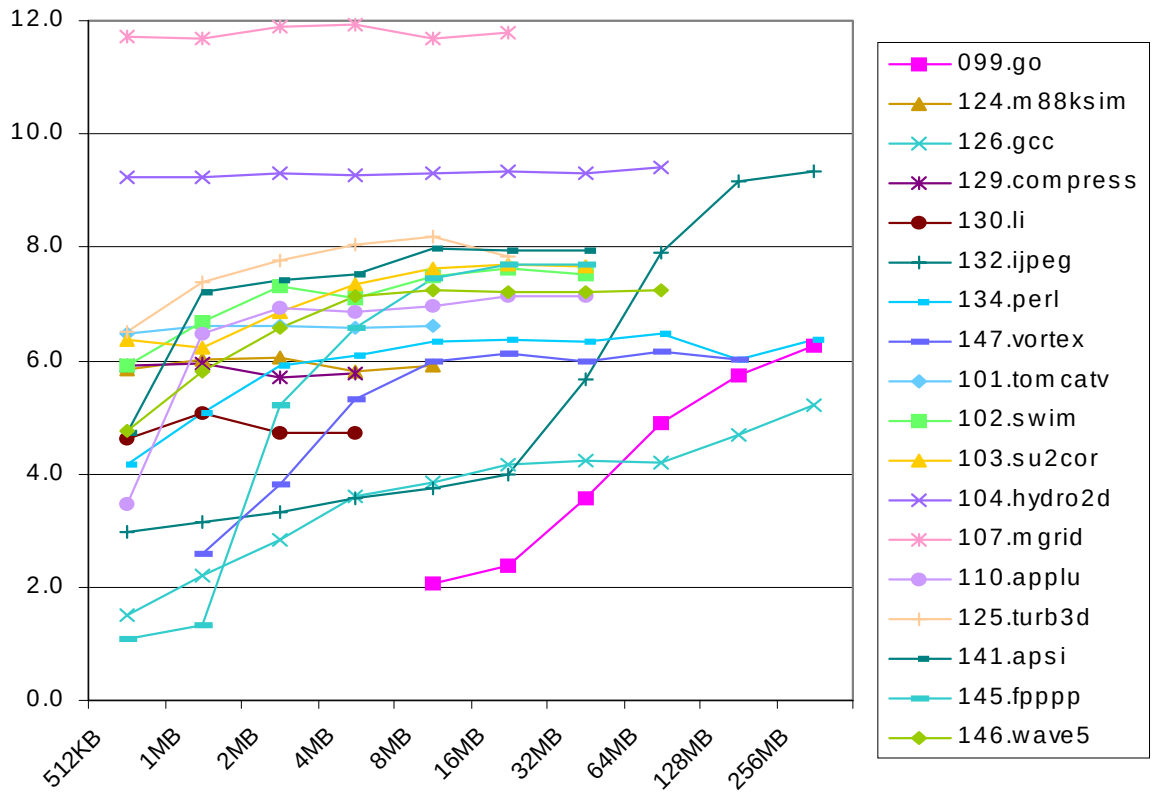

FIGURE 3.7: Memoization performance under the cache flush replacement policy. This graph shows FastSim's speedup due to memoization, when memoization cache space is limited using the cache-flush replacement policy.

umns report the average and maximum number of chained actions that fast-forwarding was able to replay without calling the detailed simulator. The large values in both these columns reflects the extremely long intervals during which only previously cached configurations were encountered.

Figure 3.7 shows the result of limiting memoization cache size using the cache-flush replacement policy. The graph shows simulator speed-up (non-memoized/memoized time) for memoization cache sizes ranging from 512KB to 256MB. Most benchmarks could tolerate an order-of-

magnitude reduction in memoization cache size with little or no impact on simulator performance. This includes the go benchmark, which naturally uses 889MB but shows no slowdown when limited to 256MB and only moderate slowdown at 64MB. A few benchmarks did not perform well with reduced cache sizes—notably ijpeg, which slowed dramatically with only moderate cache reductions—although even these benchmarks ran several times faster than simulations without memoization for all but the most restrictive cache sizes.

Experiments were also tried using a garbage collector to clean up the memoization cache, keeping only those configurations and actions that had been used since the last garbage collection. Despite the potential savings from keeping useful actions in the cache, FastSim's performance with garbage collection was nearly identical to its performance using the simple flush on full policy. Furthermore, since a copying garbage collector was used, the total memory in use during a collection could be up to twice the maximum allowed memoization cache size. Taking this into account, garbage collecting the memoization cache is almost always worse than simply flushing it. Experiments with a generational garbage collector were no better. The additional complexity—e.g., handling pointers from older generations back to younger generations—offset any savings from copying smaller portions of the cache.

The garbage collector's poor performance can be attributed to two factors: Garbage collections (or cache flushes) are infrequent and few actions survive each collection. 1-4 garbage collections or cache flushes occur when the memoization cache is sized just smaller than the maximum space used by a benchmark. For each factor of two decrease in cache size there is only a 3.8 times

increase in the number of collections on average. Infrequent collections mean that few configurations are discarded over a program's execution and that the amortized cost of regenerating them is small. Another factor is that only 18% of the memoization cache survives each garbage collection on average. Little is gained by finding and copying these actions over flushing the cache and regenerating them, when compared to the total simulation time.

3.4. Remarks on FastSim v.1

FastSim v.1 uses two, well-known, but previously un-applied techniques to greatly speed the detailed, cycle-accurate simulation of an out-of-order uniprocessor. FastSim demonstrated that direct-execution is compatible with out-of-order simulation, although the benefits are small because of the much greater cost of simulating a complex micro-architectural model.

FastSim also directly attacked this cost, using memoization to dramatically reduce the cost of detailed simulation. A key observation is that out-of-order micro-architecture configurations are often repeated and result in identical simulator behavior. By caching these configurations and their corresponding simulator actions, subsequent visits to a configuration can be replayed many times faster. This fast-forwarding optimization speeds processor simulation by a factor of 5–12 times, at the cost of increased memory consumption.

Experiments with cache replacement policies show that most benchmarks only need a fraction of the memoized data they generate over the course of simulation. A simple flush on full policy is sufficient to limit the memoization cache size without a large impact on performance. More com-

plex cache replacement policies, such as copying garbage collection, are not worth the effort, since they are difficult to implement and perform no better than the simple flush on full policy.

FastSim v.1 showed that memoization can be a highly effective optimization in processor simulation, but it has a significant drawback: the complexity of its implementation. A memoizing simulator is actually two simulators that must cooperate perfectly, so they both perform exactly the same actions in the same order. Beyond the complexity of simulating an out-of-order processor, much work was needed to implement these two cooperating simulators. The contents of a processor configuration and the set of dynamic action types had to be determined, a fast-forwarding simulator implemented to execute actions of these types, and a detailed simulator implemented to generate actions into the memoization cache. The detailed simulator was especially difficult to implement, since it also had to recover from memoization cache misses, seamlessly switching from fast-forwarded simulation back to detailed simulation.

In addition to the typical errors encountered when developing any instruction level simulator, fast-forwarding simulators introduce some new ones. One kind of memoization error involves miscommunication between the two synchronized simulators. In developing FastSim, synchronization errors could usually be detected with assertions in the simulator code, although in a few cases no assertion was broken and the simulation simply got the wrong result. A second, and more subtle kind of error, involves dynamic behavior of the simulator that was not captured by any fast-forwarding action type. For example, an early version of FastSim v.1 did not generate actions to insure loads and stores to the same address were executed in the correct order, although the

detailed simulator did order loads and stores correctly. When fast-forwarding, memory operations were executed in whichever order they executed the first time they were encountered—i.e., when actions were generated into the memoization cache—which may not be correct for the current addresses being accessed. This kind of error is more difficult to detect, since FastSim usually simulated benchmarks to completion and even computed the benchmark results correctly. These errors were often only detected by discrepancies between the simulation statistics reported with and without memoization. In many cases, meticulous (and time consuming) comparison of simulator executions with and without memoization was needed to debug the memoizing simulator.

One solution to the complexity of implementing memoizing simulators, is to automate the application of this optimization. Chapter IV introduces a new special purpose language—Facile—for writing simulators that are optimized to use memoization, and Chapter V discusses how a Facile simulator is compiled and automatically optimized to used memoization. The result is that a programmer writes a single (un-memoized) simulator, and the Facile compiler automatically generates the two synchronized simulators needed for memoization. Analyses used in this compiler insure that all dynamic simulator actions are handled by the fast-forwarding simulator and that the two simulators generated by compilation communicate perfectly. Thus reducing the complexity of implementing a memoizing simulator and removing the potential for many memoization errors.

The simulation strategy used in FastSim v.1 has another drawback, orthogonal to memoization: Instruction timing is simulated separately from instruction semantics. This separation was necessary to implement FastSim's speculative direct-execution, since instruction semantics are

simulated by directly executing target instructions on the host processor. Unfortunately, this separation makes it difficult to verify the correctness of the out-of-order processor timing simulation. Benchmarks can simulate correctly, and the simulator produce simulated execution statistics, even if the timing simulation is wrong. Section 6.1 of Chapter VI describes the implementation and performance of a memoized out-of-order processor simulator that evaluates instruction semantics out-of-order, as part of the dynamic execution pipeline simulation. This implementation lends greater credibility to the assertion that out-of-order simulation is correct.

Multi-processor simulation is not supported by FastSim, partly because of the separation of semantic simulation from timing simulation. In order to simulate multi-processor memory accesses correctly, a simulator needs to know the exact cycle in which a load or store executes. But in FastSim, this timing information is not known until later, since the timing simulator is not run until after the instructions have been directly executed. One solution to this problem, would be to execute memory operations speculatively, then rollback the simulation if the timing simulator determined an incorrect value had been loaded. These rollbacks should be infrequent, because speculation can only fail when there is a race condition. A better solution would be for the simulator to evaluate instructions at their correct time—i.e., in the order they are executed by the processor being modeled. The out-of-order simulator in Chapter VI is an example of an out-of-order simulator that executes instructions at their correct time, although multi-processor simulation is not studied in this dissertation.

CHAPTER IV: Facile—The FastSim Simulation Language

FastSim Version 2 addresses some of the problems of the original FastSim simulator by providing a flexible simulation system, rather than a single optimized simulator. Simulators are written in Facile—FastSim's new simulation language—then analyzed and automatically transformed to use memoization. The primary contribution of Facile's language design is its support for implementing memoization in detailed micro-architecture simulators. From a single Facile simulator description, the two halves of a fast-forwarding simulator—the slow/complete simulator version and a fast/residual version—are generated. They are guaranteed to cooperate correctly and implement the fast-forwarding optimization discussed in Chapter III. This approach allows programmers to concentrate on modeling processor micro-architecture, while the details of memoization are largely handled automatically.

A simulator written in Facile consists of an instruction set architecture (ISA) description and general programming constructs that model micro-architecture behavior. A Facile ISA description contains both instruction binary encodings and instruction semantics. From an ISA description the Facile compiler automatically generates code to decode instructions and simulate their behavior. ISA descriptions in Facile are compact: For example, all the user level SPARC Version 9 instruction encodings, register sets, and instruction semantics can be described in just 689 lines of Facile code. General programming constructs (e.g., functions, loops, array) are used to program a micro-architecture simulator on top of an ISA description, or within the code describing instruction

semantics. Facile is general enough to model complex micro-architecture features, such as, an out-of-order execution pipeline.

Restrictions built into the language dictate the general structure of simulator code—which constrains how a simulator is memoized—and simplify the compiler analyses needed to implement fast-forwarding. For example, in Facile there are no pointers and there is no recursion. The restriction on pointers greatly simplifies alias analysis, while the absence of recursion is necessary to efficiently switch between fast-forwarded and detailed simulation.

The structure of a simulator written in Facile directs how the simulator gets memoized. Simulators are written within the context of an implicit outer loop, provided by the run-time system. The main simulator function—written in Facile and called `main`—is called repeatedly, advancing simulation one step each time it is called. The amount of work performed in each of these steps is determined by the programmer—e.g., a step could simulate a single instruction or several instructions. Part of the data passed from one simulation step to the next (i.e., passed between calls to `main`) is used as an index into the memoization cache. All simulation work that depends only on this index can be skipped over by fast-forwarding. Facile simulators express the set of data to include in a memoization index in a natural way by encoding it as arguments to `main` (see Figure 4.1). Like conventional memoization systems, a Facile simulator is memoized by caching result data indexed by the arguments to the `main` function. Fast-forwarding is different from conventional memoization in that cached results represent residual dynamic computations instead of simple values.

```
fun main(pc, npc)
{
    // Code to simulate an instruction at the given pc
}
```

FIGURE 4.1: Arguments to main. This example shows the declaration of a Facile `main` function. Argument values for the function parameters `pc` (program counter) and `npc` (next program counter) are used to index into the memoization cache. This code fragment is taken from the complete simulator in Appendix B.

The rest of this chapter describes the Facile programming language. Syntax for specifying binary encodings and the semantics of instructions in an ISA is discussed in section 4.1. Section 4.2 explains how Facile simulator code is structured to control the way memoization is used in the resulting compiled code. Finally, section 4.3 discusses other features of the language and how they are either useful in implementing simulators or in simplifying compiler analyses. A reference manual for the Facile programming language is given in Appendix A. Most of the examples used in this chapter are excerpts from an actual simulator for the SPARC-V9 ISA, the complete text of which is given in Appendix B. The next chapter (Chapter V) continues the description of FastSim v.2 by describing how Facile is analyzed and compiled, and the run-time support needed to produce a memoizing simulator.

4.1. Architecture Description

Facile's architecture description syntax is based on my earlier work with Spawn and the Spawn Architecture Description Language (SADL) [39], which in turn is similar to the New Jersey Machine Code Toolkit architecture description language developed by Ramsey and Fernandez [60]. The NJ Machine-Code Toolkit describes instruction encodings with *tokens*, *fields*, *patterns*,

and *constructors*. Patterns associate instruction names with streams of fixed width tokens and conditions on bit fields contained in those tokens. Constructors map between literals in assembly language and their encoding in corresponding binary instructions. From descriptions in this language, the NJ Machine Code Toolkit automatically generates code that encodes and decodes binary machine instructions.

Facile also uses tokens, fields, and patterns to associate instruction names with their binary encodings. Facile's token declaration defines a fixed-width token that can appear in an instruction encoding, and also declares a set of field names for accessing sub-fields within the token. Facile's pattern declarations map mnemonic names to streams of tokens, with conditions on the values of fields in those tokens. Instead of constructors that define bidirectional mappings from assembly to machine language, Facile uses a new feature—semantic declarations—that associate instruction names (previously defined in pattern declarations) to the semantic code that simulates them. From an ISA specification built from pattern and semantic declarations, Facile automatically generates code to decode binary instructions and simulate their behavior.

Architecture descriptions in Facile are compact, reducing the opportunity for error, and are similar to some descriptions found in architecture manuals. Architecture manuals, such as the one for SPARC Version 9 [78], often use tables to describe the binary encodings of several instructions simultaneously. Facile, like Ramsey and Fernandez's architecture description language, can describe instruction encodings in similar tables (see Figure 4.2). This approach should reduce errors by making descriptions in Facile look similar to descriptions in the source reference man-

op2 [2:0]							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc	Bicc	BPr	SETHI NOP	FBPfcc	FBfcc	—

```
pat [   _trap   _bpcc   bicc    bpr     sethi   fbpfcc  fbfcc   _       ]
    = op==0 && op2 in [0..7];
```

FIGURE 4.2: Instruction Encoding Descriptions in Tables. This example shows the same instruction encoding description as it appears in The SPARC Architecture Manual (Version 9) and in the Facile simulator in Appendix B. These tables distinguish between several categories of instructions, mostly branches. Additional tables for each category of branch (not shown here) describe the encodings of individual branch instructions in the SPARC ISA.

ual, and by sharing parts of the description code among several simultaneous pattern declarations. One advantage is that an error in one instruction is likely to reoccur in several instructions, making the error easier to detect and fix. Facile also allows the semantics of several instructions to be declared simultaneously in tabular format. Although architecture manuals do not usually define semantics in this way, the resulting semantic specifications are more compact and errors are again easier to find and fix.

4.1.1. Tokens and Token Fields

Instructions are described as streams of fixed width tokens. Figure 4.3 shows how instructions may be built up from fixed width tokens for the SPARC (fixed width instruction) and Intel x86 (variable width instruction) ISAs. In architectures with fixed instruction widths—e.g., SPARC, MIPS, PowerPC, etc.—each instruction can be described by a single token. Multiple tokens are

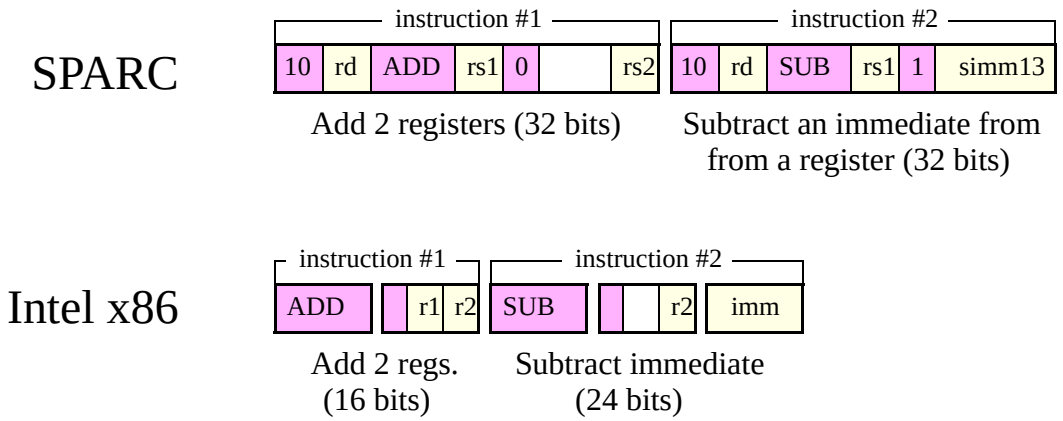

FIGURE 4.3: SPARC & Intel x86 instruction tokens. Two instructions from possible instruction streams for SPARC and Intel x86 processors—an ADD that adds two registers followed by a SUB that subtract an immediate from a register. Each instruction is divided into fixed width tokens. Since SPARC instructions are all 32 bits wide, each instruction is represented by a single token. Intel x86 instructions are variable width, so the ADD is represented with two tokens, and the SUB uses three.

needed to describe instruction set architectures with variable width instructions, such as VAX or Intel x86 processors.

Facile's `token` declaration defines a single token type and a set of names used to access contiguous bit-fields within that token. These field names are used in subsequent pattern expressions to describe instruction encodings and in semantic code to access field values from an instruction. Figure 4.4 shows the token and field declarations used to describe the SPARC Version 9 ISA. It defines a single 32 bit wide token called "`instruction`" and several field names, e.g., "`op`" in bits 30 and 31, "`op2`" in bits 22 through 24, etc. Note that token bits are numbered in big-endian order[1]—i.e., the right-most bit is number 0, and the left most bit is number $width - 1$.

```
token instruction[32] fields
    op 30:31, op2 22:24, op3 19:24, opf 5:13, rd 25:29, rs1 14:18, rs2 0:4,
    opf_cc 11:13, opf_low 5:10, cond 25:28, mcond 14:17, rcond 10:12,
    disp30 0:29, disp22 0:21, disp19 0:18, d16hi 20:21, d16lo 0:13,
    imm22 0:21, simm13 0:12, simm11 0:10, simm10 0:9, imm_asi 5:12,
    shcnt32 0:4, shcnt64 0:5, sw_trap 0:6,
    movcc2 18:18, movccr 11:12, movcc1 12:12, movcc0 11:11,
    bpccr 20:21, bpcc1 21:21, bpcc0 20:20,
    a 29:29, p 19:19, i 13:13, x 12:12;
```

FIGURE 4.4: SPARC-V9 token and field declarations.

Facile interprets an executable's binary instruction stream as a sequence of tokens starting at a given program address. This sequence is called a *token stream*. The token types that occur in a token stream are determined by matching bit values in the stream against a pattern that identifies the first one or more token types. After a simulator has decoded the tokens in the first matching pattern, the simulator normally advances the token stream to the next program address, and another pattern is matched against the next instruction.

4.1.2. Instruction "Pattern" Encodings

Facile *patterns* describe the binary bit patterns that distinguish between instructions in an ISA. These bit patterns are described as conditions on the token fields within a short sequence of tokens. For example, the SPARC `call` instruction is described as a single token with the `op` field (bits 30:31) set to the value 1. The remaining 30 bits of a `call` instruction are not constrained by this pattern, and are instead read by subsequent semantic code to interpret the instruction's target address.

1. Facile's bit ordering is an artifact of designing the language for a SPARC host processor, which also uses a big-endian bit ordering. There is some support for using little-endian bit ordering in the form of a command line argument to the Facile compiler. This command line argument would instruct the Facile compiler to interpret all bit numbers as little-endian, but it is not fully implemented at this time.

A pattern is represented in disjunctive normal form as an OR-list of AND-lists of conditions on token fields. An individual condition identifies one token at some offset from the start of the token stream, a field within that token, a literal value to compare against, and a comparison operator (e.g., ==, <=, !=, etc.) The token may be offset from the start of the token stream when it follows one or more previous tokens in a multi-token pattern. Individual conditions are satisfied, if the field value in the token stream satisfies the given test. An AND-list of conditions is satisfied only if all the conditions in the AND-list are satisfied. Finally, a single pattern can include several alternate AND-lists in an OR-list, and the entire pattern matches data in a token stream if any one of these alternate (OR'ed) clauses is satisfied.

Patterns are described in Facile with pattern expression trees, which are transformed into patterns by normalizing them into disjunctive normal form. The leaves of a pattern expression compare token fields against constant integer values, and normalize to OR-lists with one AND-list containing one condition. Pattern expression operators (`||`, `&&`, `$`) combine the OR/AND-lists of their operands to create more complex patterns.

- Single conditions are written as *field-name op value* where the operator (*op*) can be one of <, <=, ==, !=, >=, or >. A single condition is converted to a pattern (in disjunctive normal form) as an OR-list containing one AND-list containing the condition.

- Several alternate conditions on the same field can be written as *field-name* `in [`*value-list*`]`, where the value list contains explicit values or sequences of values expressed as

start`..`*end* `by` *step*. This syntax is interpreted as an OR-list with one entry for each value in the value list. Each entry in the OR-list is an AND-list containing a single condition of the form *field-name*`==`*value*. The order of entries in an OR-list corresponds to the order of values given in the value list.

- Two patterns can be OR'ed together (*pattern*$_1$ `||` *pattern*$_2$). The OR-lists of the two patterns are simply concatenated to form the resulting pattern. OR-list entries from *pattern*$_1$ are listed first, then OR-list entries from *pattern*$_2$.

- Two patterns can be AND'ed together (*pattern*$_1$ `&&` *pattern*$_2$). This is interpreted as a cross-product of the two patterns. Every AND-list in *pattern*$_1$ is concatenated with every AND-list in *pattern*$_2$, and all the resulting AND-lists are OR'ed together. Facile requires that the total widths of the token sequences encoded by the AND-lists in both *pattern*$_1$ and *pattern*$_2$ must be the same, otherwise the resulting pattern would not make sense. Entries in the resulting OR-list are ordered lexicographically based on the ordering in *pattern*$_1$ and *pattern*$_2$.

- Two patterns can also be AND'ed together with the token sequences specified by the second pattern following the tokens occurring in the first pattern (*pattern*$_1$ `$` *pattern*$_2$). As with the `&&` operator, this concatenation operator is like a lexicographically ordered cross-product of the two patterns, but the offsets of all the conditions in *pattern*$_2$ are incriminated by the sum of the token widths in *pattern*$_1$. Facile requires that the widths of the token sequences specified

```
pat reg_imm13 = (i==0 && instruction?bits(5,12)==0) || i==1;

pat [   add     and     or      xor     sub     andn    orn     xnor
        addc    mulx    umul    smul    subc    udivx   udiv    sdiv
        addcc   andcc   orcc    xorcc   subcc   andncc  orncc   xnorcc
        addccc  _       umulcc  smulcc  subccc  _       udivcc  sdivcc
        taddcc  tsubcc  taddcctv tsubcctv mulscc _      _       _
        _       _       _       _       _       sdivx   _popc   _
        wr      _       _       _       _       _       _       _
        jmpl    _retrn  _       _flush  save    restore _       _       ]
    = op==2 && op3 in [0x00..0x3f] && reg_imm13;
```

FIGURE 4.5: SPARC-V9 instruction encodings.

by all AND-lists in $pattern_1$ be the same. This is necessary in order to compute the new offsets of conditions from $pattern_2$.

- Finally, previously defined pattern names can be referred to by name in subsequent pattern expressions.

Facile's `pat` (short for pattern) declarations associate mnemonic names, like instruction names in assembly language, to patterns that are constructed with a pattern expression. Figure 4.5 shows how `pat` declarations can describe the binary encoding of several SPARC instructions. These instructions have a value of 2 in the `op` field (bits 30:31) and values in the range 0x00 through 0x3f in the `op3` field (bits 19:24). The `reg_imm13` pattern name does not correspond to any SPARC instruction, but is used here to represent a sub-expression within the pattern associated with each instruction. This sub-expression constrains the pattern for each instruction, requiring that bits 5:12 equal 0 if the instruction does not use an immediate value (i.e., if the `i` flag in bit 13 equals 0, then bits 5:12 must also be zero).

An important feature of these declarations is their ability to declare several pattern names simultaneously in a tabular format that is similar to tables found in architecture manuals. In a `pat` declaration where several names are declared simultaneously, each name is associated to a single element of the OR-list described by the pattern expression. The number of elements in the pattern OR-list must be the same as the number of names being declared. The wildcard name (_) can be used instead of a pattern name to skip an element in the OR-list, without giving it a name. The order of elements in a pattern's OR-list determines which clause maps to which name.

A potential problem with simultaneous declaration of several pattern names is that names may need to be associated with patterns that contain more than one element in their OR-list. To handle this, Facile does not expand previously declared pattern names in a pattern expression until after all the names in a simultaneous declaration have been associated with AND-lists. After the names have been associated to the corresponding parts of the given pattern, previously declared pattern names are expanded and the associated pattern expressions are re-normalized into disjunctive normal form. In Figure 4.5 above, `reg_imm13` is not expanded until after all the instruction names are bound, so each instruction name is bound to a pattern containing two OR'ed AND-lists.

Given a set of pattern declarations and a list of which patterns represent actual instructions in an ISA[1], the Facile compiler automatically generates code to decode a token stream. The generated code consists of nested switch and if statements written in C, with code at the leaves to executes the semantic behavior of each instruction. (The C code to decode target instructions is

1. Pattern names represent actual instruction names if the name appears in both a pattern (`pat`) declaration and a semantic (`sem`) declaration.

described in more detail in section 5.1.1.3 of the next chapter.) Semantic behavior is specified using semantic declarations or a Facile switch statement. Both are described in the next section.

4.1.3. Instruction Semantics

Facile has two syntactic constructs that identify patterns to use when parsing a token stream, and that associate semantic code with each pattern. These are semantic declarations, used to declare the semantic behavior of instructions in an ISA, and switch statements that are used in general simulator code to select among patterns that may or may not encode actual instructions in an ISA.

First, Facile's switch statement is similar to the switch statement in C, in that it selects between a list of cases. But Facile's switch statement selects between both integer and pattern valued cases. Normal, C-like, switch cases that test integer values are written using the `case` reserved word followed by the value[1] and a colon (i.e., `case` *N*`:`) and cases that test for patterns are written using the `pat` reserved word (i.e., `pat` *pattern*`:`). The pattern in a `pat` case can be any pattern expression, although it is typically a single name declared by an earlier `pat` declaration.

Figure 4.6 shows a contrived switch statement that selects between several possible control transfer instructions to appropriately compute a control transfer instruction's target address.

1. Facile `switch` statements can also select between more complicated datatype values (e.g., structures and tuples) and bind variable names to part or all of the condition value. See Appendix A for a complete description of the Facile `switch` statement.

```
// Assume we already know the instruction at PC is a control
// transfer instruction; Compute the branch target address.
switch(PC) {
 pat call:
    target = PC + (disp30?ext(32) << 2);
 pat jmpl:
    target = (R4[rs1] + SRC2)?cvt(stream);

// Only branch patterns remain

 pat a==0b1:         // this is an annulled branch
    if(taken) target = PC + (disp22?sext(32) << 2);
    else target = PC + 8;
 pat a==0b0:         // this branch is not annulled
    target = PC + (disp22?sext(32) << 2);
}
```

FIGURE 4.6: Pattern cases in a Facile switch statement. The variables `PC` and `target` are token streams, and `taken` is a boolean. Other names used in code associated with each case are token field names, which are implicitly declared in pattern cases where the matched token types are known and the bits represented by named fields can be uniquely determined.

Notice that cases are tested sequentially—i.e., if two pattern cases match the given token stream, then the first matching case is selected. In this example, we assume that previous code has already determined that the instruction is a `call`, `jmpl`, or branch instruction. If a token stream does not match the patterns for either direct or indirect `call` instructions (`call` and `jmpl` respectively), then this switch statement assumes it has a branch instruction, and only the `a` field need be tested to distinguish between annulled and non-annulled branches.

Token field names can be used in semantic code associated with a pattern case, so long as the bits represented by each field name can be uniquely determined. In the above example, field names like `disp30`, `disp22`, and `rs1` are used to access bit values from the first instruction in the token stream. It may not be possible to uniquely determine the bits associated with a field

name, if the field's token type occurs more than once in the matched pattern or if the token occurs at conflicting offsets in alternate conditions in the pattern. When field names cannot be uniquely determined, they are not defined and cannot be used in semantic code.

Semantic (`sem`) declarations provide another syntax for matching patterns and selecting semantic code to execute. They behave much like pattern cases in a switch statement, but their syntax is specialized for associating simulation code with pattern names for instructions. In the Facile compiler, semantic declarations are actually translated into pattern cases in a single switch statement, with one pattern case for each instruction name. All instruction semantics in an entire Facile file are collected into the same switch statement, which is then accessed using the `?exec` attribute call in Facile. Given a token stream variable—e.g., `PC`—the expression `PC?exec()` evaluates the generated switch to decode and simulate the first instruction in the token stream `PC`.

Figure 4.7 shows `sem` declarations for several SPARC instructions. A `call` instruction calculates its target address and stores the current `PC` address in integer register 15 (register `%o7` in SPARC assembly language syntax). The next five instructions compute various integer operations and are declared together because their semantic code is nearly identical. The last five instructions compute the same five operations as before, but also set the condition code register (`CCR`). Facile's built-in `?cc` attribute computes a standard set of condition code values for the attributed expression. The functions `Rx` (with one or two arguments) and `get_src2` are defined elsewhere, and are used to read integer operands and write the integer result register. As in explicit pattern

```
#define SRC2 get_src2(i,rs2,simm13)

sem call {
    nPC2 = PC + disp30?sext(32)<<2;
    Rx(15,PC?addr?ext(64));
};

sem [ add sub and or xor ]
{ Rx(rd, op(Rx(rs1),SRC2)); }
where op in [ + - & | ^ ];

sem [ addcc subcc andcc orcc xorcc ]
{ Rx(rd, op(Rx(rs1),SRC2)?cc(CCR)); }
where op in [ + - & | ^ ];
```

FIGURE 4.7: Semantic declarations for several SPARC-V9 instructions. The first `sem` declaration associates semantics with the `call` instruction pattern. The next two declarations define the semantics of several arithmetic instructions that share related behavior, by parameterizing the code on the value of `op`.

cases, semantic code in a `sem` declaration can access fields in the token stream using token field names, such as `i`, `rs1`, `rs2`, `simm13`, and `disp30`.

Unlike pattern cases in an explicit switch statement, `sem` declarations, like those above, can declare the semantics of several instructions simultaneously. Semantic code for several similar instructions is parameterized using a `where` clause. By sharing code in this way, ISA semantics are expressed more compactly and errors are more readily detected and fixed. In Figure 4.7, `op` maps to an operator in a list of operators, where the operator corresponds to the instruction name being defined.

Although Facile normally does not allow pointers to functions or for operators to be treated as functions, the `where` clause is an exception. A parameter defined in a `where` clause can be

bound to an operator name (interpreted as a function of one or two arguments as appropriate), to a function name declared earlier in the code, to a single atomic value, or to any more complicated expression enclosed in parenthesis. The usefulness of this exception can be seen in the example, where semantic code is parameterized on the operation being computed. These operations are most easily expressed using Facile's built-in operator names.

4.2. Controlling Memoization

The structure of a simulator written in Facile controls how memoization is used to optimize its execution. Fast-forwarding—my term for memoization as used in simulator optimization—works by caching data that represents the residual, unavoidable parts of simulator execution, indexed by a carefully chosen subset of the micro-architecture state. The structure of a Facile simulator identifies, in a natural way, which subset of data to include in the memoization cache index and how often an index entry should be generated.

4.2.1. Implicit Outer Loop

All Facile simulators are written with an implicit outer loop that calls the simulation code repeatedly to step through the execution of a simulator. Whether written in Facile or not, a top level loop is a common component, at least conceptually, of every instruction-level simulator. In simple fetch-decode-execute style simulators, and most simulators that pre-decode instructions, the outer loop is explicitly written in simulator code. Cross-compilation style simulators (e.g., using direct-execution) and some threaded-code implementations do not use an explicit outer

loop, but they still have an implicit evaluation cycle in which instructions are fetched, decoded, executed, and extra simulation work is performed.

Facile uses its implicit outer loop to control when memoization index entries are generated. At the beginning of each iteration of the implicit loop, a fast-forwarding optimized simulator generates a memoization cache index entry. If the index entry already exists in the memoization cache, then the residual—fast—version of a simulator is executed, otherwise the complete—slow—version of the simulator is called. The slow simulator, if called, caches data that drives later fast simulation, and associates that data with the current iteration's index entry. By controlling how much work is done in each iteration, the programmer controls how often a memoization cache index is generated and how often it is possible to switch to fast simulation.

4.2.2. Top-Level Simulation Function

Simulators are written as code contained in or called from a single function that is called once for each iteration of the implicit outer loop. This top-level simulation function is called `main`[1] in Facile. Just as conventional memoization systems cache function results indexed by the function's name and argument values, FastSim caches residual simulation indexed by the arguments of `main`. The programmer decides which data to pass as arguments to `main` and which data to pass in global variables. The values in `main`'s arguments are packed into a memoization cache index entry, which is used to look for repeated work in subsequent simulation.

1. When compiled into C code by the Facile compiler, this function is renamed `xmain` (for slow simulation) or `ff_main` (for fast simulation). The `main` function in C is implemented within the FastSim runtime library, and is responsible for calling `xmain` or `ff_main` as needed.

```
val init = (system?start_pc, system?start_pc+4);

fun main(pc, npc)
{
    // Copy argument values to global variables
    PC = pc; nPC = npc;

    nPC2 = nPC + 4;         // default next nPC
    PC?exec();              // execute instruction

    // Set init for next call to main
    init = (nPC, nPC2);
}
```

FIGURE 4.8: Simple Simulator #1. This code fragment is taken from a variation of the complete simulator in Appendix B. It shows the `main` function for a very simple SPARC simulator, and the `init` variable where argument values for `main` are stored between calls. This function simulates exactly one target instruction, then sets up argument values for the next iteration and returns. Note that `init` is initialized with the entry point of the target program, but is reassigned by `main` each time `main` is called.

The number and types of arguments passed to `main` is specified by the programmer. Argument data values are taken from a special global variable—called `init`—that holds data for all of `main`'s parameters. Figure 4.8 shows the `main` function and `init` variable declarations used in a simple SPARC simulator. This function executes exactly one instruction from the target executable for each iteration of Facile's implicit outer loop. Memoization index entries are generated before simulating each target instruction, and contain two token stream values: the current program counter (`pc`) and next program counter (`npc`) of the instruction to be executed. At the start of every call to `main`, the values in `main`'s parameters are copied to global variables so they can be referenced by semantic code associated with branch and call instructions, and then the current instruction is simulated. Finally, the next program counter (`nPC`) and next-next program counter (`nPC2`) are stored in `init` for use as arguments to `main` in the next iteration.

```
val init = system?start_pc;

fun main(pc)
{
    PC = pc;         // copy pc to global variable PC
    nPC = PC + 4;    // compute nPC from PC value
    taken = false;   // initialize branch taken flag

    while(!taken || nPC != PC + 4) {
        nPC2 = nPC + 4;          // default next nPC
        PC?exec();               // execute instruction
        PC = nPC; nPC = nPC2;    // update PC and nPC
    }

    // Set init for next call to main
    init = PC;
}
```

FIGURE 4.9: Simple Simulator #2. This code fragment is from a more efficient version of the simple simulator in Figure 4.8. This `main` function simulates several instructions, looping until a branch has been taken and the next PC follows the current PC sequentially. The global variable `taken` is set to true in semantic code (not shown) associated with branch, call, and indirect jump instructions, when the control transfer is taken. This implementation generates fewer memoization index entries, with less data stored per entry.

Figure 4.9 shows how a programmer can vary the content and frequency of memoization index entries. This `main` function simulates several instructions before returning. In particular, it simulates instructions until the next sequence of straight line code (i.e. `nPC == PC + 4`) following a taken control transfer instruction. The boolean variable `taken` is set in the semantic code associated with control transfer instructions (not shown) when a branch is taken. By simulating more instructions per call to `main`, this implementation generates fewer index entries, and saves space in the memoization cache. Space is also saved by only passing the current program counter (`pc`) as an argument to `main`, so less data is stored in each index entry. This savings is

possible because of the simulator design guarantees that `nPC == PC + 4` at the start of each call to `main`.

Although simulators can be designed to generate arbitrarily few memoization index entries, care must be taken. Each memoization index entry represents a potential for finding repeated simulation that can be replayed by fast-forwarding. If there are too few index entries, opportunities to start fast simulation may be missed. A more serious problem is the cost associated with recovering when fast-forwarding fails. This happens when some execution path is encountered by the fast simulator that was not previously encountered by the slow simulator, so no data for it exists in the memoization cache. To recover, simulation is rolled back to the beginning of the current call to `main` and restarted. A long running `main` function can be very expensive to roll back. Performing more work in a each call to `main` also increases the probability that a given iteration will fail when fast-forwarding, because there are more decision points where memoized data may be missing. The effect of design decisions like these are explored more fully in Chapter VI on performance and writing efficient memoizing simulators.

4.2.3. Static, run-time static, and dynamic code & data

In addition to controlling the content and generation frequency of memoization index entries, a Facile programmer must also consider how a memoizing simulator will be split into its two complementary versions. The fast version of a simulator contains only the code that cannot be skipped by fast-forwarding. The less code contained in this residual version, the faster it will run. By careful simulator design and a small amount of additional annotation, the programmer can

control how much work is performed in the fast simulator, and how much is skipped over by fast-forwarding.

The key to understanding how the Facile compiler generates a fast-forwarding simulator is to understand the division of Facile code and data into binding time classes. The binding time of a data value is the earliest time at which that value can be computed. Similarly, the binding time of a piece of code is the earliest time that code can be evaluated to produce its result value. For example, a literal constant (e.g., `5` or `0x1f`) is known at compile time, as is any expression that depends only on literal values (e.g., `(5*6)+7`).

For fast-forwarding, there are three relevant binding time classes: *Static* code and data is known at compile time (e.g., literal expressions like `5+1`). *Run-time static* values are computed by the slow version of a simulator, but are considered known constant values when fast-forwarding (e.g., the arguments to `main`). Finally, *dynamic* values cannot be known until the moment they are computed by either the slow or fast versions of a simulator. Obviously, dynamic code cannot be skipped over by fast-forwarding, since its result cannot be pre-computed. But static and run-time static code can be skipped, so it is left out of the fast version of a simulator, leaving only dynamic code to be executed.

The Facile programmer controls which code and data belong to each binding time class. An expression is made static by writing it so it depends exclusively on literal values and other static expressions. Not all simulation code cannot be made static—i.e., known at compile time—but the

programmer has absolute control in deciding whether the remaining, non-static code should be run-time static or dynamic. This is accomplished through the choice of arguments to `main`, placement of conditional expressions, and occasional use of binding time annotations. Figure 4.10 contains code fragments from a simple simulator that show how binding time classes are assigned. In general, this simulator is designed so that instruction decode is run-time static, but most instruction semantic code is dynamic. The next few paragraphs explain briefly how this division is computed.

At the start of every iteration of Facile's implicit loop, `main`'s arguments and the `init` variable are assumed to be run-time static, while all other global variables are dynamic. The binding time classes of subsequent code are largely determined by this initial division. Any subsequent expression that is not static but depends only on static and run-time static data is itself run-time static. Variables on the left hand side of an assignment take on the binding time class of the expressions they are assigned from. For example, the first two lines of the `main` function in Figure 4.10 are run-time static, because they compute values based only on literal constants and the run-time static parameter `pc`. Note that the global variables `PC` and `nPC` contain run-time static data after these assignments, even though they started out dynamic when `main` was called. The rest of the code in `main` is mostly run-time static because it also depends only on literal values and the parameter `pc`, although some dependencies are more complex. Also, the `?exec` attribute call in the expression `PC?exec()` is labeled dynamic in this example because it calls a function containing both run-time static and dynamic code, although the code to decode target instructions is entirely run-time static because the variable `PC` is run-time static.

```
fun Rx(i0) {                          // get 64-bit register value
    val ii = i0?ext(32);
    if(ii == 0) return 0?ext(64);
    else if(ii < 8) return global_registers[ii-1];
    else {
        val win = (CWP?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); return register_windows[win][ii];
    }
}

sem jmpl {
    nPC2 = (Rx(rs1) + SRC2)?cvt(stream)?static;
    Rx(rd,PC?addr?ext(64)); taken = true;
};

#define i_ne    (!CCR?bit(2))
#define i_e     (CCR?bit(2))

sem [                   bne     be      bg      ble     bge     bl
        bgu     bleu    bcc     bcs     bpos    bneg    bvc     bvs ] {
    if(cond) { nPC2 = PC + disp22?sext(32)<<2; taken = true; }
    else if(a) annul();
} where cond in [       i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs];

sem [ add sub and or xor ]
{ Rx(rd, op(Rx(rs1),SRC2)); }
where op in [ + - & | ^ ];

fun main(pc)
{
    PC = pc;        // copy pc to global variable PC
    nPC = PC + 4;   // compute nPC from PC value
    taken = false;  // initialize branch taken flag

    while(!taken || nPC != PC + 4) {
        nPC2 = nPC + 4;         // default next nPC
        PC?exec();              // execute instruction
        PC = nPC; nPC = nPC2;   // update PC and nPC
    }

    // Set init for next call to main
    init = PC;
}
```

FIGURE 4.10: Simulator code with binding time labels. This code is taken from a variation of the simple SPARC simulator given in Appendix B. Dynamic code is underlined. All non-underlined code is either static or run-time static. Some functions contain both dynamic and non-dynamic code, and calls to these functions are underlined.

A second way data can become run-time static is through conditional statements—e.g., `if` and `switch`. A dynamic conditional statement—i.e., one that tests a dynamic condition—converts the result of its conditional expression to a run-time static value before it tests the condition. Hence, the direction of control flow following a dynamic conditional statement is run-time static. One result is that variables assigned run-time static values along one branch of a dynamic conditional statement remain run-time static at the end of the conditional statement, when the alternate control flow paths merge back together. To make a dynamic condition run-time static, the fast version of a memoizing simulator verifies that the dynamic condition evaluates to the same result as a previously cached result value, otherwise fast-forwarding will fail and control be passed back to the slow simulator. By verifying the dynamic result, Facile makes all control flow choices run-time static.

Consider the semantic code for branch instructions in Figure 4.10. The first if statement is dynamic because it tests the dynamic value of the condition code register variable (`CCR`). In the "true" branch of this statement, the variable `nPC2` is assigned a run-time static value. If Facile did not verify the value of the dynamic condition when fast-forwarding, then it could not know whether the assignment to `nPC2` occurred or not, and `nPC2` would be dynamic at the end of the if statement.

Finally, any dynamic variable or expression result can be made run-time static by explicit use of the `?static` attribute. In Figure 4.10, `?static` is used to make the target address of an indirect call instruction (`jmpl`) run-time static. Without this annotation, indirect call target addresses

would be dynamic because they depend on the dynamic values of simulated integer registers. As with dynamic conditional statements, a dynamic value is made run-time static by verifying it against values previously stored in the memoization cache. If there is no matching value in the cache, then fast simulation fails and control is passed back to the slow simulator. If a matching value is found, then simulator behavior has already been memoized for this dynamic value and the fast simulator can continue to replay memoized simulation.

An efficient memoizing simulator must balance the desire to skip over as much code as possible against the amount of memoization index and result data stored into the cache. Increasing the portion of run-time static code in a simulator often leads to an increase in the amount of memoization data generated. Putting more data into a memoization index (i.e., by adding values to `main`'s list of arguments) can increase memoization cache consumption for two reasons: Each index entry is larger, and more index entries may get generated because they are less likely to match existing entries. Similarly, dynamic conditional statements and `?static` annotations can increase cache consumption and increase the probability of a memoization miss, causing fast simulation to fail and a return to slow simulation.

Another source of memoized data comes from using run-time static values in dynamic computation. Every run-time static value that is used as an operand in a dynamic expression must be cached, so it will be available in subsequent fast simulation. Since static values are available at compile time, they are generated directly into the code for the fast simulator version, and only run-time static data takes up space in the cache. Hence, to reduce consumption of memoization

cache space, it is desirable to minimize the amount of run-time static data used in dynamic code by designing dynamic expressions that use only static and dynamic values wherever possible.

An ideal memoizing simulator design reduces the quantity of memoization data while simultaneously skipping as much run-time static code as possible and using this small amount of residual code as often as possible. This is accomplished by careful selection of data to include in the memoization cache index and careful crafting of the structure of simulator code. Chapter VI discusses various alternative simulator designs and their affect on performance.

4.3. Other Features

The Facile programming language incorporates several features that either help a programmer write memoizing instruction level simulators or simplify analyses in the Facile compiler. These features include language constraints that simplify analysis, built-in datatypes that can be memoized and are useful for implementing micro-architecture simulators, and a simple interface for calling non-Facile code to handle tasks that are difficult to implement in Facile and do not need to be memoized.

4.3.1. Limitations to Simplify Compiler Analysis

Facile has no pointer types. This restricts the coding style used in Facile programs, and makes it easier to perform accurate alias analysis. Alias analysis determines the storage locations referred to by each variable at each point in a program, and if multiple variables refer to the same storage location. The results of this analysis are used to determine the data dependencies in a Fac-

ile program, so each expression can be labeled with its appropriate binding time class. Too conservative an alias analysis may cause unnecessary data dependencies, resulting in potentially run-time static code being labeled dynamic by the compiler. Pointer variables are often difficult to analyze, because the storage location they point to may not be known until run-time and can change as the program executes. If the storage locations of all variables are known at compile time, then aliases can be accurately determined, improving the accuracy of the data dependence graph.

Unfortunately, there are other sources of aliasing in Facile, namely array and queue lookup expressions, and reference variables and parameters. All of these are included in the Facile language because they are needed for writing efficient micro-architecture simulators. Because the index value used to access an array or queue may not be known at compile time, the element accessed may not be known. This is an example of aliasing, because the result of a lookup may refer to any element of the array or queue. Unlike aliasing with pointers, aliasing in arrays is more controlled: the compiler knows that the storage location referenced must be one of the array's elements. Facile's compiler uses this information to limit the number of extra dependencies in its conservative data dependence graph.

Although Facile does not have pointers, it does have reference variables and allows function parameters to be passed by reference. Reference variables may refer to the same storage location as another variable, and changes to one are visible in the others. The difference between reference variables and pointers is that a reference variable cannot be changed to point to a different storage

location after it is initially bound. Hence the storage location referred to by a reference variable is known at compile time, unless other factors make the analysis inexact. For example, a reference variable initialized to the result of an array lookup may not be statically known, but this is a result of the unknown array lookup not the reference.

A second restriction of the language is the absence of recursion. A Facile function cannot call itself either directly or indirectly. This restriction simplifies some inter-procedural compiler analyses, but it is primarily needed for efficient recovery from failed fast-forwarded execution. When the fast version of a simulator cannot find data it needs in the memoization cache—i.e., a memoization miss—it fails and returns control to the slow simulator version. The no-recursion restriction allows local function variables to be stored in the global scope, so values computed by a failed fast simulation can be transferred to the slow simulator in global variables and not on a stack.

The memoization miss recovery mechanism is discussed more fully in Chapter V, but it can be described briefly as follows. All variables in a simulator have two storage locations, one when the value is dynamic and the other when the value is static or run-time static. Whether a Facile variable was declared locally or globally, the dynamic version of its storage location is always allocated globally in C, so it is accessible from any function. This is why there can be no recursion; FastSim never stores more than one copy of a variable's dynamic data during recovery. After a memoization miss the slow simulator is restarted with arguments taken from the last index entry encountered by the fast simulator, i.e., the last call to `main`. While the slow simulator is recover-

ing from a memoization miss it only executes static and run-time static code, which only reads and writes the run-time static versions of variables. Once the slow simulator catches up to the point where fast simulation failed, the run-time static and dynamic data is exactly the same as if the fast simulator had never been run, and slow simulation can continue normally.

4.3.2. Special Datatypes

In addition to data types common to many programming languages, Facile also has several data types that are explicitly included to support micro-architecture simulation, and allow their data to be memoized. The token stream type (just called `stream` in Facile) works with `token`, `pat`, and `sem` declarations and Facile's switch statement to easily decode instructions found in a target program. Variables of type `cc` represent condition codes, a common architecture feature of many ISAs. The `?cc` annotation allows condition code variables (of type `cc`) to be efficiently set with condition code values resulting from arithmetic operations. To support the implementation of out-of-order processor simulators—one of the driving examples for the design of FastSim v.2—Facile also provides a built-in double ended queue datatype. Including these types into the language, ensures Facile is able to analyze them and memoize their data.

The token stream type represents a stream of instruction tokens starting at a given program address. The primary operation on a token stream is to decode and evaluate the semantics of the first instruction in the stream, using the `?exec` attribute. Instructions are decoded and evaluated based on information provided in `token`, `pat`, and `sem` declarations, as described in section 4.1.

Other operations include adding (`+`) or subtracting (`-`) an integer offset to the token stream's base address, and converting the stream address into an integer using the `?addr` attribute.

Condition codes are sometimes generated to denote various conditions on the result of some arithmetic operator, in addition to the operator's result value. Typical integer condition codes include 1-bit flags to indicate if a result is negative (N), equal to zero (Z), caused an overflow (V), or incurred a carry (C). Typical floating-point condition codes indicate if the first operand was equal to (=), less than (<), greater than (>), or unordered (?) with respect to the second operand in a floating point compare. These condition code results are efficiently generated using the `?cc` attribute and a variable of type `cc` to hold the condition code result. For example, the expression `(op1+op2)?cc(CCR)` returns the sum of integer variables `op1` and `op2`, while simultaneously setting the variable `CCR` to the integer condition codes resulting from this summation. Condition code values are then accessed by extracting bits from the condition code variable (e.g., `CCR?bit(2)` extracts the Z flag from `CCR`).

For efficient implementation on a SPARC host, FastSim v.2 implements the `?cc` attribute for all of Facile's integer binary operators using the corresponding native SPARC instructions that produce these condition codes. Hence the condition codes generated are a near perfect match for the condition codes needed in a simulator of the SPARC instruction set. Facile's condition codes may not work as smoothly for modeling other ISAs, although the kinds of condition codes used in other architectures are very similar to those implemented here. Condition codes used in other ISAs that do not correspond to SPARC condition codes can be programmed in Facile.

Finally, double ended queues act much like arrays, but elements can be added or removed from both the beginning and end of a queue, and a queue dynamically grows and shrinks as needed. The motivation for the queue datatype is primarily for use in simulators that model hardware buffers, like in the instruction window of an out-of-order micro-architecture. Queues are a common construct in hardware, and Facile's queue type should be useful in other parts of a micro-architecture simulator. Queue elements can be accessed just like array elements, with an index in square brackets (e.g., `Q[i]`). Other queue operations include the attributes `?push_front`, `?push_back`, `?pop_front`, `?pop_back`, `?length`, and `?clear`.

The advantage of defining double ended queues and Facile's other datatypes as language built-ins, is that they can be analyzed by the Facile compiler. Facile also allows external datatypes to be defined. These external datatype definitions give a Facile type name to C's `void*` type, and objects of these types are manipulated exclusively by external (non-Facile) function calls. Facile variables can reference data with an external type, but the data cannot be stored in the memoization cache, and will always have a dynamic binding time class. Facile's built-in datatypes and types made using Facile's type constructors (e.g., `struct`, `array`, and `queue`) can be memoized, and values having these types can be skipped over by fast-forwarding.

4.3.3. A Simple interface to C

Facile contains many common programming language constructs (e.g., functions, while loops, and if statements) and data types (e.g., integers, records, and arrays), but it may not be appropriate for all the tasks required of a complex simulator implementation. It may also not be appropriate to

use memoization in all the components of a simulator. For these reasons, Facile provides a simple interface for linking a Facile simulator to external code, possibly written in a non-Facile language such as C. Using Facile's `extern` declarations, Facile code can call functions and access variables implemented in C, and C code can access functions and variables written in Facile.

External function and variable declarations provide type information that Facile needs to communicate with external code. Once variable and function names have been declared external and their types specified, they can be used in Facile code to call the named functions or to access external variables. Alternatively, externally declared names can be given code in Facile, then called from external code. In this way, not only can Facile code call out to code written in another language, but that code can call back into Facile.

Functions implemented in external code, and accessed via Facile's external declarations, are not memoized. But a simulator written in Facile can still be memoized, even though it may call external functions and access external variables. Conservative assumptions are used in place of actual analysis of external code. The Facile compiler detects all the ways external code can influence the Facile code in a simulator through its externally defined interfaces, and binding-time analysis takes this into account. All external code is considered dynamic, and any change to Facile data that may result from external code is also dynamic. Hence, every call out to external data is dynamic and cannot be skipped by fast-forwarding, any value returned by a call to an external function is dynamic, and any Facile variable that may be changed by external code becomes dynamic after every external call. Facile variables may be changed by external code if they are

```
extern set_CCR(cc) : void;
fun set_CCR(CCR1 : cc) : void { CCR = CCR1; }

extern get_R4(cwp_t,ulong) : ulong;
extern set_R4(cwp_t,ulong,ulong) : void;

fun get_R4(cwp : cwp_t, i0 : ulong) {
    val ii = i0?ext(32);
    if(ii == 0) return 0?ext(32);
    else if(ii < 8) return global_registers[ii-1]?bits(32);
    else {
        val win = (cwp?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); return register_windows[win][ii]?bits(32);
    }
}

fun set_R4(cwp : cwp_t, i0 : ulong, vv : ulong) {
    // Similar to get_R4 except it sets register i0
    // in register window cwp to values vv.
}

extern trap_sparc(ulong,cwp_t,cwp_t): void;
```

FIGURE 4.11: External function declarations. These declarations are taken from the simple simulator in Appendix B. They are used to interface Facile code for this simulator with C code that emulates SPARC Solaris operating system calls.

declared external (i.e., made visible to external code) or are set by some Facile function that is declared external.

Figure 4.11 shows an external declaration of the `trap_sparc` function that is implemented in a separate C file, and several Facile functions declared external so they can be called from C. These Facile functions are made external to allow C code access to the Facile variables that represent integer registers and integer condition codes in the simulated SPARC ISA. The `trap_sparc` function emulates system calls of the Solaris operating system, as is invoked by a simulated trap instructions. Semantic code in Facile that simulates trap instructions calls

`trap_sparc`, passing in a trap number and current values of the simulated `CWP` and `CANRESTORE` registers, used to maintain the register window state. The `trap_sparc` function, written in C, calls `get_R4` to get integer register values used by the system call, then emulates the system call[1]. Finally, `set_R4` and `set_CCR` are called from C to update the simulated registers with results from the system call.

1. Since the host machine is also running Solaris, system call emulation is accomplished by copying the simulated register values to actual host registers and executing a `ta` (trap always) instruction on the host.

CHAPTER V: Facile Compilation and Automatic Simulator Memoization

FastSim Version 2 simplifies the implementation of fast-forwarding by automatically optimizing simulators written in Facile to use memoization. The Facile compiler generates two cooperating versions of a simulator, guaranteeing that they communicate memoization data correctly and can correctly transition both from slow to fast and from fast to slow simulation. The two simulator versions are linked to a run-time library that helps manage the memoization cache and implements the rest of FastSim's simulation framework. A programmer need only write one version of a simulator, with hints to the compiler about how memoization should be used, and the compiler will automatically implement the fast-forwarding optimization.

Section 5.1 introduces the Facile compiler and run-time library, discussing features not related to memoization. Section 5.2 continues by detailing the implementation of the fast-forwarding (memoizing) optimization.

5.1. Facile Compilation

The Facile compiler translates Facile code into equivalent C code, then calls gcc to compile the C code and link it with FastSim's run-time library. Figure 5.1 shows the general stages for compiling and linking a FastSim simulator. Facile source first passes through the C preprocessor

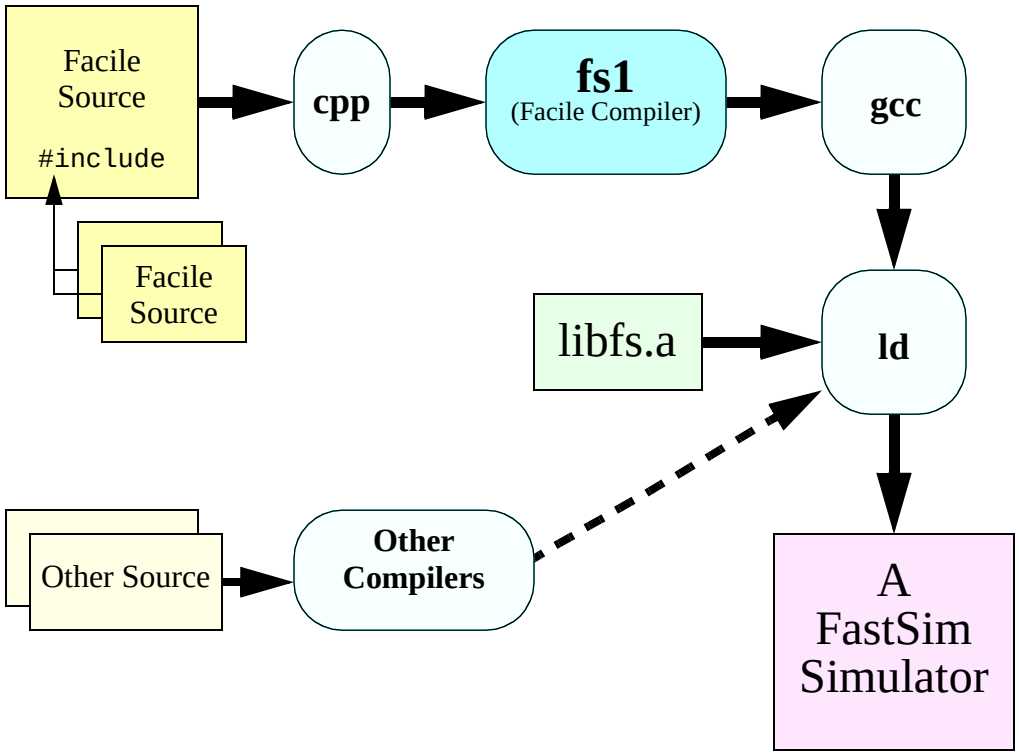

FIGURE 5.1: Stages of compilation. The Facile compiler (**fs1**) translates preprocessed Facile source code into C source code. The generated C code is compiled and linked to the FastSim library (**libfs.a**), and other simulator object code, to produce a complete simulator executable.

(**cpp**) then through the Facile compiler (**fs1**) to produce C source code. Note that Facile's compiler only optimizes a single source file to use memoization, because it must analyze all memoized code at the same time. If other files contain code to be memoized, they must be included—using the `#include` preprocessor directive—into a single Facile source file. Finally, C code produced by **fs1** is compiled by **gcc** and linked to the FastSim run-time library (libfs.a). External code written in languages like C can also be linked at this time.

5.1.1. The Facile Compiler

Facile code is transformed to C code by the Facile compiler. Most Facile language features have analogous counterparts in C—e.g., if statements, assignments, function calls, and most expression operators—and are translated in an obvious way. Other features are not so simple. For example, pattern cases in a Facile switch statement are translated into several embedded switch and if statements in C. Facile's type system also differs from C: In addition to having different base types and type constructors, Facile types do not have to be specified when variables and functions are declared. The compiler uses type inference to determine appropriate types for all variables, functions, and expressions from the context of their declaration and usage.

Ignoring the memoization optimization for now, the significant stages of compilation are:

- PARSING: Lexical analysis and parsing are implemented with flex and bison respectively.

- PATTERN NORMALIZATION: Pattern expressions—describing the binary encoding of instructions—are normalized into disjunctive normal form, i.e., OR-lists of AND-lists of conditions on token fields.

- TYPE CHECKING: Type inference assigns types to every variable, function, and expression. This analysis is complicated by potentially polymorphic and overloaded function definitions.

- CONVERTING SWITCH STATEMENTS: Pattern cases in Facile switch statements are converted into nested switch and if statements that can easily be translated into C code.

- FUNCTION INLINING: Calls to local functions and functions explicitly annotated as inlined are replaced with the text of the called function, removing the function call from the code.

- GENERATE C CODE: One C source file is generated that contains all the global variable and function definitions translated from a Facile source file. A C header file is also generated, containing variable, function, and type names declared external in Facile source code.

5.1.1.1. PATTERN NORMALIZATION

Pattern expressions appear in `pat` declarations and in pattern cases of switch statements. A pattern expression is a tree of pattern operators (e.g., `&&`, `||`, and `$`) with conditions on token fields at the leaves. The compiler normalizes these trees into OR-lists of AND-lists of conditions on token fields, i.e., disjunctive normal form. Individual conditions in the expression tree become OR-lists containing one AND-list containing the single condition. The `||` operator is normalized by appending the OR-list of its right-hand operand to the end of the OR-list from its left-hand operand. The `&&` and `$` operators are normalized by taking a cross product of their operands—i.e., every AND-list on the right-hand side is appended to every AND-list on the left-hand side and all the resulting AND-lists are OR'ed together. The `$` operator also adds an offset to conditions in its right-hand side operand, so these conditions test fields belonging to tokens later in the token stream, after the all the left-hand side operand's tokens.

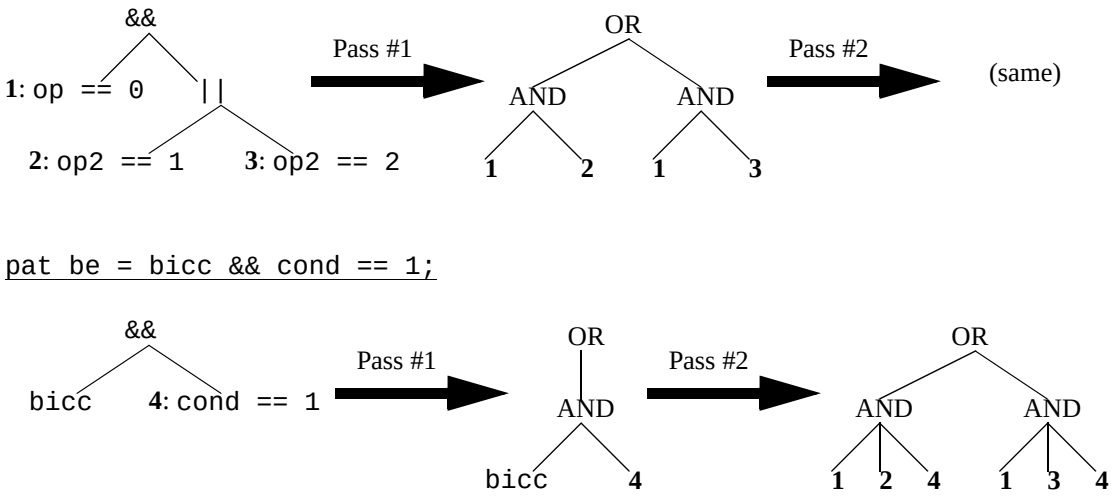

FIGURE 5.2: Normalizing pattern expressions. Pass #1 normalizes a pattern expression tree without expanding any previously declared pattern names. Pass #2 expands pattern names and re-normalizes the pattern. Individual conditions are represented by numbers (1 through 4) in this figure to save space.

Pattern expressions are normalized in two passes. The first pass normalizes all operators and conditions in a pattern expression, but does not expand any pattern names defined by earlier `pat` declarations. The second pass expands previously defined pattern names and re-normalizes the pattern. Two passes are needed because `pat` declarations that define more than one pattern name need to associate each new name with one element of the pattern OR-list before any named patterns are expanded. Figure 5.2 illustrates the normalization process for two simple pattern expressions.

```
// 5 is of type ulong, so x is of type ulong
val x = 5;

// Parameter y has type ulong, which can be inferred either of two ways:
// It is added to the ulong value 1 or because the result is assigned
// into the ulong variable x. Hence function f has type ulong → void.
fun f(y) { x = y + 1; }

// Function ident is a polymorphic function of type α → α.
fun ident(y) { return y; }

// Function ident is instantiated here with type ulong → ulong,
// so z1 has type ulong.
val z1 = ident(5);

// Function ident is instantiated here with type float → float,
// so z2 has type float.
val z2 = ident(1.0);
```

FIGURE 5.3: Type inference of variables and functions.

5.1.1.2. TYPE CHECKING

Facile is a strongly typed language, but uses type inference to reduce the number of types that must be explicitly specified by a programmer. In the absence of explicit type information, Facile's type checker infers the types of variables from the contexts in which they are used. Function types are partially inferred from the code that defines a function, but may also be influenced by the context in which a functions is called. Figure 5.3 shows how variable and function types are inferred from their context, and how a function can have different types when used in different contexts.

Function types are more complex than variable types, because functions names can be overloaded (i.e., declared several times with different types) and polymorphic (i.e., part of the function type is represented by a type variable rather than an explicit type). When an overloaded function is

called, the types of its arguments and expected return value are tested against all overloaded versions of the function to determine which version should be used. If a called function is polymorphic, a copy of the function is made in which its free type variables are instantiated with explicit types corresponding to the types used in the call. After type checking an entire program, all polymorphic type variables will have been replaced by explicit types, and overloaded function versions renamed to distinguish between them.

The Facile type checker uses an extension of the polymorphic type inference algorithm—algorithm *W*—described in [47]. Algorithm *W* walks through a source tree with a depth first traversal, assigning types to each expression. Type variables represent unknowns in the current type information for the already traversed expressions. A type environment (*E*) maps type variables to more explicit types, as the explicit type information becomes known. For example, to type check the expression `y+1`, `y` is initially given type α (an unbound type variable), and `1` is given explicit type `ulong` (the 32-bit unsigned integer type). Then the `+` operator, with initial type $(\beta, \beta) \rightarrow \beta$, is unified with type $(\alpha, \texttt{ulong}) \rightarrow \gamma$, which corresponds to applying `+` to the operands `y` and `1`. As a result of unification, type environment *E* is changed by adding the bindings $\alpha \mapsto \texttt{ulong}$, $\beta \mapsto \texttt{ulong}$, and $\gamma \mapsto \texttt{ulong}$. So `y` has type `ulong` and the expression `y+1` produces a result of type `ulong`.

Facile's type checking algorithm is similar, except it maintains a set of type environments (*E**). Each type environment E_i in *E** corresponds to using different versions of overloaded functions and operators in the traversed code. For example, Facile actually defines `+` as an overloaded

operator, with one version for integer operands and another version for floating-point operands. To type check the expression `y+1`, Facile's type checker starts by giving `+` type α, and creates a new environment set $E^{*\prime}$ that contains multiple copies of each environment E_i from the original environment set E^*, but with different bindings for α. One copy of each $E_i \in E^*$ appears in $E^{*\prime}$ with α bound to the type of the integer version of `+`, and another copy has α bound to the type of the floating-point version of `+`. Subsequent unification may discover that some type environments in $E^{*\prime}$ are illegal—e.g., the floating-point version of `+` cannot be used on the `ulong` constant `1`—and remove them from the type environment set. Type inference succeeds, so long as there is at least one legal type environment in the set E^*.

Function bindings can be overloaded by either writing multiple explicit function definitions with the same function name, or by a single function definition that type checks with more than one legal type environment, or both. If a single definition has multiple legal type environments, then the function's type must be different under each type environment, otherwise it would not be possible to distinguish between these overloaded versions. Overloaded functions are renamed to eliminate the overloading.

Polymorphic functions contain at least one type variable in their inferred type, which is free (i.e., not bound) in the type environment. Code for polymorphic functions is not generated by the compiler. Instead, the compiler waits until it discovers a call site for the function and can infer explicit types for the free type variables. Then a monomorphic instance of the polymorphic function is generated using explicit types from the call site. Different calls, with different explicit

types, will generate different instances of the function. Different calls, with the same explicit types, share the same monomorphic instance. As with overloaded functions, monomorphic instances of a polymorphic function are renamed to identify which instance is used by each call.

5.1.1.3. Converting Switch Statements

Facile switch statements are similar to C switch statements, testing the switch condition against several alternate constant values and selecting code to execute. But unlike C switch statements, Facile can test an instruction stream for a patterns containing multiple conditions on fields of a token stream. To select between these pattern cases, several conditions may need to be tested. The compiler rewrites Facile's pattern cases into multiple nested switch and if statements that only test one condition at a time. This transformed code can easily be written as C code by later stages of the Facile compiler.

Pattern expressions for each pattern case are normalized into OR-lists of AND-lists of conditions on token fields, as described in section 5.1.1.1. Token fields identify contiguous sequences of bits at fixed offsets from the start of a token stream. Conditions compare the value in a field against a given constant value, using one of the operators: <, <=, ==, !=, >=, or >. A pattern case matches the data in a token stream if every condition in at least one AND-list is satisfied. If more than one pattern case matches the bits in a given token stream, then the first matching pattern case listed in the switch statement is selected.

Pattern cases are transformed into simpler, C-like if and switch statements in two stages. First, each element of a pattern's OR-list is given its own pattern case. Code associated with the original pattern case is associated with only the first of the new pattern cases. The other new pattern cases simply goto the code associated with the first new case. After this first stage of transformation, each pattern case contains a single AND-list. The second stage transforms these simpler pattern cases into C-like if and switch statements.

Figure 5.4 details the algorithm for transforming pattern cases with single AND-lists into simpler, C-like statements. The algorithm recursively selects a condition, then generates nested switch and if statements to test that condition and divide up the list of pattern cases. Once tested, conditions are removed from their associated AND-lists, until all the conditions for a pattern case have been tested and removed. When there are no more conditions for a pattern case, the code associated with that case is inserted. The compiler detects unreachable pattern cases when the first case in a list of pattern cases has no more conditions, making all but the first case in the list unreachable.

5.1.1.4. FUNCTION INLINING

Function inlining is a program transformation that removes function calls by replacing them with the text of the called function. This optimization eliminates some call overhead—e.g., it removes call and return instructions, and a function's prologue and stack frame—and allows the called function's code to be optimized together with code in the calling function. Hence register

```
function convert_switch
argument Clist (a list of a pattern cases, with 1 AND-list each)
{
    if the first case in Clist has no conditions {
        generate code associated with the first case in Clist
        return from convert_switch
    }

    let cond = the first condition in the first case in Clist
    let field = the token field tested by cond

    if cond tests for equality (i.e., field == value) {
        split Clist into two parts (C1 and C2) so that
            (C1 followed by C2) == Clist
            all the elements of C1 contain a condition that
                tests for equality on the field named in field
            the first element of C2 does not contain
                a condition on field that tests for equality

        generate a switch statement to test the value of field
        group elements of C1 by the value they test field against
        for each group of elements in C1 {
            generate a case to test for the value used in this group
            remove the condition that tests against field from each case
            call convert_switch on the elements of this group
        }
        end the generated switch statement

        call convert_switch on C2 (to handle the fall-through case
            where none of the cases in C1 match)

    } else cond does not test for equality {
        split Clist into two parts (C1 and C2) so that
            (C1 followed by C2) == Clist,
            all the elements of C1 contain condition cond, and
            the first element of C2 does not contain cond

        generate an if statement to test condition cond
        remove cond from all elements of C1
        call convert_switch on C1
        end the generated if statement

        call convert_switch on C2 (to handle the fall-through case
            where none of the cases in C1 match)
    }
}
```

FIGURE 5.4: Algorithm to simplify pattern cases. This algorithm takes a list of pattern cases, where each case contains a single AND-list of conditions, and translates them into C-like if and switch statements.

allocation, instruction scheduling, and other intra-procedural optimizations can produce more efficient code than could be generated when calling a separate function. But function inlining may cause code blowup—a dramatic increase in code size—which decreases program performance by making less effective use of hardware caches.

The Facile compiler uses function inlining for two reasons: 1) to help convert Facile language features into C, and 2) to aid in generating the fast-forwarding optimization. Facile allows local functions to be defined within the definition of another function. Local functions are not allowed in C, where all function definitions must be made in the global scope. It is difficult to translate local functions into global functions, since they can access local variables belonging to the enclosing function. The Facile compiler handles local functions by inlining all calls to them. After function inlining, there are no calls to local functions, only calls to functions defined in the global scope.

In the initial design of the Facile compiler, I assumed that all functions would have to be inlined into a single monolithic function to implement the fast-forwarding optimization. It is possible to inline all function calls into a single function, because Facile functions are not recursive. The resulting monolithic function was inevitably very large and very slow, because indiscriminate inlining causes an exponential blowup in code size. Further thought revealed that it is sufficient for functions to be non-recursive, and they do not need to be inlined to implement fast-forwarding. Even though it is not necessary, inlining of selected functions can improve the performance of a memoizing simulator. The fast-forwarding optimization, and the reason for prohibiting recursion,

is explained in detail in section 5.2. The effect of inlining on performance is demonstrated in section 6.3.5 of Chapter VI.

5.1.1.5. GENERATE C CODE

After applying the transformations above, and when not applying the fast-forwarding optimization, it is straight forward to generate C code for corresponding Facile source code. Facile if and while statements and rewritten switch statements translate into C if, while, and switch statements respectively. Function calls that are not inlined translate into function calls in C. Most arithmetic operators in Facile translate to corresponding C operators. Finally, some operators and annotations in Facile translate into function calls in C that are implemented by FastSim's run-time library or are generated by the compiler. For example, functions that operate on queue typed objects are generated by the compiler, because different versions of the code are needed for queues with different element types.

All translated Facile code is put into a single C source file, except for external function, variable, and type declarations that are put in a separate C header file. The C source file contains type declarations for all structured types (i.e., structure, tuple, array, and queue types), global variable declarations for both static and external variables, static function declarations for all non-external functions, and global function definitions containing translated Facile code. The C header file defines the external interface to translated Facile code, and can be included by other C files to access external Facile variables and functions. An 'x' is added to the front of every external vari-

able and function name to avoid name conflicts with other generated code. For example, Facile's `main` function and `init` variable are called `xmain` and `xinit` respectively in C.

Additional functions are also put into the generated C source file to manipulate queue type objects, and to initialize global variables. Queue manipulation functions—e.g., constructing a new queue, pushing elements onto the front or back, popping from the front or back, and clearing a queue—are generated for each different element type, but only if that function is needed somewhere in the translated Facile code. To initialize global variables, expressions on the right hand sides of each global variable definition are collected into a single function, called `init` in C, that is called by the FastSim run-time library at the start of simulation. This initialization function is necessary, because initialization expressions for global variables in C must be computable at compile time, but Facile allows more complex initialization expressions and evaluates them at run-time.

5.1.2. The FastSim Run-time Library

The FastSim run-time library (libfs.a) loads a target executable, calls the top level Facile simulator function (called `xmain` in C), and implements Facile operations that are not generated by the Facile compiler. Libfs.a also implements the C `main` function that is the entry point for the entire simulator. In a memoizing simulator, this library also switches between slow and fast versions of a simulator and manages the memoization cache, but this will be discussed in section 5.2.

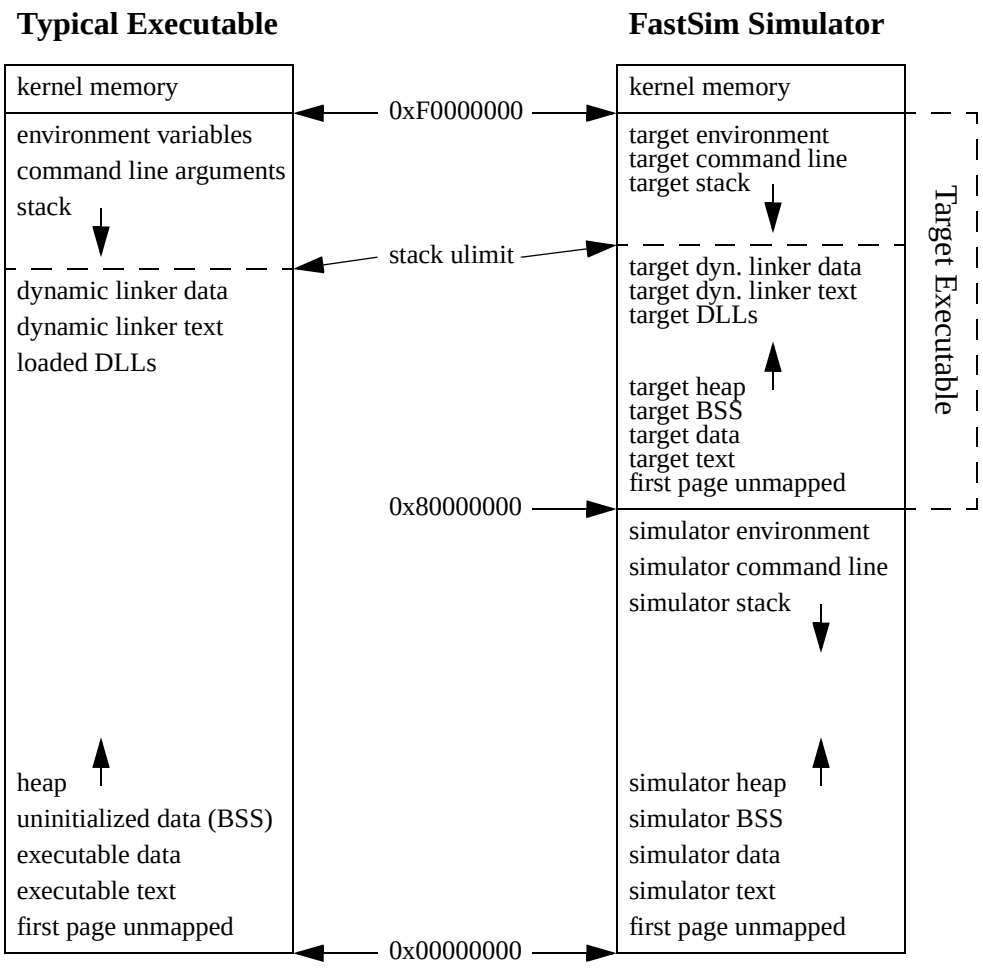

FIGURE 5.5: Address space organization. The figure on the left depicts the memory organization of a typical Solaris executable; The dynamic linker and loaded DLLs are optional, and do not occur in statically linked executables. The figure in the right depicts memory organization in a FastSim simulator: The simulator's text, data, and stack are located below 0x80000000, and the simulated target executable is loaded above 0x80000000.

5.1.2.1. LAYOUT OF SIMULATOR MEMORY

At simulator start-up, a simulator's address space is reorganized to make room for the target executable. Figure 5.5 shows the usual organization of a Solaris executable and how the address space is reorganized in a FastSim simulator. First, a simulator's environment variables and stack

frames are moved below address 0x80000000, into the lower half of the virtual address space. The upper half of the address space (not counting kernel addresses, 0xF0000000 and above) is used to simulate a target executable's address space. Target addresses are translated to simulator addresses by or'ing the target address with the value 0x80000000.

After a simulator's environment variables and stack frames have been relocated, a target executable is loaded by mmap'ing its text and data segments into simulator memory at address 0x80000000 and above. Then the environment variables, command line arguments, and an initial stack frame are constructed below address 0xF0000000. Target environment variables are given the same values as a simulator's environment variables, and target command line arguments are taken form the end of a simulator's command line arguments.

If a target executable is statically linked, then its start address is simply read from the executable header. If a target executable is dynamically linked, then a dynamic linker is loaded that will load all the other DLLs. The path name of a dynamic linker is read from a dynamically linked target executable, and FastSim mmap's the named dynamic linker into the target address space immediately below the lowest possible stack address (determined by subtracting the stack ulimit from 0xF0000000). The start address of a dynamically linked executable is the start address of the mmap'ed dynamic linker. Note that FastSim simulators are statically linked, so the only DLLs are those loaded by a target executable.

In subsequent simulator execution, memory access for the target executable is performed by or'ing every target address with 0x80000000. This translates all target addresses into simulator addresses in the upper half of the simulator's virtual address space. Memory references for a target executable never conflict with memory used for other simulator operations. This address translation also removes half the target executable's available address space (from 0x40000000 to 0xC0000000), but few executables use these addresses.

This memory layout has the following advantages: Address translation is fast (accomplished with a single bit-wise or). A simulated executable cannot access simulator data by mistake, since all target addresses are translated above 0x80000000 and all internal simulator data is located below address 0x80000000. And simulator code is compiled and executed normally without any code relocation. It is easier to relocate code from the target executable, because it is just data to the simulator, than it is to relocate simulator code that actually executes on the host. The disadvantage is that half of the target address space cannot be simulated, i.e., addresses 0x40000000 to 0xC0000000.

5.1.2.2. Run-Time Support for Facile Features

The FastSim run-time library implements Facile's outer simulation loop, operators that compute condition code values, functions for accessing data from a token stream, and other functions with no direct C equivalent like sign extension. Before any calls to `xmain` (the Facile `main` function), the `init` function, generated by Facile's compiler, is called to initialize global Facile vari-

ables. Then the run-time library enters a loop that repeatedly calls `xmain`. Other FastSim library code is called from the compiled Facile code.

The `?cc` attribute in Facile can be applied to many of Facile's built-in operators to get the condition codes associated with execution of the operator. For example, the expression `(5-7)?cc(CCR)` computes the result -2, and also saves condition codes for the subtraction into variable `CCR`. FastSim simulators run on SPARC hosts, so the SPARC condition code setting instructions are used to generate condition code values for Facile. Hand coded assembly language functions for each arithmetic operator in Facile use native SPARC instructions to compute the operator result and generate its condition code values. Then condition codes are read from the host `%ccr` or `%fsr` registers and saved in the given Facile condition code variable, while the operator result is returned.

5.1.2.3. Support for the Solaris Operating System

FastSim simulators run on SPARC processors under the Solaris operating system. FastSim is not host independent, because it relies on SPARC ISA features (e.g., SPARC condition codes) and the particular way Solaris lays out process memory. For simplicity, my simulated target architectures have also modeled a SPARC ISA under Solaris. Common functions to simulate the SPARC/Solaris interface are included in FastSim's run-time library. These functions emulate Solaris system calls, register window spills and restores, and access to special architecture registers like the SPARC floating-point state register (FSR). To emulate different operating systems or ISAs, programmers can write their own emulation routines.

In a normal executable, Solaris system calls are made using the SPARC trap instructions, e.g., trap always (`ta`), trap on equal (`te`), etc. To simulate these instructions, Facile simulator code calls the external library function `trap_sparc` (called `xtrap_sparc` in C and implemented in libfs.a). This function expects the Facile simulator code to implement a function—`get_R4`—that gets simulated register values, and uses this function to get the system call arguments. Then it translates any pointer arguments into simulator addresses (by or'ing them with 0x80000000), and calls an assembly language routine that contains a native `ta` (trap always) instruction. Finally, `trap_sparc` translates any pointer return values back into the simulated address space and calls `set_R4` (provided by the Facile programmer) to copy result values back into Facile variables.

The SPARC ISA uses hardware register windows to optimize function calls and returns. SPARC's `save` and `restore` instructions push and pop the register window stack respectively. Since there are a limited number of register windows implemented in hardware, the operating system must spill and restore these windows whenever the function call depth exceeds the number of hardware register windows. The external library functions `save_regs` and `restore_regs` (called `xsave_regs` and `xrestore_regs` in C) spill and restore simulated register windows respectively. Like `trap_sparc`, these functions use `get_R4` and `set_R4` to access simulated registers stored in Facile variables. The algorithms to spill and restore registers are identical to the algorithms used in the Solaris operating system, so the register windows behave the same when simulated as when the target is executed directly.

Some special purpose registers are easier to emulate in C and assembly code than to model completely in Facile. An example of this is the floating-point state register (FSR). Among other things, the FSR controls the floating-point round direction and whether to trap for various floating-point exceptions. The external functions `get_FSR` and `set_FSR` (implemented in SPARC assembly code as `xget_FSR` and `xset_FSR`) maintain the FSR value for a SPARC simulator. The simulated FSR value is stored in the actual host FSR register, so subsequent floating-point operations behave correctly for the simulated executable.

5.2. The Fast-Forwarding Optimization

One example of a memoizing simulator was described in Chapter III, but that simulator was painstakingly optimized by hand. This section describes how the Facile compiler and the FastSim run-time library generate the fast-forwarding optimization automatically. The idea is to generate two simulator versions (as in figure 5.6): a slow but complete version that stores result data in the memoization cache, and a fast version that uses memoized results to skip over repeated parts of the simulation. When combined with run-time support for switching between the two simulator versions, this implements fast-forwarding.

5.2.1. Overview of Simulator Memoization

The primary contribution of the Facile compiler it its automatic separation of a simulator into slow and fast versions. First, the compiler performs binding-time analysis to determine which

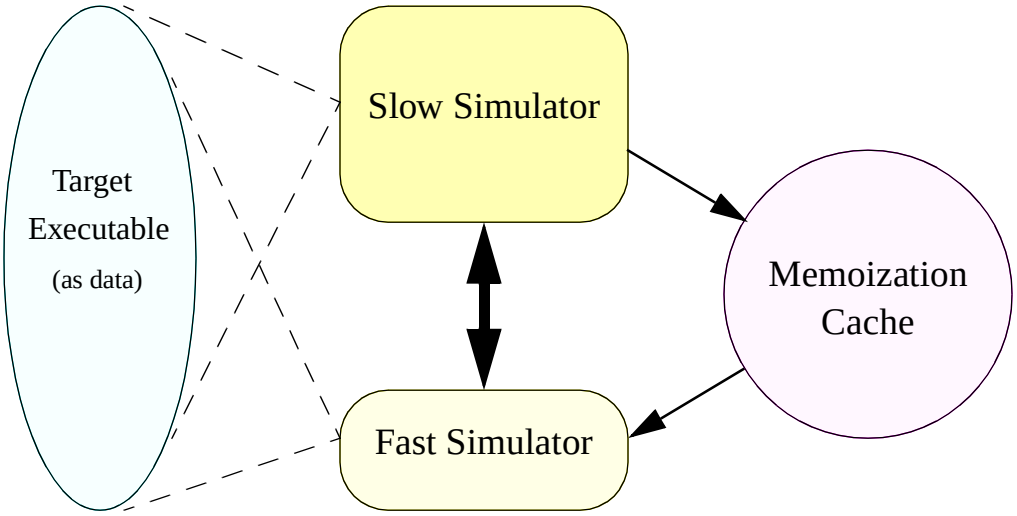

FIGURE 5.6: Structure of a fast-forwarding simulator. Two versions of the same source simulator cooperate to simulate a target executable. The slow simulator version stores result data into the memoization cache, which is later used by the fast simulator to skip repeated simulation. Only one simulator versions runs at any given time.

parts of a simulator can be left out of the residual (fast) simulator—i.e., skipped over by fast-forwarding. Code that cannot be skipped—i.e., labeled dynamic by binding-time analysis—is then split into single-entry/single-exit basic blocks, and each block is assigned an action number. A fast version of the simulator is generated, which loops over a single C switch statement that has one case for each action number. Every iteration of the loop reads one action number from the memoization cache and executes the dynamic basic block code associated with that action number. Additional data may be read from the memoization cache for run-time static data values that are used in dynamic basic-block code.

```
fun main(pc)
{
    val npc = pc + 4;
    switch(pc) {
     pat add:
        if(i) R[rd] = R[rs1] + simm13?sext(32);
        else R[rd] = R[rs1] + R[rs2];
     pat beq:
        if(CCR?bit(2))
            npc = pc + disp22?sext(32)<<2;
    }
    init = npc;
}
```

FIGURE 5.7: Binding-time division of a simple simulator. For simplicity, only two kinds of instructions are simulated: add and branch-on-equal. The instructions are encoded like instructions in the SPARC ISA, but their semantics are simplified to ignore register windows, branch delay slots, and other complexities. Dynamic code is underlined. All other code is either static or run-time static.

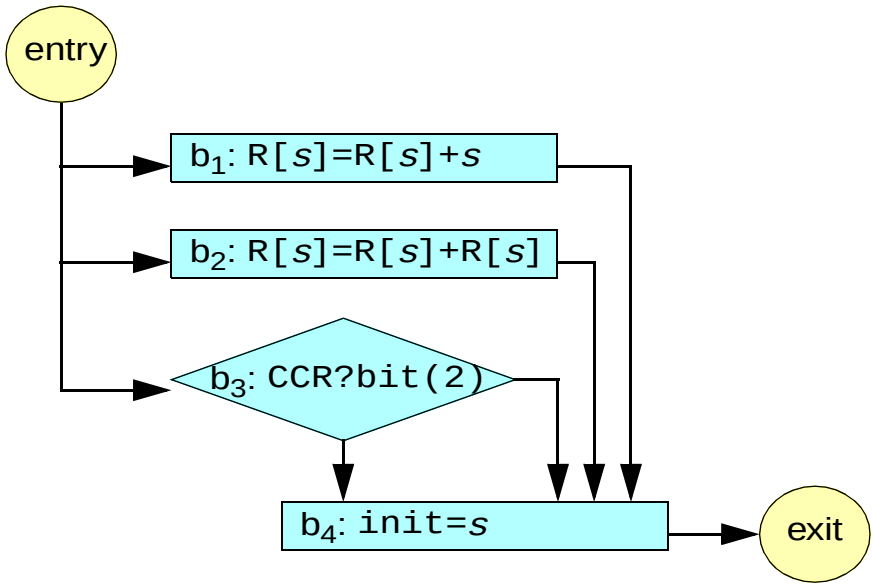

FIGURE 5.8: Dynamic control flow graph and basic blocks. This is the dynamic control flow graph for the example simulator in figure 5.7. It has four dynamic basic blocks numbered 1 through 4.

Figure 5.7 illustrates run-time static and dynamic code in a simple simulator example. Underlined code is dynamic and cannot be skipped by fast-forwarding. The remaining code is either static or run-time static and can be skipped.

Figure 5.8 shows the control flow graph and basic blocks corresponding to the dynamic code in figure 5.7. In this example, each dynamic statement has its own block. In a richer simulator, a dynamic basic block would contain multiple statements. These basic blocks contain only dynamic code; the run-time static sub-expressions are replaced by the place-holder `s`. Notice that block b_3 differs from the others. This block contains a condition expression in an if statement. The result of this dynamic expression determines the simulator's control flow path, so this action can have more than one successor action sequence stored in the memoization cache.

Figure 5.9 shows the fast simulator generated for the simulator in figure 5.7. For simplicity, this example is written in Facile-like pseudo code, although Facile's compiler actually generates C code. Important elements of this code include: a function name—`ff_main`—known to the run-time library, an outer loop that executes dynamic basic blocks until there is a memoization miss, and a switch statement that selects dynamic basic blocks associated with action numbers read from the memoization cache. Regardless of the original scope of variables in Facile, the variables used by `ff_main` are all declared the global scope, so their values will be available to the slow simulator following a memoization miss.

```
fun ff_main(action) {
    while(true) {
        switch(action.number) {
         case INDEX_ACTION:
            verify_static_input();
         case 1:
            read_static_data(r1, r2, t1);
            R[r1] = R[r2] + t1;
         case 2:
            read_static_data(r1, r2, r3);
            R[r1] = R[r2] + R[r3];
         case 3:
            val t2 = CCR?bit(2);
            ff_find_result_1(t2);
         case 4:
            read_static_data(npc);
            init = npc;
        }
        action = get_next_action();
    }
}
```

FIGURE 5.9: Sample of fast simulator code. The above example shows fast simulator pseudo-code for the simulator in figure 5.7. This code is written in a Facile-like syntax, although the Facile compiler actually generates C code.

Figure 5.10 shows the slow simulator generated for the Facile code in figure 5.7. Extra code that is not directly translated from Facile source code is shown in bold. The slow simulator contains a complete copy of the simulator's code, plus additional code to write data into the memoization cache and restart the slow simulator after a memoization cache miss. Dynamic basic-block code that also appears in the fast simulator is only a subset of the code in the slow simulator, and is mixed in with a simulator's static and run-time static code. Additional code writes action numbers into the memoization cache before the first statement of each dynamic basic block. Run-time static data that is used in dynamic expressions is also written into the memoization cache. These

```
fun xmain(s_pc) {
    val s_npc = s_pc + 4;
    switch(s_pc) {
     pat add:
        if(s_i) {
            ff_write_action(1);
            val s_t1 = s_simm13?sext(32);
            ff_alloc(s_r1, s_r2, s_t1);
            if(!recover) R[s_r1] = R[s_r2] + t1;
        } else {
            ff_write_action(2);
            ff_alloc(s_r1, s_r2, s_r3);
            if(!recover) R[s_r1] = R[s_r2] + R[s_r3];
        }
     pat beq:
        ff_write_action(3);
        val s_t2;
        if(recover) {
            ff_recover_result_1(s_t2);
        } else {
            s_t2 = CCR?bit(2);
            ff_write_result_1(t2);
        }
        if(s_t2) s_npc = s_pc + s_disp22?sext(32)<<2;
    }
    ff_write_action(4);
    ff_alloc(s_npc);
    s_init = npc;
    if(!recover) init = s_init;
}
```

FIGURE 5.10: Sample of slow simulator code. The above example shows slow simulator pseudo-code for the simulator in figure 5.7. This code is written in a Facile-like syntax, although the Facile compiler actually generates C code. Extra code added to implement fast-forwarding is shown in bold. Static/run-time static versions of variable names are preceded by "s_".

cached actions and run-time static data values are interpreted by the fast simulator to accelerate later simulation.

The FastSim run-time system switches between slow and fast simulation, using the fast simulator whenever memoized data is available. In each iteration of FastSim's implicit outer loop,

`main`'s argument values are looked up in the memoization cache index. If an existing entry is found, then the fast simulator is called and interprets actions already stored in the memoization cache to execute the simulation. If `main`'s argument values are not found in the memoization cache index, then a new index entry is created and the slow simulator is called to execute the simulation and write more actions into the cache.

Dynamic conditions in the fast simulator (e.g., if or switch statements that test a dynamic value) may result in a memoization cache miss. Dynamic results from previous executions of a condition by the slow simulator are listed in the memoization cache and associated with the action sequences that follow each different result. When the fast simulator encounters a dynamic condition, it calculates a result value and looks for that value in the list of previously encountered results. If the result has already been encountered, then fast simulation continues. Otherwise there is a memoization miss, the fast simulator fails, and simulator execution switches back to the slow simulator.

Recovering from a memoization cache miss is difficult. FastSim rolls back simulation to the beginning of the most recent memoized call to `main`, gets `main`'s current arguments from the memoization cache index, then calls `main` in the slow simulator. The difficulty is that the fast simulator may have already updated global variables with new dynamic data, or called external functions that execute an unknown amount of dynamic code and cannot be rolled back. The solution is to not execute any dynamic code until the slow simulator catches up to the point where fast simulation failed.

In order to temporarily skip dynamic code in the slow simulator, there are two versions of every variable: one for dynamic values and the other for non-dynamic values. Whenever a variable is dynamic, its value is stored in the dynamic version, otherwise its value is stored in the non-dynamic version. All dynamic variable versions are declared in C's global scope so they can be shared between the fast and slow simulators. Non-dynamic variables are declared normally, in local or global scopes as specified by the Facile source code. To recover from a memoization miss, the slow simulator only executes non-dynamic code until reaches the point where fast simulation failed, so it only reads and writes non-dynamic variables. Once the slow simulator catches up, all dynamic and non-dynamic variables contain up to date values for that point in the simulation and the slow simulator continues normally, executing both dynamic and non-dynamic code.

5.2.2. Binding Time Analysis (BTA)

Binding time analysis is performed by the Facile compiler. Its purpose is to label every expression in a subject program as *static* (i.e., computable at compile time), *run-time static* (i.e., computable by the slow simulator but skipped by the fast simulator), or *dynamic* (i.e., must be executed every time). This labeling is accomplished by iterating over a program's control flow graph until the binding-time information reaches a fixed-point and no longer changes. The following sections describe Facile's binding time analysis in more detail.

5.2.2.1. BINDING-TIME DATA

Binding-time labels used in this analysis are: KNOWN for static data whose value is known to the compiler, STATIC for static data that the compiler did not bother to compute, RT-STAT for

run-time static data, and DYNAMIC for dynamic data. Structured data, such as in arrays and structures, can have different binding time labels for each element of the array or structure. Structured data that has different binding-time labels for different elements is said to have a mixed binding time.

One reason for using KNOWN labels is so binding-time analysis can use the value of static data to analyze binding-times more accurately. For example, consider an array with mixed binding times. If an element is selected from this array using a KNOWN index value, then the binding time of the actual array element can be determined. If the index value is not KNOW, then the binding time of the selected element must be the worst case binding time of all the array elements.

Facile's compiler uses a polyvarient binding-time analysis. This means that functions can have different binding-time labelings—called *divisions*—when called from different call sites. If functions were only allowed one division (i.e., monovarient division), then dynamic data passed to the function at one call site could force all call sites for the same function to be labeled dynamic. Polyvarient divisions allow more code to be labeled with earlier binding times (e.g., KNOWN, STATIC, and RT-STAT indicate earlier binding times than DYNAMIC), since late binding times at one call site do not influence other call sites. But polyvarient division can also increase the size of generated code. This increase occurs because different versions of a function are generated for each different division of the function used in a program.

5.2.2.2. FIXED-POINT ITERATION

Binding-time analysis is performed using fixed-point iteration, an algorithm used to solve many data-flow analysis problems. The idea is to propagate binding-time data through the program's control-flow graph (CFG), and iterate over loops in the CFG until the binding-time data stops changing. Figure 5.11 outlines the fixed-point iteration algorithm used for Facile's BTA.

Inter-procedural binding-time analysis is simplified by the absence of recursion in Facile programs. Because there is no recursion, there are no cycles in the function call graph. Facile's inter-procedural binding-time analysis starts at the top of the call graph (with the `main` function) and calls `BT_analysis` recursively for any function calls it encounters. A limited form of memoization is used in the compiler to optimize this process. The result of each analysis is cached, indexed by the called function's name and its initial binding time division. If the called function has already been analyzed with the same initial division, then the cached results are used instead of re-analyzing the called function.

To start off binding-time analysis, the `BT_analysis` routine is called to analyze `main`—Facile's top level simulator function. The binding-time division at the start of `main` labels global variables as DYNAMIC, and `main`'s parameters as RT-STAT. As analysis progresses, KNOWN values arise from integer constants and integer operators applied to KNOWN values. STATIC values arise from non-integer constants and operations on other STATIC values. RT-STAT values

```
let start_set = a set for storing pointers to statements
        in a control-flow graph, initially empty.

function BT_analysis
argument CFG (the control flow graph of the current function)
argument BTD_start (the binding time division at the start of this function)
{
    store BTD_start with the first statement in the CFG
    start_set = a singleton set containing
        the first statement in the given CFG.

    while start_set is not empty {

        let start = any one element in start_set
        remove start from start_set

        let BTD = the binding-time data associated with statement start
        call iteration_helper on CFG, start, and BTD
    }
}

function iteration_helper
arguments CFG, start, and BTD
{
    let S = start
    loop {
        if S != start and S has > 1 predecessors in CFG {
            merge BTD with the binding time data associated with S
                and associate the new binding-time data with S

            if the binding-time data for S has changed then
                add S to start_set

            return from iteration_helper
        }

        evaluate statement S for its effect on the binding-time data in BTD

        if S does not have exactly 1 successor in CFG {
            for each successor of S in CFG
                call iteration_helper on CFG, S, and BTD

            return from iteration_helper

        } else
            S = the successor to S
    }
}
```

FIGURE 5.11: Fixed-point iteration algorithm for binding-time analysis.

arise from code that depends only on KNOWN, STATIC, or RT-STAT data. Finally, DYNAMIC values result from any code that depends on at least one DYNAMIC value.

A vital step in the binding time analysis algorithm is the merging of two sets of binding time data. In order to guarantee termination of the analysis, the set of possible binding time divisions and the merge operation define a lattice with finite ascending chains. That is, binding time data can only be merged with other binding time data to produce different binding times a finite number of times, before a fixed point is reached. Facile's binding time data has this lattice property because: 1) The result of merging individual binding time labels is the label with the later binding time;[1] 2) An individual binding time label can change at most three times before it is DYNAMIC, and cannot change any more after that; And 3) for any given program, there are a finite number of program variables for which binding time data is calculated. The lattice property guarantees that BTA's fixed point iteration will terminate, although it could take a long time to do so. In practice, the performance of Facile's BTA is not a problem, and it is fast enough to analyze complex micro-architecture simulators in an acceptable amount of time.

In addition to calculating a division of a program's data, Facile's BTA labels every statement and expression as either STATIC, RT-STAT, or DYNAMIC. STATIC code can be evaluated at compile time (although the current Facile compiler does not do this). RT-STAT code is executed by the slow simulator, and skipped by the fast simulator. DYNAMIC code is executed by both the slow and fast simulator versions. Unlike data labels, code is never labeled KNOWN, since code

1. Except for when two KNOWN labels are merged. The result is KNOWN only if the two values are the same, otherwise it is STATIC.

labels indicate the earliest time it can be evaluated and not its result value. Code labels are used by later compiler stages to generate fast and slow versions of a simulator.

5.2.2.3. External Functions & Variables

One complication in BTA is the handling of external functions and variables, because external code is not analyzed by the Facile compiler and its effects are unknown. The worst case must be assumed. A call to external code may change values stored in external variables, or call other Facile functions that are externally visible. Hence, after any call to a function that is implemented externally, all external variables become dynamic and all global variables written to by external functions in Facile also become dynamic.

Global variables that may be read by an external function in Facile must also be made dynamic before any call to external code. This is because variables all have two versions, to support recovery from a memoization miss: one version is used to store dynamic values, and the other version stores non-dynamic values. Facile functions that are called from external code can only access the dynamic versions of variables. Hence any variable that may be read by Facile code called from an external source, must be made dynamic before any call to external code.

Before starting binding-time analysis, the Facile compiler finds the set of all variables that are declared external. It then traverses the Facile code for each function to find every global variable accessed by that function or by a function it calls. All global variables accessed by an externally visible Facile function are added to the set of external variables. Binding-time analysis labels all

the variables in this set of external variables DYNAMIC whenever a call to external code is encountered.

5.2.2.4. VISUALIZING BTA RESULTS

Although the effect of individual pieces of source code on binding-time data is easy to understand, it is difficult to predict the results of BTA when applied to an entire simulator. A programmer usually has a general idea how binding-time analysis should turn out, but small errors in simulator design or implementation can result in more code being labeled dynamic then expected. For this reason, the Facile compiler optionally generates an annotated version of simulator source code, showing the binding-time labels derived for every piece of Facile code. By iteratively changing the source code and re-generating this annotated output, a programmer can arrive at the desired division of code into static, run-tim static, and dynamic binding-times.

Annotated Facile source is written out in HTML format, suitable for viewing with any web browser. Code labeled STATIC by BTA is colored black in the HTML file, RT-STAT code is colored green, and DYNAMIC code is colored red. Code with a mixed binding-time—i.e., it produces data that is part dynamic and part non-dynamic—is colored turquoise. Each annotated function definition is put in its own HTML file, and function calls are marked as hyper-text links to the HTML file for the called function. By examining this annotated Facile source, it is easy to see the binding-time division derived by the Facile compiler.

Annotated code, in HTML files, has the same spacing and line breaks as in the Facile source file. To accomplish this, the compiler keeps track of the precise character positions of every code fragment parsed from a Facile source file. This position information is kept throughout all stages of compilation, and is also used to produce informative error messaged and `#line` directives in generated C files. After BTA, a simulator's code tree is traversed and color annotations are computed for every statement, expression, and sub-expression, with the annotation's color determined by the code's binding-time label. After computing all the color annotations, the Facile source file is re-read from disk and written out as HTML, with extra HTML tags adding color at the character positions corresponding to each annotated code fragment.

5.2.3. The Memoization Cache

The memoization cache is a single contiguous region of memory, plus a hash table for fast lookup of cache index entries. It is managed jointly by code generated by the Facile compiler and routines implemented in FastSim's run-time library (libfs.a). Figure 5.12 illustrates the structure of data as it is stored in the memoization cache. The kinds of data stored in the memoization cache include: index entries, action numbers, run-time static data records, and dynamic result lists. Most of these entries are stored sequentially in memory in the order they were generated by the slow simulator.

At the start of simulation, the memoization cache is empty. During slow simulation, new entries are allocated onto the end of previously cached data by simply incrementing a pointer.

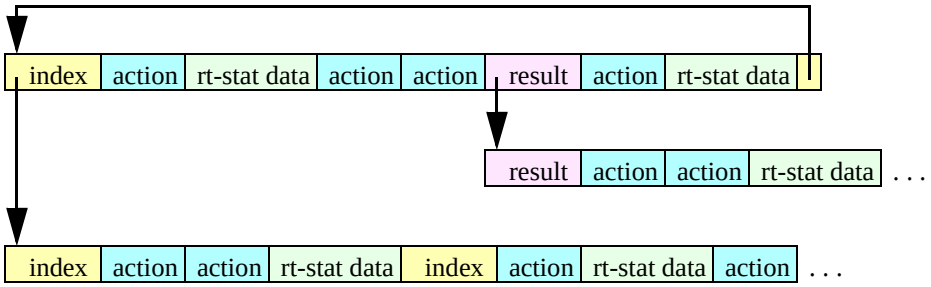

FIGURE 5.12: Structure of data in the memoization cache. The cache contains index entries, action numbers, run-time static data records, and result lists. These entries are usually arranged sequentially in memory. Result lists and lists of index entries represent alternative paths that could be taken by an execution, and are represented non-sequentially by linked lists. Loops in the data may occur when the slow simulator finds an existing index entry. When this happens, a pointer to the matching index entry is appended to the end of the current action sequence, and the fast simulator is started for the first action following the matched index.

Entries are not removed from the memoization cache until the cache is full. When the cache is full and more space is needed, the entire cache is flushed by resetting the allocation pointer back to the beginning of the cache and clearing the index hash table. New data, or a new copy of old data that was flushed, is generated by the slow simulator in subsequent simulation. This and other replacement policies were studied with FastSim Version 1 (discussed in chapter III), and this cache flush replacement policy was found to be both easy to implement and the most efficient alternative of those studied.

5.2.3.1. INDEX ENTRIES

Index entries in the memoization cache represent all the places where the simulator can switch from slow to fast simulation. Index entries are generated for each call to `main` in the slow simulator, and each entry contains a copy of the argument values for that call. To accelerate index

lookup, index entries are pointed to by a hash table, although the index entries themselves are stored in the memoization cache. Cached simulation results associated with an index entry are located sequentially after the index entry in memoization cache memory.

When the slow simulator finds an existing index entry (at the start of a call to `main`), a pointer to the existing entry is appended to the end of the current action sequence in the memoization cache and control is passed to the fast simulator. The fast simulator reads the sequence of actions following the matched index entry and interprets them. Fast simulation does not stop when another index entry is encountered at the end of an action sequence in the cache. Instead, it verifies that the current values of `main`'s arguments match the values stored in the index entry. If the values match, then the fast simulator skips past that index entry to the next action number. Because it is possible for an index value to not match, alternate index entries for this point in the simulation are arranged in a linked list. If none of the index entries in this linked list match, then a new index entry is created, added to the list, and control is passed back to the slow simulator which generates action sequences for the new index.

Figure 5.13 shows a sequence of index entries in the memoization cache, corresponding to three SPARC instructions simulated by the simulator in Appendix B. Each index entry contains a copy of `main`'s arguments—`pc` and `npc`. The left most index entry corresponds to the call to `main` that simulated the `tst` instruction at address 0x11194. The next index entry corresponds to simulating the `be` (branch on equal) instruction. The last two index entries correspond to executing the `ld` instruction in the branch delay slot, when the branch fell through and when the branch

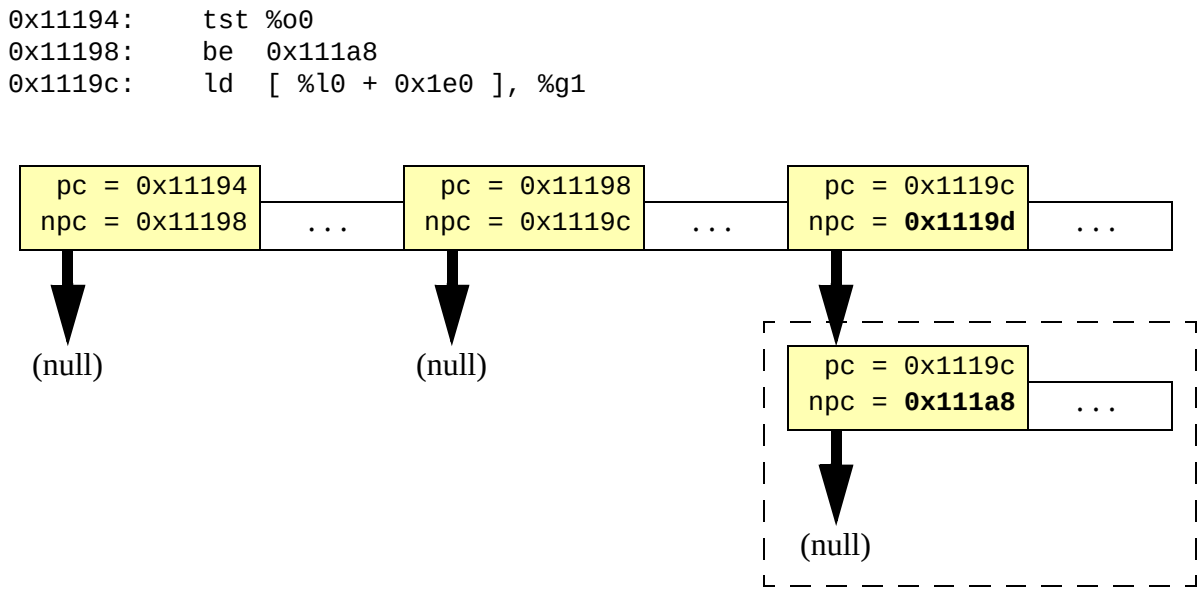

FIGURE 5.13: Memoization Index Entries. This figure shows the sequence of index entries that are generated by the simulator in Appendix B, corresponding to the given 3 target instruction sequence. The three SPARC instructions test a register, branch if zero, and load a value respectively. Index entries are shown as large boxes, and their associated action sequences are abbreviated as short boxes with ellipses (...). Note that the action sequence associated with the second index entry has two successors. Its first successor corresponds to the branch not taken, and its second successor is generated later when the branch is taken.

was taken respectively. This example assumes that the branch fell through the first time it was simulated, and was taken in some later simulation, so the fall-through case is listed first. The index entry for the branch-taken case does not follow the previous action sequence sequentially in the memoization cache, because it is generated later. Instead it is pointed to by the fall-through index entry, forming a linked list of possible successor indexes.

5.2.3.2. ACTION NUMBERS & RUN-TIME STATIC DATA RECORDS

An action number identifies a dynamic code fragment to execute in the fast simulator. These dynamic code fragments are single-entry/single-exit basic blocks in the dynamic control flow

graph. A sequence of action numbers stored in the memoization cache strings together several dynamic basic blocks to replay the dynamic simulation originally executed by the slow simulator. Hence, a fast simulator can be thought of as an interpreter that evaluates code specified by sequences of action numbers.

Some action numbers are followed by a structure containing run-time static data stored in the memoization cache. This data is used by the dynamic basic block associated with the given action number. Whenever code in one the these dynamic basic blocks uses a run-time static value, that value must be stored in the memoization cache. A structure is allocated following the basic block's action number in the cache, and every run-time static value used by the basic block is given a field in this structure.

5.2.3.3. Dynamic Result Lists

Sometimes a dynamic value must be changed into a run-time static value. This happens at the start of a call to the `main` function, since dynamic values generated by a previous call to `main` may be passed as run-time static arguments to this call. (This is why index entries are grouped into linked lists, to handle alternate dynamic values to `main`.) Dynamic data is also made run-time static for every if or switch statement that tests a dynamic condition, so the control flow of a memoized program is always run-time static. These dynamic results are also grouped into linked lists, with each entry corresponding to a dynamic value that has already been seen by the slow simulator and has an action sequence associated with it (see figure 5.14). Finally, dynamic data can be made run-time static by explicit use of the `?static` attribute in Facile, and this is also

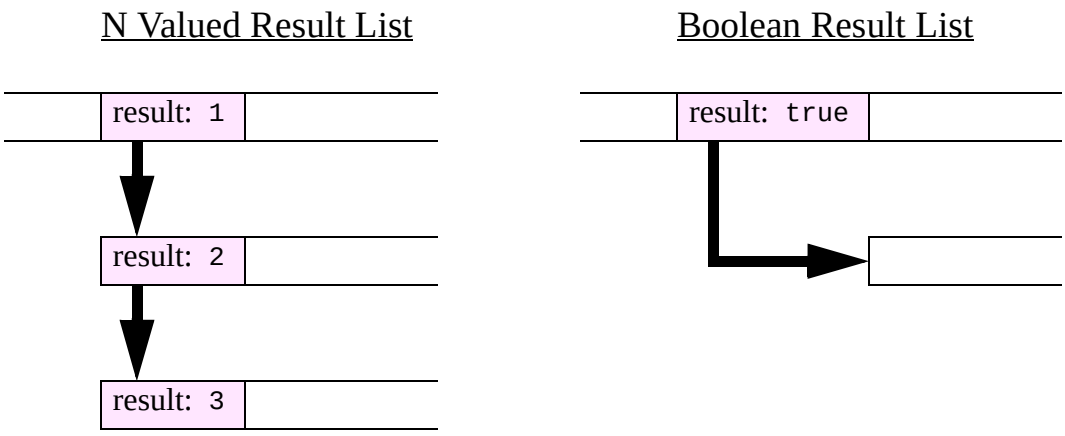

FIGURE 5.14: Memoization Result Lists. This figure shows how result lists are laid out in memoization cache memory. The first entry in a result list follows the previous action sequence sequentially, and points to a linked list of alternate result list entries. Result lists to test boolean values only need to list the first value, because there are only two possible boolean values (`true` and `false`). The action sequence associated with each dynamic result value follows the corresponding result list entry sequentially.

handled with a linked list of previously simulated values. The difference between lists of index entries and other dynamic result lists is that only index entries can be looked up in the memoization cache index.

When making a dynamic value run-time static, the fast simulator tries to match the current dynamic value with one of the values listed in the memoization cache. If a match is found, then the action sequence that follows the matching list entry is used. A memoization miss occurs when the current dynamic value does not match any value in the list. When this happens, FastSim's run-time library searches backward through the previous action sequences to find the most recently visited index entry, then restarts the slow simulator with the argument values found in that index entry.

The FastSim run-time library supports three different representations of dynamic result lists in the memoization cache. These representations correspond to boolean results (i.e., with only two possible values), results values that fit in 8 bits or less, and other values of arbitrary width. Different representations are used for these three cases to minimize the space required to store these results, and to optimize the code needed to generate them in slow simulation and test them in fast simulation. For example, most dynamic result lists are for if statements that test dynamic boolean expressions. These boolean results lists do not have to store a result value with every element of the list, because there are at most two list elements and if the first one does not match then the second one will.

5.2.4. The Fast Simulator

The fast simulator reads previously generated result data from the memoization cache and skips over repeated (i.e., run-time static) work in the simulation. The Facile compiler generates the fast simulator with the following steps: 1) Build a control-flow graph that contains only dynamic simulator code (dCFG); 2) Assign action numbers to basic-blocks within a dCFG; And 3) generate C code for each basic-block and put it in a switch case labeled by the block's action number. The resulting fast simulator loops over this switch statement, repeatedly reading an action number from the memoization cache then executing the code associated with that action, skipping all run-time static simulator code.

5.2.4.1. Dynamic Control-Flow Analysis

After binding-time analysis, the compiler has control-flow graphs for all functions in a subject simulator, and has binding-time labels that identify which code is dynamic. The next step is to construct a control-flow graph for the entire simulator that contains only (and all) the dynamic code. First a control flow graph of the entire program is constructed from the program's call graph and the control-flow graphs of each individual function. Then dynamic control-flow analysis traverses the program's CFG, editing out non-dynamic nodes to produce a dynamic control-flow graph (dCFG) for the entire program.

Figure 5.15 describes the algorithm for calculating a program's dCFG. This is another fixed-point iteration algorithm. It computes the set of dynamic predecessors—the last dynamic statements seen prior to the current statement in the full CFG—for every statement in a program. The dCFG is then constructed by adding graph edges to each dynamic statement from its dynamic predecessors.

5.2.4.2. Dynamic Basic Blocks & Action Numbers

Action numbers are assigned to every single-entry/single-exit basic block in the dCFG. These dynamic basic blocks are the building blocks for fast-forwarded simulation. Fast simulation involves executing sequences of these basic blocks directed by sequences of action numbers stored in the memoization cache.

```
let start_set = a set for storing pointers to statements
        in a control-flow graph, initially empty.

function dynamic_cfa
argument CFG (the program control flow graph)
{
    store an empty predecessor set with the first statement in the CFG
    start_set = a set containing one element,
        the first statement in the given CFG.

    while start_set is not empty {

        let start = any one element in start_set
        remove start from start_set

        let pred_set = the predecessor set associated with start
        call iteration_helper on CFG, start, and pred_set
    }
}

function iteration_helper
arguments CFG, start, and pred_set
{
    let S = start
    loop {
        if S != start and S has > 1 predecessors in CFG {
            union pred_set with the predecessor set associated with S
                and associate the new predecessor set with S

            if the predecessor set for S changed then
                add S to start_set

            return from iteration_helper
        }

        if S is labeled DYNAMIC then
            pred_set = the singleton set { S }

        if S does not have exactly 1 successor in CFG {
            for each successor of S in CFG
                call iteration_helper on CFG, S, and pred_set

            return from iteration_helper

        } else
            S = the successor to S
    }
}
```

FIGURE 5.15: Dynamic control-flow analysis.

```
let next_action = 1

function dynamic_split
argument dCFG (a dynamic control flow graph)
argument start (a statement in the dCFG)
{
    if start already has an action number then
        return from dyn_split;

    associate the action number next_action to statement start
    next_action = next_action + 1

    let S = start_action
    while S has exactly 1 predecessor and 1 successor in dCFG
        S = the successor to S in dCFG

    if S has > 1 predecessor {
        call dynamic_split on dCFG and S
        return from dyn_split
    }

    if S has > 1 successor in dCFG {
        for each successor S' of S in dCFG
            call dynamic_split on dCFG and S'
    }
}
```

FIGURE 5.16: Dynamic basic blocks and action numbers.

Basic blocks are found, and action numbers assigned, with a depth first traversal of the dCFG. Every dCFG node is visited exactly once, and a new action number is assigned to each node that has multiple predecessors or that follows a node with multiple successors. Pseudo-code for this algorithm is given in figure 5.16.

5.2.4.3. GENERATED C CODE

Figure 5.17 shows part of the fast simulator code generated for the Facile code in Appendix B. Important elements of this code include: a function name—`ff_main`—known to the run-time

```
void
ff_main(action_t* ff_action)
{
    void *ff_current;
    for(;;) {
        ff_current = (char*)(ff_action + 1);

        switch(ff_action->action) {

         case INDEX_ACTION:
            tuple_1556_pack_data(xinit);
            ff_current = ff_index_action(ff_action);
            break;

         case 3:
         L_3: {
             if(fs_cond_xcc_206) {
                 ff_current = ff_find_result_1(ff_current,1);
             } else {
                 ff_current = ff_find_result_1(ff_current,0);
                 goto L_933;
             }
             break;
         }

         case 933:
         L_933: {
             record_2386 *ff_rts_data = (record_2386*)ff_current;
             (char*)ff_current += (sizeof(record_2386) + 3) & ~0x3;
             fs_nPC2_2 = ff_rts_data->t_2385;
             break;
         }
        }

        ff_action = (action_t*)ff_current;
    }
}
```

FIGURE 5.17: Sample of fast simulator C code. The above example shows selected parts of the fast simulator C code generated for the simulator in Appendix B. Function `ff_main` takes a pointer to the first action following a matched index entry and runs until a memoization miss occurs. The `INDEX_ACTION` case handles subsequent index entries in the interpreted action sequence by calling the run-time library routine `ff_index_action`. The two action cases shown in this example implement part of the dynamic code for simulating a SPARC branch instruction. Case 3 tests a condition code value and calls `ff_find_result_1` to find memoized data for the boolean result (either 0 or 1) or to handle a memoization miss with a longjmp back to slow simulation. Case 933 updates the Facile variable `nPC2` (called `fs_nPC2_2` in C) with a run-time static branch target address.

library, an outer loop that executes dynamic basic blocks until there is a memoization miss, and a switch statement that selects dynamic basic blocks associated with action numbers read from the memoization cache. Not shown here are all the simulator variable and type declarations. Regardless of the original scope of variables in Facile, the variables used by `ff_main` are all declared the global scope, so their values will be available to the slow simulator following a memoization miss.

During slow simulation, the memoization cache index is checked before every call to the `main` simulator function (called `xmain` in C). If this function's current arguments match an index entry in the memoization cache, then FastSim's run-time library calls `ff_main` instead of `xmain`. A pointer to the first action number following the matched index entry is given as an argument to `ff_main`. Once started, `ff_main` runs until there is a memoization miss—i.e., until there is no data in the memoization cache for some dynamic result. `ff_main` has no return statements, since returning to slow simulation is handled by a jongjmp in the run-time library code that handles memoization misses.

The fast simulator iteratively reads action numbers from the memoization cache and executes the dynamic basic block associated with each action number. The code for a dynamic basic block typically executes just the dynamic expressions from the dynamic statements in the block, then breaks out of the switch statement to fetch another action number from the memoization cache. Some blocks have extra features. If a dynamic expression has a run-time static argument, then the value of that argument is also stored in the memoization cache. When run-time static data is

needed, basic block code begins by declaring a pointer to a run-time static data record stored in the memoization cache immediately after the current action number. In figure 5.17, the case for action number 933 reads a run-time static branch target address from the memoization cache.

If a dynamic basic block ends with a dynamic test—e.g., a Facile if or switch statement that tests a dynamic condition—then code is generated that finds the current condition value among a list of values already stored in the memoization cache. Code for a dynamic test is put at the end of the dynamic basic block. Each branch of the dynamic test calls a function in the run-time library to search the memoization cache for the condition value, then breaks out of the switch to fetch the next action number. Run-time library routines `ff_find_result_1`, `ff_find_result_8`, and `ff_find_result_N` search for 1 bit (boolean), 2-8 bit, and larger result values respectively. Different library functions exist, since these three cases are encoded differently in the memoization cache (see section 5.2.3 for a description of memoization cache organization). If the current result is found in the memoization cache, then a pointer to the corresponding action sequence is returned. Otherwise, there is a memoization miss and the run-time library prepares for return to slow simulation and uses a jongjmp to exit from the fast simulator.

As an optimization, some dynamic basic blocks jump directly to the next dynamic basic block, without reading an action number from the memoization cache. This saves time by not reading a new action number and not executing `ff_main`'s switch statement to select the next block. It also saves space by omitting the action number from the memoization cache. An action number for a basic block is omitted from the memoization cache, if every predecessor to the block

in the dCFG has exactly 1 successor. If any predecessor block has more than 1 successor, then the action number for the current block is needed to distinguish it from the other possible successors. At the end of a dynamic basic block that has only one successor, and if that successor's action number is omitted, then a goto statement is generated to go directly to the successor block. This optimization can be seen in the code for action number 3 in figure 5.17 above.

In the processes of generating code for the dynamic basic blocks of a program, the Facile compiler also constructs structure types to hold the run-time static values used by each basic block. Each basic block that uses run-time static data has its own structure type. Each piece of run-time static data used in a dynamic basic block is given its own field in the structure. The slow simulator stores run-time static data into the memoization cache by allocating the appropriate structure type in the memoization cache before the first statement of a dynamic basic block, and populating its fields with the results of run-time static expressions.

To optimize memoization cache size, run-time static data is stored in fields that are often smaller than the width of the data's type. Often, small run-time static values—e.g., register numbers and small immediate values—are converted to larger types before being used in dynamic expressions. But only the bits belonging to the original smaller value are significant. For example, in the SPARC ISA, many immediate values are stored in 13 bit wide instruction fields, but are sign extended to 64 bit integers before being used. Only the first 13 bits are significant in this case, and they can easily be stored in a 2 byte field in the memoization cache, saving 6 bytes. Size hints are calculated during binding-time analysis, and provide a conservative approximation of the

number of significant bits in each piece of static and run-time static data. When a run-time static value is used in a dynamic expression, this size hint (if available for the particular value) is used instead of the value's type to determine the width of data stored into the memoization cache. Dynamic basic block code in the fast simulator extends these value (with sign extension if necessary) to restore the value to its proper type.

5.2.5. The Slow Simulator

The slow version of a simulator executes all Facile simulator code, skipping nothing, and writes data into the memoization cache for later fast simulation. Slow simulator code is similar to the code generated for a simulator that is not optimized with memoization (as described in section 5.1), but extra code is added to write memoization data and recover from memoization misses. This extra code includes calls to the run-time library that write action numbers and dynamic result values into the memoization cache, extra assignment statements to copy run-time static data into the cache, if statements around dynamic code, and two versions of every variable with extra assignments to copy data between these two versions.

5.2.5.1. TWO COPIES OF EVERY VARIABLE

Every variable has two possible storage locations: one for dynamic values, and the other for static and run-time static values. The dynamic versions of variables are also used by the fast simulator. Regardless of the scope of a variable's declaration in Facile, its dynamic version is declared in C's global scope, so dynamic values computed by the fast simulator are accessible to the slow

simulator after a memoization miss. Run-time static versions of variables are declared in the C scope corresponding to their original Facile scope, and are only used in slow simulator code.

The dynamic version of a variable is used when the variable contains a dynamic value, and the run-time static version is used when the variable contains static or run-time static values. Normally, values move from run-time static storage to dynamic storage locations as a result of dynamic code that uses (run-time) static operands. But run-time static values also become dynamic when two or more control flow paths come together and run-time static values from one path are merged with dynamic values for the same variables from another path. These run-time static to dynamic transitions do not correspond to any statements in Facile source code, so new assignment statements are generated by the Facile compiler to move values from their run-time static to dynamic storage locations.

5.2.5.2. WRITING ACTIONS TO THE MEMOIZATION CACHE

Action numbers and dynamic result values are written into the memoization cache to direct subsequent fast simulation. Action numbers are associated with basic blocks in the dynamic control flow graph (dCFG). The code in these basic blocks is mixed in among run-time static code in the slow simulator. Just before the first statement of each dynamic basic block, a call to the library function `ff_write_action` is inserted into the slow simulator to write the basic block's action number into the memoization cache. The sequence in which action numbers are written into the memoization cache controls the sequence that dynamic basic blocks will be replayed by the fast simulator.

If a dynamic basic block uses any run-time static operand values, then space for a structure to contain these run-time static values is allocated—by calling `ff_alloc`—immediately after the action number is written. A different structure type is defined for each dynamic basic block that needs one. The value of each run-time static operand that is used in a dynamic operation is assigned to a field in the allocated structure, then used to evaluate the dynamic operation in the slow simulator.

It is not always necessary to store the entire run-time static operand value into the memoization cache, if the entire value can be reconstructed from a subset of its bits. For example, small integer values are often read from a simulated instruction, but are extended to a 32 or 64 bit integer representation before being used in a dynamic calculation. In addition to labeling a program with binding-time labels, Facile's binding-time analysis computes a conservative approximation of the minimum number of bits needed to represent static or run-time static values. Space is conserved in the memoization cache by only allocating space for these smaller value widths.

To convert dynamic values into run-time static values (e.g., if and switch statements that test a dynamic condition, or explicit uses of Facile's `?static` attribute) the dynamic values encountered by the slow simulator are written into the memoization cache. Fast simulator code tests its value against the values stored in the memoization cache to verify that memoized results exist for the fast simulator's dynamic result. The slow simulator calls the library functions `ff_write_result_1`, `ff_write_result_8`, and `ff_write_result_N` to store 1 bit

(boolean) values, 2-8 bit values, and arbitrary length values respectively. These functions write dynamic values into the memoization cache, and return a run-time static version of the variable.

During recovery from a memoization cache miss, the functions above return the dynamic value encountered by the failed fast simulation. These values cannot be computed by the slow simulator, since dynamic data in the slow simulator is unreliable until slow simulation catches up to the failed fast simulation. Dynamic code in the slow simulator is also guarded with extra if statements to prevent it from executing during memoization miss recovery. The next section discuses FastSim's memoization miss recovery mechanisms more fully.

5.2.6. Recovering From a Memoization Cache Miss

A memoization cache miss occurs when the fast simulator calls `ff_find_result_1`, `ff_find_result_8`, or `ff_find_result_N` for a result value that does not yet exist in the memoization cache. When this happens, FastSim's run-time library searches backward through the cached action sequences until it finds the most recent index entry. Action numbers and dynamic result values encountered in this backwards search are pushed onto a stack—called the recovery stack—for subsequent use in memoization miss recovery. Since each index entry corresponds to a call to Facile's `main` function, the data associated with the most recent index entry is unpacked to recover `main`'s argument values. The slow simulator is then restarted by calling `main` with these unpacked argument values.

The slow simulator can not immediately begin normal execution. Dynamic versions of variables may contain values from later in the execution, and external functions may have already been called by the fast simulator for this call to `main`. This dynamic behavior was already performed by the fast simulator and cannot be rolled back. Ideally, the slow simulator should be started at the same point that fast simulation failed, but this is not possible, because run-time static values that were not computed by the fast simulator are needed by the slow simulator.

The solution is to only execute static and run-time static code in the slow simulator, until it catches up to the failed fast simulation. To do this, every dynamic statement in the slow simulator is guarded by an if statement, and is only allowed to execute after the slow simulator has caught up. Library calls that allocate action numbers in the memoization cache simply compare the requested allocations against the action numbers in the recovery stack, constructed just after the fast simulator failed. The sequence of action numbers encountered by the slow simulator should be exactly the same as the sequence executed by fast simulation, up to the point where the memoization miss occurred. Dynamic result values cannot be computed by the slow simulator during recovery, so the values used in fast simulation are read from the result stack as needed. After all action and result values on this stack have been replayed by the slow simulator, memoization miss recovery is finished and the slow simulator is allowed to execute dynamic as well as static and run-time static code.

CHAPTER VI: Writing Efficient Memoizing Simulators (& Performance Results)

Fast-forwarding is a very effective optimization for accelerating out-of-order micro-architecture simulation. This was demonstrated with FastSim Version 1 (in Chapter III), where memoization achieved an order of magnitude speedup over simulation without memoization. FastSim Version 2 automatically optimizes simulators written in Facile to use memoization, and for out-of-order simulation it also achieves and order of magnitude speedup over simulation without memoization. This result and an out-of-order simulator implemented in Facile are discussed in section 6.1.

Although FastSim v.2 makes memoization accessible to simulator writers, they may not have the background to effectively use this optimization. Designing a simulator that uses memoization effectively requires an understanding of the fast-forwarding optimization, and a knowledge of simulator implementation alternatives and their affect on performance. Section 6.2 catalogs the factors that influence the performance of a fast-forwarding simulator, and derives the equation governing fast-forwarding performance. Section 6.3 discusses how to design a simulator in Facile to get the most out of FastSim's fast-forwarding optimization. Experimental results with variations on the simple simulator in Appendix B are used to illustrate the effect of several simulator design decisions.

6.1. Out-Of-Order Processor Simulation

As shown in Chapter III, memoization greatly improves the performance of an out-of-order micro-architecture simulator. The same is true of an out-of-order micro-architecture simulator written in Facile and optimized with memoization automatically. My original memoizing simulator—FastSim v.1—ran with a 2,000 times slowdown without memoization and only a 250 times slowdown with memoization. An out-of-order simulator written in Facile is not as fast, due to inefficiencies in the compiled code produced by the Facile compiler and because it does not use direct execution, but it still out-performs contemporary out-of-order simulators, such as SimpleScalar, which has approximately a 4,000 times slowdown. Without memoization, my out-of-order micro-architecture simulator written in Facile simulates target executables with a 15,000 times slowdown on average. With memoization it simulates with an average slowdown of only 1,500, an order of magnitude improvement over simulation without memoization, which makes it faster than SimpleScalar.

Besides being optimized automatically, this out-of-order simulator in Facile executes target instructions in the order specified by the out-of-order pipeline model. This is an improvement over FastSim v.1, which executed target instructions in program order using direct execution and simulated the out-of-order pipeline timing separately. Since correct execution of a target program now depends on correct implementation of the simulated out-of-order execution engine, measured behavior of the modeled micro-architecture is more likely to be correct. This does not imply that the simulator correctly models a particular processor micro-architecture, but that the modeled architecture is actually capable of correct out-of-order execution.

As with program optimization using partial evaluation, programmer knowledge about out-of-order micro-architecture simulation and the implementation of fast-forwarding is needed to effectively use this optimization. Section 6.1.1 describes the design of the out-of-order simulator in Facile to make effective use of FastSim v.2's fast-forwarding optimization. An analysis of the performance and other execution statistics for this simulator are given in Section 6.1.2.

6.1.1. Design of the Out-Of-Order Simulator

This simulator models a hypothetical out-of-order micro-architecture for the SPARC-V9 ISA. It models an 32 instruction out-of-order window, register renaming, a two-level non-blocking data cache, and a fixed number of integer and floating-point arithmetic units. Figure 6.1 shows the major components in this micro-architecture model, and table 6.1 lists various model parameters.

TABLE 6.1: Out-of-order processor model parameters.

Decode 4 instructions per cycle.
Buffer up to 32 instructions for out-of-order execution.
2 integer ALUs, 2 FPUs, and 1 load/store address adder.
2-bit/512-entry branch history table for branch prediction.
Speculatively execute instructions following multiple conditional branches.
Non-blocking L1 and L2 data caches, 8 MSHRs each.
16 KByte 2-way set associative write through L1 data cache.
1 MByte 2-way set associative write through L2 data cache.

Multiple instructions are fetched and decoded each cycle. While decoding, conditional branches are predicted and architectural registers are renamed, mapping them to physical registers. Each cycle, instructions are chosen from the out-of-order instruction window to execute on

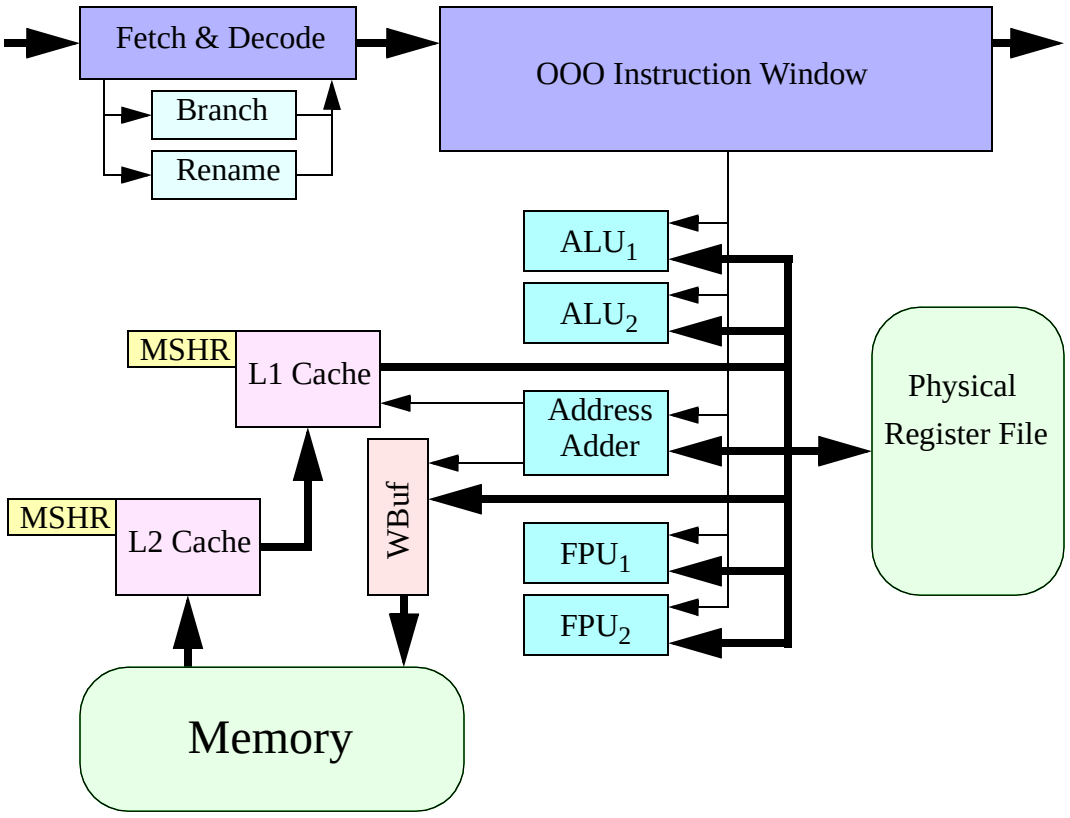

FIGURE 6.1: Out-Of-Order Processor Model.

available functional units: integer operations execute on ALUs, floating point operations execute on FPUs, and loads and stores use the Address Adder. After computing their memory address, loads get their value via a two-level non-blocking data cache, and stores write their value to a write buffer (WBuf) that manages the asynchronous transfer of data to memory. Instructions retire (and are removed from the window) in program order. An instruction is retired when it is finished executing and all older instructions in the window have also finished executing.

Part of the challenge in designing this simulator in Facile was to make effective use of fast-forwarding. While the data structures and algorithms used in this implementation are faithful to the micro-architecture model, they are often different from the obvious implementations that would be used in a simulator without memoization.

THE INSTRUCTION QUEUE. As with the simulator in Chapter III, the way to memoize an out-of-order simulator is to cache residual simulation indexed by the set of instructions in the current out-of-order window. Nearly all the data needed to specify the timing and order of execution of instructions in the pipeline is collected into a single data structure called the *instruction queue*. The instruction queue is not an actual component in the micro-architecture model, just a convenient data structure used in the simulator to centralize the run-time static part of simulated micro-architecture state. The instruction queue lists all instructions in the current instruction window, in program order, along with additional information about the current state of each instruction in the out-of-order pipeline. Every cycle of execution of the target micro-architecture, the simulator updates its instruction queue by 1) fetching new instructions, 2) scheduling instructions to execute on available simulated functional units, 3) evaluating instruction semantics for instructions that have spent sufficient time executing on a function unit, and 4) retiring instructions by popping them off the queue.

Figure 6.2 show a snapshot of the instruction queue taken between simulated machine cycles. In this example, the first nine instructions in the target instruction sequence are currently in the

Source Instructions

```
bne 0x124d4
sethi %hi(0x24000), %o0
ld [ %o0 + 0x238 ], %o0
cmp %o0, 0
bne 0x124d4
sethi %hi(0x13800), %l0
add %l0, 0x28c, %o0
call 0x23c64
mov %i5, %o2
sethi %hi(0x48000), %g1
sethi %hi(0xef763400), %g1
jmp %g1 + 0x308
sethi %hi(0x4b000), %g1
```

Instruction Queue

pc	npc	op	srcq	destq	ftype	ftime	other...
0x1247c	0x12480	BNE	...	...	ALU	1	...
0x12480	0x12484	NOP	...	...	DONE	0	...
0x12484	0x12488	LDUW	...	...	DONE	0	...
0x12488	0x1248c	SUBCC	...	...	DONE	0	...
0x1248c	0x12490	BNE	...	...	ALU	1	...
0x12490	0x12494	NOP	...	...	DONE	0	...
0x12494	0x12498	ADD	...	...	ALU	1	...
0x12498	0x1249c	NOP	...	...	DONE	0	...
0x1249c	0x23c64	OR	...	...	ALU	1	...

FIGURE 6.2: Sample Instruction Queue. The first nine instructions are in the current out-of-order instruction window, so have corresponding records in the instruction queue. The instruction queue lists each instruction's **pc**, **npc**, an internal operator identifier (**op**), where to find each source operand (**srcq**), where to write results (**destq**), which functional unit to execute on (**ftype**), how long to execute on the functional unit (**ftime**), and several other fields. Note that the **srcq** and **destq** fields in the instruction queue only record where to read and write data respectively. Actual data values are stored in other simulator data structures.

out-of-order execution window. In the next simulated cycle the next four target instructions will be fetched, the conditional branches at 0x1247c and 0x1248c will be evaluated (both were predicted correctly), and the oldest six instructions will be retired.

Many micro-architecture model parameters are implemented with simple counters. E.g., At most 4 instructions are fetched per cycle and at most 32 instructions are allowed in the instruction window. Counters keep track of the number of instructions fetched and the total number of instructions in the window. The simulator stops fetching instructions when either of these limits is

reached. A limited number of function units is implemented in the same way. The internal workings of each functional unit are not modeled by this simulator. Counters simply enforce a limit of 2 integer ALUs, 2 FPUs, and 1 Address Adder by counting the number of instructions using each kind of functional unit as the instruction queue is scanned from oldest to newest instructions. This gives priority to the oldest instructions in the instruction window. Note that the implementation of these limitations does not pass any extra data from one simulated cycle to the next, since each limitation is computed from scratch each cycle.

REGISTER RENAMING. Register renaming is more difficult to model in a memoization friendly way. If there are only 32 architectural registers and 64 physical registers, then there are 64^{32} possible register renamings. My original simulator—FastSim v.1—noted that the precise mapping of logical to physical registers is not needed to simulate the effect of register renaming on pipeline timing, so it simply recomputes data dependencies each cycle and uses a simple counter to limit the number of physical registers that can be allocated.

In the out-of-order simulator written in Facile, data values are maintained by the micro-architecture simulator, so storage locations are needed to store the register values. But pipeline timing still does not depend on the actual map from logical to physical registers. To implement register renaming in a memoization friendly way, the simulator stores result values of each instruction in a data structure called the destination queue. Like the instruction queue, the destination queue is just a convenient data structure for collecting data associated with instructions in the current out-of-order window.

Each element of the destination queue corresponds to one instruction in the instruction queue. It may contain multiple values, if an instruction assigns to more than one result register. Instead of maintaining a map from logical registers—the way they appear in a target instruction—to physical registers, the simulator maps each instruction operand to either the result of some previous instruction stored in the destination queue or to an architectural register file in the case that the register does not depend on any previous instruction in the queue. Newer instructions are not allowed to execute until the instructions they depend on have finished executing and have written their results into the destination queue. As in FastSim v.1, the number of physical registers can be limited with simple counters, while register values are managed by the destination queue and data dependencies are managed by the instruction queue. As instructions retire out of the instruction queue, their result values are popped from the destination queue and written into appropriate architectural register files.

The destination queue is labeled dynamic by BTA in the Facile compiler because it contains actual register values. But the data dependencies between instructions—stored in the instruction queue—are run-time static, since the data dependencies do not depend on register values. Register dependency information, describing which instruction operands depend on which previous instruction results, are computed as each instruction is fetched, and is stored in the instruction queue. The instruction queue is labeled run-time static by the Facile compiler. This means that the fast simulator magically knows where to read instruction operand values from—e.g., a particular index in the destination queue or from an architectural register file—when replaying cached simulation results. Note that data dependencies can be recomputed at any time by re-decoding the

instruction in the instruction window, so this information does not have to be passed from one call to `main` to the next. I.e., it is not a part of the memoization cache index.

NON-BLOCKING CACHE SIMULATION. The two-level non-blocking cache simulation is not memoized. The cache simulator is written in C and called using Facile's external function call interface. Note that the cache simulator in this implementation does not manipulate actual data values, it only simulates the time to access data under its simulated cache and memory model.

The interface between the memoized pipeline simulator and the non-memoized cache simulator is designed to minimize their interaction. A call to the cache simulator is made when a memory access instruction first accesses the cache. This call either returns that the access succeeded, or it returns a number of cycles for the pipeline simulator to wait before calling the cache simulator again for the same load or store.

For load instructions, the first call to the cache simulator either returns that the load hit in the L1 cache, missed in the L1 cache and will take N_1 cycles to access the L2 cache (usually 6 cycles), or missed in the L1 cache but failed to allocate a MSHR. A load that misses in the L1 cache is retried after N_1 cycles, and the cache simulator returns a hit if the address is in the L2 cache, or it returns a miss and another delay N_2 cycles, i.e., the time to access memory. Besides the variability of hitting or missing in each level of the cache, the length of a delay can vary, e.g., when loads are coalesced with other loads already in progress. Cache behavior also changes based on address dependencies between loads and stores currently submitted to the cache simulator.

Store instructions require special treatment when executing along a speculative branch path. For non-speculative stores, the cache simulator simply allocates a simulated write buffer and returns. The store will complete silently with no further calls from the pipeline simulator to the cache simulator. The write buffer is released automatically after the store has had time to make it out to simulated memory. Cache simulator calls for speculative stores also allocate a write buffer and return, but a subsequent call to the cache simulator is needed to either commit or rollback the store after the speculative branch is evaluated. If the branch was predicted correctly, then speculative stores following that branch are committed when the prediction is verified, and the store values can begin their trip out to simulated memory. Otherwise, speculative stores are rolled back and their write buffers are freed.

Actually loading and storing data from and to memory is handled by the pipeline simulator written in Facile. After the cache simulator returns that a load has hit in the L1 cache, L2 cache, or memory, or after a store has committed, the memory access is executed by the pipeline simulator. Hence, values are loaded and memory is modified out-of-order based on pipeline and cache behavior.

MEMOIZATION CACHE INDEX ENTRIES. In order to save space in the memoization cache, data in the instruction queue is compressed before it is stored into the `init` variable for the next call to `main`. At the start of a call to `main`, the instruction queue is reconstructed from the compressed data passed in `main`'s arguments.

Most data in the instruction queue can be reconstructed by re-decoding the corresponding target instructions. To begin this reconstruction, the simulator must know the `pc` and `npc` addresses of the oldest instruction in the queue, the length of the instruction queue (i.e., number of instructions), the target addresses of indirect jumps, and the direction taken by each conditional branch. This is enough information to identify exactly which instructions are in the instruction queue. By re-decoding target instructions, the simulator reconstructs most other instruction queue fields (e.g., op, srcq, destq). The ftype and ftime fields record the execution progress of each instruction, and cannot be reconstructed from the source instructions. A list of ftypes and ftimes is also passed as arguments to `main`.

The parameters to `main` in this out-of-order simulator are `pc`, `npc`, `cwp`, `cansave`, `canrestore`, `fuQ`, and `jmpQ`. Parameters `pc` and `npc` (4 bytes each) are the program counter and next program counter of the first instruction in the instruction queue. Parameters `cwp`, `cansave`, and `canrestore` (1 byte each) are not related to the instruction queue, but are included in `main`'s argument list to make the SPARC register window calculations run-time static. The function-unit queue—`fuQ`—contains the ftype and ftime fields for every instruction in the compressed instruction queue. For conditional branch instructions ftype also encodes the branch direction. The length of `fuQ` is the length of the instruction queue. Each entry in `fuQ` consumes 2 bytes in the memoization cache. The indirect jump queue—`jmpQ`—contains one entry for each indirect jump in the instruction queue, recording the indirect jump target addresses (4 bytes per address).

At the start of each call to `main`, the instruction queue is reconstructed from its compressed form, passed in `main`'s arguments. The simulator executes for a while using the reconstructed instruction queue. Then, just before `main` returns, it compresses the instruction queue again and stores this compressed representation into the `init` variable. In this way, all the data in the instruction queue is run-time static, while storing a minimum amount of data in each memoization index entry.

6.1.2. Out-Of-Order Simulator Performance

Measurements of the out-of-order simulator were made for the SPEC95 benchmarks, with and without memoization. To save time, all the benchmarks were run using their "test" input set, except for compress, which was run with its "train" input set. The host system for all these experiments was a Sun Microsystems Ultra Enterprise E5000 with 167MHz UltraSPARC processors and 2 GBytes of physical memory. When memoizing, a 256 MByte memoization cache was used. The cache is flushed when full, and new actions are memoized into the empty cache as needed.

With memoization the simulator is 11.6 times faster on average than without memoization. Figure 6.3 shows the how memoization accelerates simulator performance for each of the SPEC95 benchmarks[1]. Without memoization simulation slowdown ranges from 8,600 (vortex) up to 22,600 (turb3d). The slowdown tends to be worse for floating-point benchmarks (the 10 bench-

1. Un-memoized execution of applu failed after executing for more than two weeks, so SlowSim performance for applu is not shown. Since applu ran with memoization, this failure was probably due to some event on the host beyond the control of the simulator.

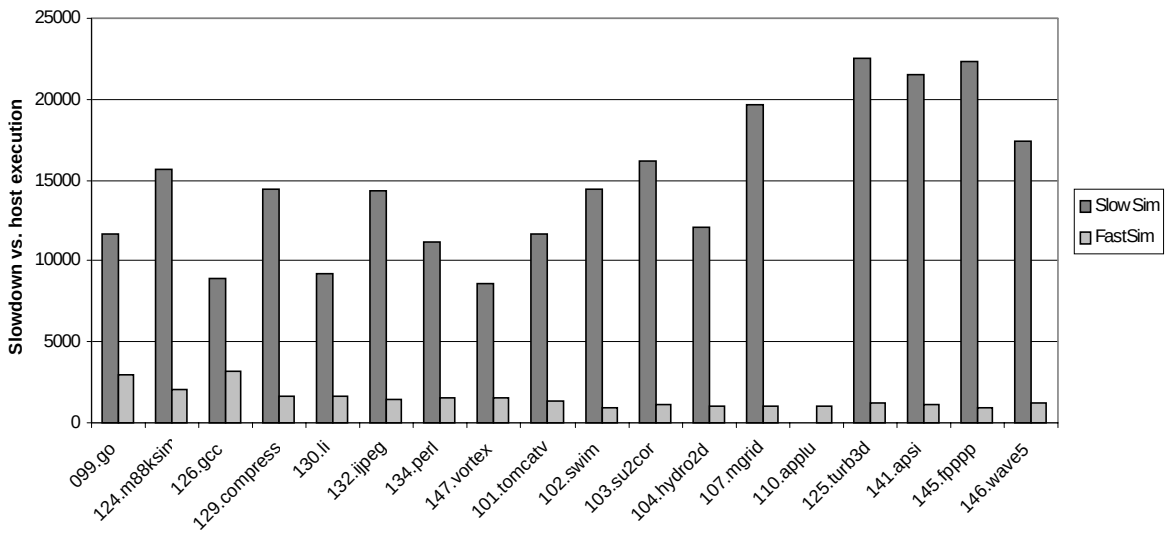

FIGURE 6.3: Out-of-order simulator performance. SlowSim is the out-of-order simulator without memoization. FastSim is the simulator with memoization.

marks on the right) than for integer benchmarks (the 8 benchmarks on the left), because floating-point benchmarks tend to have more instructions in the out-of-order window on average, and this simulator implementation loops over the out-of-order window once per simulated cycle.

With memoization, simulator slowdown ranges from 940 (for fpppp) up to 3,150 (for gcc). Among other things, memoization removes the loop that scans through the instruction window each cycle, so floating-point benchmarks no longer perform worse than integer benchmarks. On the contrary, floating-point benchmarks tend to be more regular and are more likely to repeat previously memoized simulation. They out perform the integer benchmarks on average. The two worst slowdowns belong to go (3,020 times slowdown) and gcc (3,150 times slowdown). Both these benchmarks memoized significantly more data than fit in the 256 MByte memoization cache used for these experiments. They were forced to frequently flush the cache and re-memoize

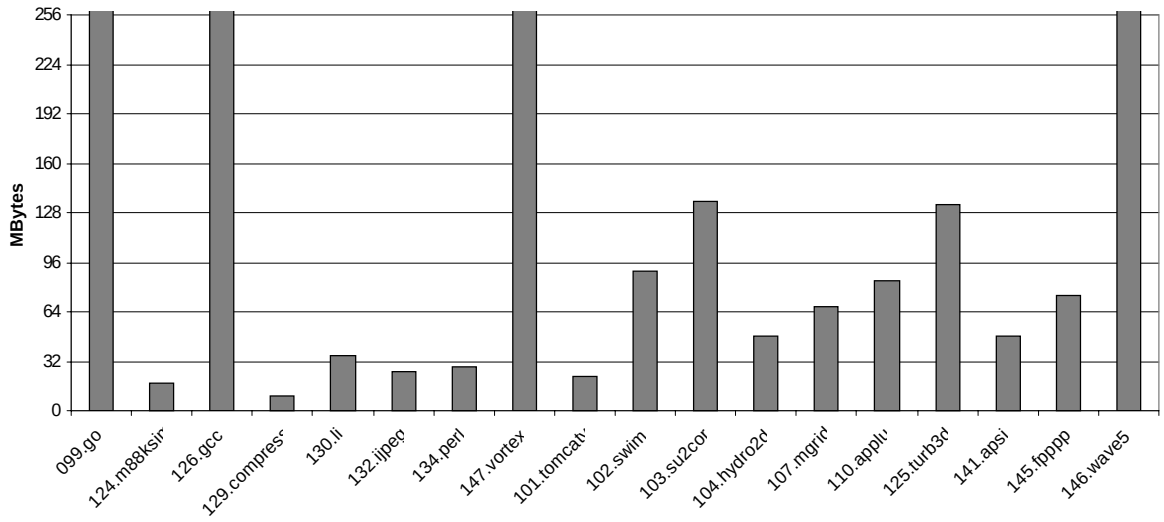

FIGURE 6.4: Quantity of data memoized for out-of-order simulation. The memoization cache was limited to 256 MBytes. The cache is flushed when full and new actions are memoized as needed.

data needed for fast simulation. Despite this, memoization still improved their rate of simulation by a factor of 3.9 and 2.8 for go and gcc respectively.

Figure 6.4 shows the amount of data memoized for each benchmark. The memoization cache size for these experiments was 256 MBytes. Benchmarks that overflow this limit flush the memoization cache and start over memoizing new simulator actions. Go, gcc, vortex, and wave5 flushed the memoization cache 271, 29, 4, and 6 times respectively. The amount of data memoized for a benchmark that overflows the cache increases sharply, since new data must be memoized to replace the discarded data.

Benchmark	$\%W_{slow}$	$\%W_{fast}$	$\%W_{miss}$
099.go	2.52%	95.40%	2.07%
124.m88ksim	0.02%	99.97%	0.01%
126.gcc	3.45%	94.92%	1.63%
129.compress	0.20%	99.73%	0.07%
130.li	0.02%	99.97%	0.01%
132.ijpeg	0.04%	99.94%	0.02%
134.perl	1.91%	97.59%	0.51%
147.vortex	0.06%	99.89%	0.05%
101.tomcatv	0.01%	99.99%	0.00%
102.swim	0.23%	99.29%	0.48%
103.su2cor	0.04%	99.91%	0.04%
104.hydro2d	0.04%	99.94%	0.02%
107.mgrid	0.07%	99.89%	0.03%
110.applu	0.00%	99.99%	0.00%
125.turb3d	0.00%	99.99%	0.00%
141.apsi	0.01%	99.98%	0.00%
145.fpppp	0.11%	99.83%	0.05%
146.wave5	0.02%	99.92%	0.07%

TABLE 6.2: Out-of-order simulation work breakdown. This table lists the percentage of calls to `main` that are handled in slow, fast, and miss recovery mode. $\%W_{slow}$ is the percentage of calls to `main` in slow simulation. $\%W_{fast}$ is the percentage of calls to `main` replayed by the fast simulator without a memoization miss. $\%W_{miss}$ is the percentage of calls that result in a memoization miss in the fast simulator.

Compared to the hand coded simulator discussed in Chapter III—FastSim v.1—the simulator written in Facile generated from 7.3 (ijpeg) times less to 14.1 (turb3d) times more memoized data (not counting those benchmarks that overflowed the cache). On average, the Facile simulator allocated 4.9 times more data in the memoization cache than FastSim v.1 on benchmarks that fit in the 256 MByte memoization cache.

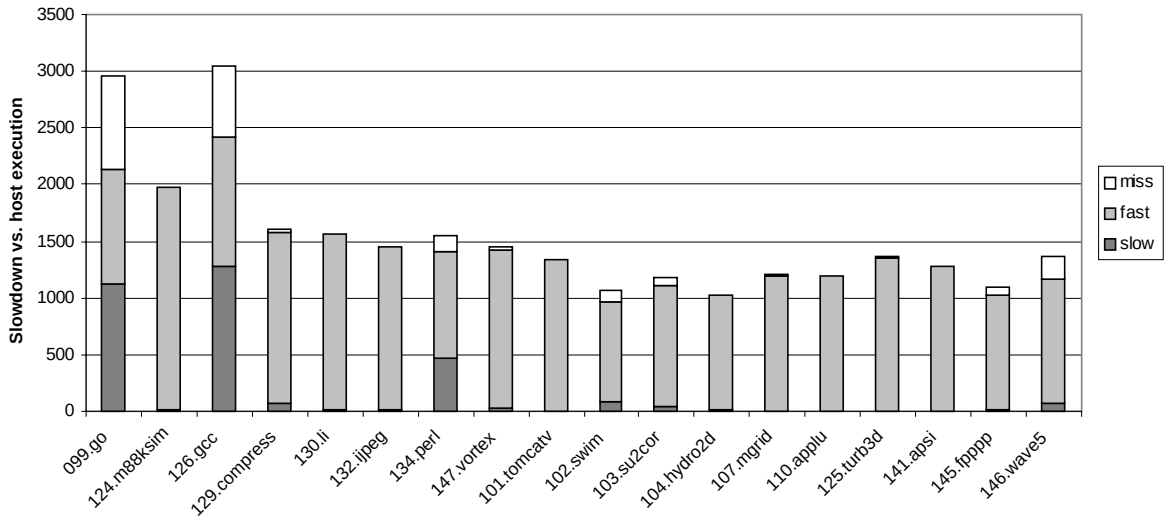

FIGURE 6.5: Breakdown of simulator execution time. This chart shows the simulation slowdown of the memoized out-of-order simulator with respect to executing benchmarks directly on the host. Each bar is divided into the fraction of execution time spent in each of the three simulation modes: slow, fast, and miss recovery.

Table 6.2 lists the percentage of calls to `main` that were executed by the slow simulation, fast simulator, or by a combination of both slow and fast simulator to handle memoization misses. The vast majority of calls to `main` are fast-forwarded. The gcc benchmark executes the largest percentage of work in the slow simulator (3.45%), but the go benchmark has the largest percentage of memoization misses (2.07%). On average, the benchmarks performed 0.49%, 99.23%, and 0.28% of their work in slow, fast, and miss recovery mode respectively.

Figure 6.5 breaks down the execution time for simulating each benchmark, showing the contribution from each of the slow, fast, and miss recovery modes. Even though very little work is performed in slow and miss recovery modes, these modes are sometimes responsible for a signifi-

cant fraction of the execution time. This is because slow simulation is 51 times slower than fast simulation, and memoization miss recovery is 66 times slower than fast simulation. Compared to the non-memoized version of this out-of-order simulator, the fast simulator is 12.3 times faster, the slow simulator is 4.2 times slower, and miss recovery is 5.1 times slower on average.

6.2. Performance Model for Fast-Forwarding Simulators

Memoization makes the trade-off of increasing memory consumption to accelerate program execution. The hope is that by caching and reusing results from earlier in a program's execution, the time required to complete the entire execution will be decreased. The effectiveness of this optimization depends on several factors, including: 1) the fraction of a program's work that can be replayed using cached data, 2) the cost of using cached data relative to the rate of execution without memoization, 3) the cost of generating cached data, and 4) the amount of data generated into the memoization cache.

FastSim's fast-forwarding optimization is similar to traditional memoization, but it caches actions that control a residual/dynamic simulator instead of just caching function return values. With traditional memoization, the cost of looking up and reusing a previously cached result is constant. But with fast-forwarding, the cost of replaying cached actions is often much greater than looking up a single value. This cost changes from one simulator design to another and even varies within a single simulator's execution, depending on the fraction of executed code that can be skipped over when replayed.

Fast-forwarding also has a variable cost for handling memoization misses. The cost of a memoization miss in traditional memoization is constant. Memoization simply fails to find a previous instance of a function call in the cache, so it calls the function and caches its new result for later use. Fast-forwarding uses a more costly memoization miss recovery mechanism, with a variable cost depending on the amount of simulation that must be rolled back and recomputed by the slow simulator.

6.2.1. The Fast-Forwarding Performance Equation

A fast-forwarding simulator (S_{ff}) produces exactly the same results as an un-optimized version (S_0) of the same simulator, only the running time and memory consumption are different. Let T_0 and T_{ff} be the execution times of S_0 and S_{ff} respectively, when applied to the same target executable with identical starting conditions (e.g., identical command line arguments and environment variable values). Fast-forwarding is an effective optimization of a simulator if and only if T_{ff} is less than T_0.

Both simulators—S_0 and S_{ff}—perform the same simulation work, e.g., they simulate the same sequences of target instructions and compute the same target processor statistics. One way to measure simulation work performed by a Facile simulator, is to count calls to the simulator's `main` function. Note that in a fast-forwarding simulator the `main` function is not actually called during fast simulation, but logically each index entry encountered by the fast simulator corresponds to a call to `main`.

Let W be the work performed by a simulator, measured in calls to `main` (or index entries encountered) during the simulation of a target executable. The execution time of an un-optimized simulator S_0 can be described as $T_0 = Wt_0$, where t_0 is the average time taken to execute one call to `main`. Describing the execution time of a fast-forwarding simulator S_{ff} is more complicated. Some work in S_{ff} is performed by a slow version of the simulator, more work is performed by a fast simulator version, and the remaining work is performed by a combination of the fast and slow simulators working together to handle a memoization cache miss.

- Let W_{slow} = the number of times the slow simulator's `main` function is executed, when not recovering from a memoization miss.

- Let W_{fast} = the number of index entries replayed by the fast simulator, where a memoization miss does not occur before the next index entry is encountered.

- Let W_{miss} = the number of memoization cache misses encountered by the fast simulator. Each miss results in activation of FastSim's memoization miss recovery mechanism.

For a given target executable and target input data, the number of calls to `main` in S_0 is the same as the sum of these three components in S_{ff} (i.e., $W = W_{slow} + W_{fast} + W_{miss}$).

The performance of a fast-forwarding simulator can be described as follows:

$$T_{ff} = W_{slow}t_{slow} + W_{fast}t_{fast} + W_{miss}t_{miss}$$

Where t_{slow}, t_{fast}, and t_{miss} represent the average time to execute `main` in the slow simulator, fast simulator, and during miss recovery respectively.

In a simulator that makes effective use of the fast-forwarding optimization, the fast simulator is faster than the un-optimized simulator (i.e., $t_{fast} < t_0$) because run-time static code is skipped. But the slow simulator is slower (i.e., $t_{slow} > t_0$) because of extra overhead from memoizing result data. A memoization miss is even more time consuming to simulate than slow simulation, because a portion of the call to `main` is executed twice—once in the fast simulator, then again in the slow simulator—and extra time is spent in FastSim's run-time library managing the transition back to slow simulation.

The amount of work performed in each mode of simulation—W_{slow}, W_{fast}, W_{miss}—depends on the likelihood that memoized data can be reused by the simulator. Simulator designers normally have some intuition about how often memoized data will be reused. They know that some design changes will increase (or decrease) the likelihood of encountering repeated work, or change the rate of execution in one or more of the three modes of execution.

The following definitions are used to describe the probability that work can be replayed by the fast simulator and how much work is replayed before returning to slow simulation.

- Let N_{fast} = the number of times S_{ff} transitions from slow simulation to fast simulation. This happens when `main`'s argument values match an existing index entry. Note that

$N_{fast} = W_{miss}$, because every transition to fast simulation eventually leads to a transition back to slow simulation following a memoization miss.

- Let θ = the probability that arguments to `main` will match an existing index entry during slow simulation. θW_{slow} calls to `main` in the slow simulator result in a transition to the fast simulator, so $\theta W_{slow} = N_{fast} = W_{miss}$.

- Let α = the probability that the value in Facile's `init` variable matches an existing index entry during fast simulation. This index check is performed whenever the fast simulator reaches the end of an action sequence associated with the previous index entry.

- Let β = the probability that a dynamic result value matches one of the values already listed in the memoization cache. These tests transform dynamic result values into run-time static values by comparing dynamic results computed by the fast simulator against a set of values previously witnessed by the slow simulator. Several dynamic results can be tested when replaying the action sequence associated with a single index entry.

- Let r_f = the average number of dynamic result values that are tested when replaying the action sequence associated with an index entry. This is the number of dynamic results tested per index entry replayed by the fast simulator. A similar value describes result value tests per index entry generated by the slow simulator (r_s), and is used in the equations describing memoization cache size discussed in section 6.2.2.

- Let φ = the probability that a memoization miss occurs during fast simulation (i.e., $W_{miss} = \varphi W_{fast}$). A memoization miss occurs if no matching index entry is found anywhere in the memoization cache or if a dynamic result value is not found in the current action sequence. Hence $\varphi = 1 - \alpha\beta^{r_f}$.

The following relationships are used to derive a new fast-forwarding performance equation based on the probabilities of transitioning to and from fast simulation, and the execution speed of each of the three modes of simulation:

1. $W_{miss} = \theta W_{slow}$

2. $W_{miss} = \varphi W_{fast}$, where $\varphi = 1 - \alpha\beta^{r_f}$

3. $W = W_{slow} + W_{fast} + W_{miss}$

   Hence $W_{slow} = \frac{\varphi}{\varphi + \theta + \varphi\theta} W$, $W_{fast} = \frac{\theta}{\varphi + \theta + \varphi\theta} W$, and $W_{miss} = \frac{\varphi\theta}{\varphi + \theta + \varphi\theta} W$.

4. $T_{ff} = W_{slow} t_{slow} + W_{fast} t_{fast} + W_{miss} t_{miss}$

By substituting values for W_{slow}, W_{fast} and W_{miss} in equation 4 we get a description of fast-forwarding performance with respect to the performance of the simulator's three modes of opera-

tion and the probabilities of transitioning to and from fast simulation. This is the *fast-forwarding performance equation*:

$$T_{ff} = \frac{W}{\varphi + \theta + \varphi\theta}[\varphi t_{slow} + \theta t_{fast} + \varphi\theta t_{miss}]$$, where $\varphi = 1 - \alpha\beta^{r_f}$, θ is the hit rate for slow simulation, α is the hit rate for fast simulation, and β is the hit rate for dynamic values.

6.2.1.1. SLOW SIMULATOR PERFORMANCE

The first mode of fast-forwarding simulator operation is slow simulation. The performance of the slow simulator is described by t_{slow}—the average time to execute one call to `main` during slow simulation. The slow simulator (S_{slow}) performs the same operations as the un-optimized simulator (S_0) skipping nothing. S_{slow} also generates data into the memoization cache and looks for previously memoized data so it can hand off execution to the fast simulator. The average time taken to execute each call to the slow simulator's `main` function can be broken down into the following components:

$$t_{slow} = t_0 + t_{lookup} + (a_s t_{gen\text{-}action} + d_s t_{gen\text{-}data} + r_s t_{gen\text{-}result}) + t_{extra}$$

CORE SIMULATION. As defined earlier, t_0 is the average time taken by a call to `main` when fast-forwarding is not used. This is the same time taken to execute a call to `main` in the un-optimized simulator S_0. It includes the time to execute all code (static, run-time static, and dynamic) that is in or called by `main`, but does not include the time to execute any extra code that supports memoization.

INDEX LOOKUP. t_{lookup} represents the average time to lookup `main`'s arguments in the memoization cache. Note that this lookup fails, since a successful lookup would result in calling the fast simulator instead of the slow simulator.

The memoization cache index is accessed via a hash table. A hash value for `main`'s arguments—stored in Facile's `init` variable—is generated, looked up in the hash table, and all index entries in that hash bucket are compared to the current argument values. Hence t_{lookup} depends on the amount of data in `init` and the number of index entries in a hash bucket. Increasing the amount of data in `init` increases to time required to compute the hash function and compare the current value against values in existing index entries. Increasing the size of the hash table[1] reduces the average number of index entries in each hash bucket.

CACHING DATA. The time to write an action sequence into the memoization cache is primarily related to the amount of code in S_{slow} that was labeled dynamic by Facile's BTA and executes in a typical call to `main`. More specifically, time to cache a memoized action sequence is $a_s t_{gen\text{-}action} + d_s t_{gen\text{-}data} + r_s t_{gen\text{-}result}$. a_s and r_s represent the average number of actions and dynamic result tests respectively that occur in an action sequence generated by the slow simulator. $t_{gen\text{-}action}$ and $t_{gen\text{-}result}$ are the average times to allocate a single action (with associated run-time static data) and single dynamic result respectively. d_s is the average number of bytes of run-time

1. A hash table size can be specified on the command line of any memoizing simulator generated by FastSim v.2. The default hash table size is $1/256^{th}$ of the size of the memoization cache. The size of the memoization cache can also be specified on the simulator command line.

static data associated with all the actions in an action sequence, and $t_{gen\text{-}data}$ is the average time to write a byte of run-time static data into the cache.

The number of actions (a_s) depends on the complexity of a simulator's dynamic control flow graph (dCFG). Simulator designs that have more complex dynamic control flow graphs typically generate more actions per action sequence. Careful programming of a Facile simulator can sometimes reduce the number of actions stored in the memoization cache by simplifying the dCFG without altering the proportions of dynamic and run-time static code.

The total amount of run-time static data in an action sequence (d_s) depends on the amount of dynamic simulator code that uses run-time static values. Simulator designs that use fewer run-time static values in dynamic expressions generate less run-time static data into the memoization cache. Run-time static data records are associated with individual actions, but increasing the number of actions does not necessarily increase the amount of run-time static data. If the same amount of run-time static data is used in dynamic expressions, then increasing a_s simply decreases the average amount of run-time static data associated with each action.

The number of dynamic result values in an action sequence (r_s) depends on the number of dynamic conditional statements (e.g., if or switch statements) or explicit uses of Facile's `?static` attribute that are encountered by a typical call to `main` in the slow simulator. Simulator designers can influence r_s by changing the proportion of dynamic code in a simulator or changing the number of instructions simulated per call to `main`.

EXTRA COSTS. t_{extra} represents other costs incurred by the slow simulator. These extra costs result mostly from support added to S_{slow} for memoization miss recovery. Extra statements copy data between run-time static and dynamic variable versions, and extra if statements disable dynamic code during miss recovery. Because of all this extra code and duplicate variables, S_{slow} cannot be optimized as highly by the C compiler as S_0. The extra time needed to execute this less efficiently compiled code is also included in t_{extra}.

Extra statements—to copy run-time static variables to dynamic version of the same variables—occur along a control flow path wherever run-time static values are merged with dynamic values from a different control flow path. For example, the then-clause of an if statement may set a variable x with a run-time static value, while the else-clause leaves x with a dynamic value. When this if statement ends, and the two control flow paths merge back together, the run-time static value in x at the end of the then-clause becomes dynamic and must be copied into the dynamic version of x. This transition from run-time static to dynamic does not correspond to any statement in the Facile source code, so an extra statement is generated.

The number of extra statements that copy from run-time static to dynamic variables can be reduced by designing a simulator so that variables that store dynamic values always store dynamic values. Similarly, variables that store run-time static values should always be used to store run-time static values. It is only when some control flow path changes a variable from run-time static to dynamic that these extra statements are inserted into the code.

A cost not related to miss recovery is Facile source statements that are rewritten as two or more C statements, because memoization support code is needed in the middle of the source statement. For example, if Facile's `?static` attribute is used in a sub expression of a larger expression, the larger expression is split into multiple statements. One statement computes the dynamic value that will be made static and stores that value in a temporary variable. Then a run-time library call is made to write the dynamic value into the memoization cache. The remainder of the original expression is written as another C statement. Multiple C statements, with extra temporary variables to store intermediate results, may be generated to implement the same order of evaluation as in the source simulator code. These multiple statements and extra temporary variables can not be compiled as efficiently as a single C statement.

6.2.1.2. FAST SIMULATOR PERFORMANCE

Fast simulator performance is represented by t_{fast}, the average time to replay a single index entry and its associated action sequence. t_{fast} can be broken down into the time to check an index entry and the time to replay all the actions and dynamic results associated with that index.

$$t_{fast} = t_{index} + a_f t_{action} + r_f t_{result}$$

INDEX VERIFICATION. t_{index} represents the average time to find an index entry for the next sequence of actions that should be replayed. Verifying an index entry in the fast simulator serves the same purpose as looking-up `main`'s argument values in the slow simulator. Part of the work performed by a replayed action sequence is to set up Facile's `init` variable with a value for the

next index/call to `main`. This value is checked against the next expected index entries in the memoization cache to verify that relevant actions exist, and fast simulation can continue.

Logically, the fast simulator looks for an index entry to match `main`'s current argument values just like the slow simulator does. But index lookup in the fast simulator is optimized by listing all the index entries that are known to follow the previous action sequence at the end of that previous action sequence. When the fast simulator finishes executing an action sequence, it first searches that sequence's list of known successors to find an index entry that matches the current value of `init`. Often a successor list has only one element and that element matches the value in `init`, so this optimization saves time over looking for an index entry in the hash table. If the current value of `init` does not match any index entries in the successor list, then the fast simulator looks for an index entry using the hash table to search among all index entries in the memoization cache, just like the slow simulator. If an index entry is not found anywhere in the memoization cache, then it is a memoization miss and control is passed back to the slow simulator via FastSim's miss recovery mechanism.

A simulator designer can reduce the length of index successor lists and reduce the number of hash table lookups performed by the fast simulator by storing less dynamic data into the `init` variable. A simulator design that only stores run-time static data into `init` will generate successor lists with exactly one element that always matches the index value searched for by the fast simulator. In general, multiple successor index entries differ only in the values of the dynamic parts of `init`. Note that in FastSim's current implementation the entire value of `init` is com-

pared against each possible index entry, even if some of the data is run-time static and guaranteed to match. A better implementation would only test the dynamic parts of `init` when comparing it to the index entries in a successor list.

REPLAYING ACTIONS. Action sequences associated with an index entry encode dynamic computation that is not skipped by fast forwarding. Each action number identifies a dynamic basic block implemented in the fast simulator's switch statement, and data following an action number in the cache provides run-time static data values used by dynamic basic block code. Hence, the time needed to replay actions associated with an index entry depends on the average number of actions in an action sequence encountered by the fast simulator (a_f) and the average time to execute a single action (t_{action}).

Replaying a single action involves reading the action number from the memoization cache, jumping to the dynamic basic block code associated with that action, then executing the dynamic code. The total amount of dynamic code associated with all the actions in an action sequence depends only on the fraction of a simulator's execution that is dynamic. A simulator design with a more complex dCFG but the same fraction of dynamic code will replay more actions but has less dynamic code per action. Even if the total amount of dynamic code stays the same, changes in the number of actions replayed will effect performance, because of the overhead of reading an action number and jumping to its associated dynamic code.

REPLAYING DYNAMIC RESULTS. The fast simulator converts dynamic values into run-time static values by comparing a dynamic result value to a list of previously seen result values already stored in the memoization cache. Dynamic results are made run-time static when evaluating any dynamic conditional statement—because all control flow in a fast-forwarding simulator is made run-time static—and when dynamic data is explicitly made run-time static using the `?static` attribute in Facile. Dynamic result lists are intermixed with actions numbers and run-time static data records in an action sequence. The time spent converting dynamic values depends on the average number of conversions (r_f) associated with a replayed index entry and the average time to find a previously computed value in a result list (t_{result}).

The time to convert a dynamic result value depends on the length of the result list, which depends on the number of different dynamic values that were previously encountered at this point in the action sequence. When an action sequence is first generated by the slow simulator, each dynamic result list in that action sequence contains exactly one result value. New entries are added to a result list when the fast simulator encounters a dynamic value that is not yet in the list. When this happens a new list entry is appended to the end of the list, and the slow simulator is restarted via the memoization miss recovery mechanism. The slow simulator associates new actions to this new result list entry, so the fast simulator can handle this result value in the future.

6.2.1.3. MEMOIZATION MISS RECOVERY

The third and final mode of execution in a fast-forwarding simulator is memoization miss recovery. t_{miss} represents time spent processing a memoization miss and returning to slow simula-

tion. A memoization miss occurs when data needed to continue replaying memoized simulation cannot be found in the memoization cache. This happens when either no index entry is found that matches data in the `init` variable or a dynamic result is tested whose value is not listed in the current action sequence.

FastSim's recovery mechanism performs several steps to return from fast simulation to slow simulation: 1) It searches backward through the replayed action sequences to find the most recently checked index entry. During this search it also pushes action numbers and replayed dynamic result values onto a stack—called the recovery stack—for later stages of recovery. 2) The slow simulator's `main` function is started in recovery mode, with argument values taken from the last index value. In recovery mode, the slow simulator does not execute any dynamic code. Instead of putting new data in the memoization cache, it simply verifies action numbers and reads dynamic result values from the recovery stack. 3) When all the actions and dynamic results in the recovery stack have been replayed by the slow simulator, the slow simulator leaves recovery mode and returns to normal execution. The call to `main` that started in recovery mode finishes executing in normal slow simulation mode.

$$t_{miss} = f t_{fast} + f t_{rollback} + f t_{recover} + (1 - f) t_{slow}$$

PARTIAL FAST SIMULATION. Before a memoization miss, part of the action sequence associated with the last checked index value is replayed by the fast simulator. Time spent in this partial replay is represented by $f t_{fast}$ where f is the average fraction of an action sequence that is replayed before a miss and t_{fast} is the average time to replay the entire action sequence when no miss

occurs. Note that, if memoization misses only occur when checking index values, then no actions are replayed and $f = 0$. f is greater than 0 when memoization misses result from missed dynamic result values.

ACTION SEQUENCE ROLLBACK. $ft_{rollback}$ represents the time to roll back the fraction of an action sequence that was executed by the fast simulator before a memoization miss. Action numbers and dynamic result values that were used in fast simulation are pushed onto the recovery stack for use in slow simulation while in recovery mode. If misses only occur when checking index values, then $f = 0$, no rollback is needed, and no actions are pushed onto the recovery stack.

RECOVERY MODE SIMULATION. $t_{recover}$ represents the average time to execute the slow simulator's `main` function in recovery mode. $ft_{recover}$ represents the average time to execute the fraction of the slow simulator's `main` function in recovery mode that occurs before the memoization miss being recovered. When in recovery mode the slow simulator does not execute any dynamic code. All dynamic statements in the simulator are guarded by C if statements that only allow dynamic code to execute when not in recovery mode. The slow simulator also skips extra statements that were inserted by the Facile compiler to move data from run-time static variables into the corresponding dynamic variables.

In recovery mode, the slow simulator still calls run-time library functions to put action numbers, run-time static data, and dynamic result values into the memoization cache. But these functions do not actually write data into the cache. For action numbers, the run-time library simply

verifies the numbers generated by the slow simulator against action numbers in the recovery stack.[1] For run-time static data, the run-time library just throws away the generated data. For dynamic result values the run-time library returns the value used by the fast simulator, since these dynamic values cannot be computed reliably by the slow simulator until recovery is finished.

PARTIAL SLOW SIMULATION. After the slow simulator, executing in recovery mode, reaches the point where the memoization miss occurred, it returns to normal slow simulation. $(1 - f)t_{slow}$ represents the time to execute the slow simulator's `main` function following the point where the memoization miss occurred. As defined earlier, t_{slow} is the time to execute `main`, when the slow simulator is in normal execution mode.

6.2.2. Memoization Cache Size

The amount of data put into the memoization cache during simulation of a given target executable can vary greatly among different simulator designs. The amount of memoized data depends on the number of index entries that are written into the cache and the amount of data associated with each index entry. Let S stand for the total amount of memoization data allocated by a fast-forwarding simulator. $S = N_{index}s_1$, where N_{index} is the number of index entries allocated and s_1 is the average size of an index entry and associated action sequences up to the next index entry. Note that the number of index entries allocated equals the number of calls to `main` in the slow simulator, so $N_{index} = W_{slow}$ and $S = W_{slow}s_1$.

1. This error checking was useful for debugging FastSim and does not add much to simulator execution time. It was left in as a sanity check, since it verifies that the two simulator versions—fast and slow—are communicating correctly.

In the worst case, the amount of data associated with a cached index entry is proportional to the size of an index entry plus the size of the tree of possible action sequences following that index. $s_1 \sim x + (4a_s + d_s + 8r_s)l^{r_s}$, where x is the average size of an index entry. a_s, d_s, and r_s are the average number of actions (4 bytes each), bytes of run-time static data, and dynamic results[1] respectively in an action sequence generated by the slow simulator. *l* is the average length of a dynamic result list. Note that a tree of action sequences is associated with each index entry, because each alternate dynamic result value is followed by a different sequence of actions. Hence, the worst case size of data associated with an index entry increases exponentially as r_s increases in alternate simulator implementations.

A worst case estimate of s_1 is useful for designing simulators that memoize less data. The biggest effect on the size of s_1 comes from reducing the number of dynamic results tested and the number of values in each dynamic result list. Note that moving dynamic result tests into the index check can reduce s_1, but increases the number of index entries, so it may not reduce the total amount of memoized data. Changes in the number of actions or bytes of run-time static data is the next biggest effect in determining s_1, and these changes can sometimes be accomplished without changing the number of index entries allocated (see 6.3.5, “Removing Actions Via Function Inlining”). Finally, the amount of data passed to `main` and stored in each index entry contributes to the size of s_1.

1. Each dynamic result consumes at least 8 bytes. Result values less than or equal to 8 bits wide are included in the 8 byte header. Wider values consume 8 bytes plus space to store the value.

In an actual simulator execution $s_1 = (4a_s + d_s + 8r_s)\left(1 + (1-f)\frac{W_{miss}}{W_{slow}}\right) + x$. This is the size of the index entry itself (x) plus the average size of its associated tree of action sequences. An action sequence without any branches has size $4a_s + d_s + 8r_s$. An additional branch is added to the action sequences associated with one index entry in the cache each time there is a memoization miss. But only the actions, run-time static data, and dynamic results following the miss are added. Hence the average size of a tree of action sequences is $(4a_s + d_s + 8r_s)\left(1 + (1-f)\frac{W_{miss}}{W_{slow}}\right)$.

Therefor $S = (4a_s + d_s + 8r_s)(W_{slow} + (1-f)W_{miss}) + xW_{slow}$, or in terms of φ and θ:

$$S = \frac{\varphi W}{\varphi + \theta + \varphi\theta}[(4a_s + d_s + 8r_s)(1 + (1-f)\theta) + x]$$

6.3. Designing an Efficient Memoizing Simulator

The structure of a simulator written in Facile controls how the fast-forwarding optimization is applied by the Facile compiler. Choices made in the design of a memoizing simulator influence different components of fast-forwarding performance in different ways. For example, passing more data in arguments to `main` may increase the amount of run-time static code skipped by fast simulation, i.e., decreasing t_{fast}. But at the same time this change may also decrease the fraction of calls to `main` that are executed by the fast simulator, i.e., decreasing θ (the probability of transitioning to fast simulation) or increasing φ (the probability of transitioning back to slow simulation).

This section explores several implementation options and their effect on fast-forwarding performance in a simple functional simulator of the SPARC-V9 ISA. The simulator designs used in these experiments are all simple variations on the simulator given in Appendix B. These simulators do not model any micro-architecture details, and only simulate the functional behavior of SPARC-V9 user-level instructions. Below is the list of design variations used in these experiments:

- ARGS-TO-MAIN changes the proportion of simulator code that is run-time static (i.e., can be skipped over) versus dynamic code that cannot be skipped. This proportion is changed by passing more data as arguments to `main`, which makes the argument data and all computations that depend on it run-time static.

- COMBINING changes the amount of run-time static data used in dynamic computations. In the SPARC simulators in these experiments, the SPARC's integer register windows are represented by a two dimensional array. When registers are read and written, both the window number and the register number within that window are run-time static (in some versions of the simulator). These two values can be stored separately in the memoization cache, or they can be combined into a single run-time static value to save space.

- INLINING changes the number of actions per index entry. The semantic code for many SPARC instructions uses a function (`get_src2`) to get either an immediate value or a register value for the instruction's second operand. Function inlining removes one action for each time

`get_src2` is called, compared to when `get_src2` is called as a separate function. Unfortunately, inlining also increases code size, possibly decreasing simulator performance.

- INSTS. PER MAIN changes the number of instructions simulated by each call to `main`. The simulator in Appendix B simulates exactly one target instruction per call to `main`. To simulate more instructions per call to `main`, the simulator is altered to count the number of taken branches, calls, or indirect jumps. By placing a loop around the call to `?exec`—the call that decodes and simulates a target instruction—`main` runs until a number of taken branches have been simulated.

Measurements of all simulator versions were taken for the SPEC95 benchmarks, with and without memoization. To save time, all the benchmarks were run using their “test” input set, except for compress, which was run with its “train” input set. The host system for all these experiments was a Sun Microsystems Ultra Enterprise E5000 with 167MHz UltraSPARC processors and 2 GBytes of physical memory. When memoizing, a 256 MByte memoization cache was used. The cache is flushed when full, and new actions are memoized into the empty cache as needed.

Note that fast-forwarding is not an effective optimization for the simple simulators used in these experiments. In the most efficient versions, the simulator runs just as fast with memoization as without it. Most versions of the simple simulator ran slower, when using memoization. The problem is that not enough code can be skipped over to justify the extra cost of fast-forwarding.

Despite this problem, these simulators are useful for illustrating design decisions that produce more efficient memoizing simulators.

6.3.1. Base Simulator Version

Appendix B contains source code for the simulator used in these experiments in its simplest form. Call this the base simulator. The base simulator executes one instruction per call to `main`, has no function inlining, and accesses simulator registers with separate register window and register number values (i.e. no combining). By passing the simulated `pc` and `npc` values as arguments to `main`, the code to decode simulated instructions is run-time static in the base simulator. Most code to simulate the semantics of each instruction is dynamic.

TABLE 6.3: Base simulator configuration.

Simulator	Args-to-main	Combining	Inlining	Insts./main
base	pc, npc	no	no	1

Figure 6.6 graphs the execution slowdown of the base simulator (with and without memoization) versus executing benchmarks directly on host hardware. Without memoization, the base simulator suffers an average 112 times slowdown. With memoization, the simulator is 2.7 times slower than simulation without memoization.

The memoized base simulator spends nearly all its time in fast simulation: All but one benchmark spends ≤ 0.02% of time in slow simulation. The perl benchmark spends 0.21% of its execu-

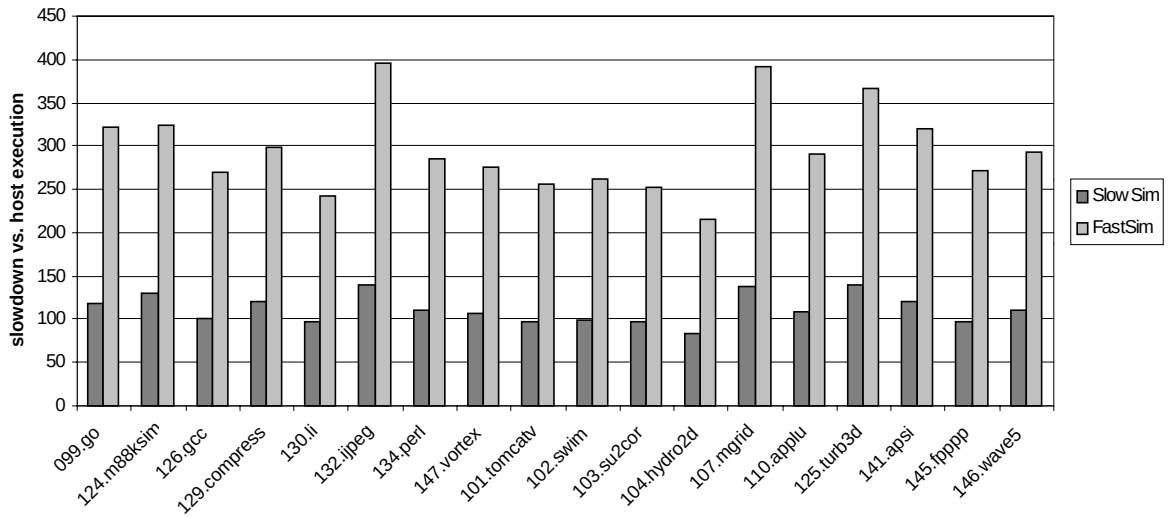

FIGURE 6.6: Base simulator performance. Base simulator slowdown vs. execution directly on the host processor. The two bars for each benchmark represent the slowdown without memoization (SlowSim) and with memoization (FastSim).

tion time in slow simulation. Memoization miss recovery time is also negligible, ≤ 0.01% of total simulation time. Unfortunately, the fast simulator is slower than the simulator without memoization (i.e., $t_{fast} > t_0$) because of extra overhead from interpreting memoized action sequences and because not enough run-time static code can be skipped. Moreover, 28% of fast simulation time is spent verifying index entries, so reducing the number of index verify operations is a good place to start to design a more efficient memoizing simulator.

Figure 6.7 shows the amount of memoization cache data allocated. The amount of memoized data ranges from 704 KBytes (compress) to just over 12 MBytes (gcc). In the base simulator, the amount of memoized data corresponds to the number of different PC addresses simulated. In other simulator versions, the amount of memoized data may depend on other factors in addition to

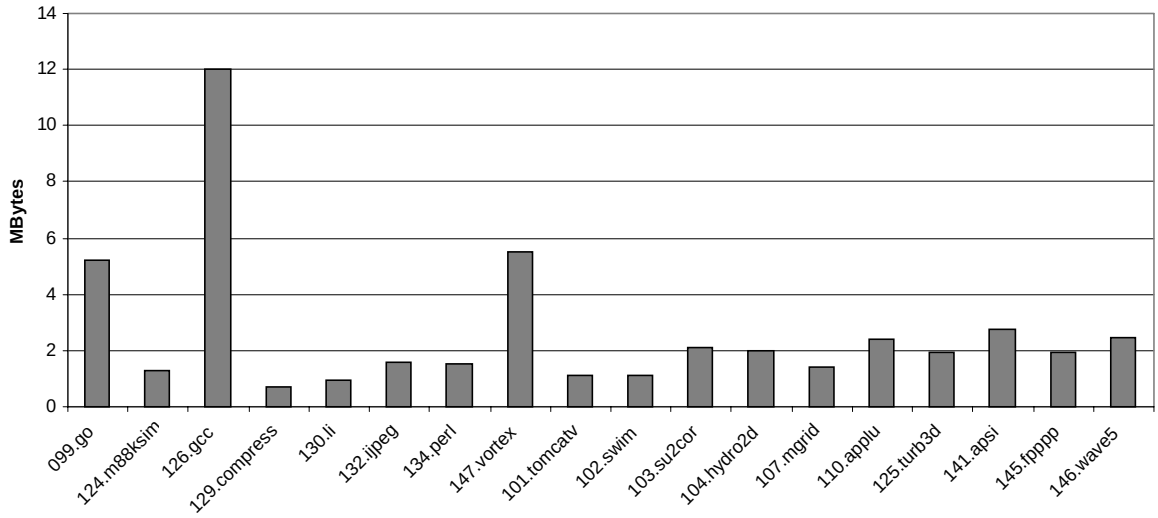

FIGURE 6.7: Base simulator cache size. The base simulator allocates between 704KB (compress) and 12MB (gcc) of data into the memoization cache.

the number of different PC addresses simulated. These factors are discussed in the following sections.

6.3.2. More Simulation Per Call To Main

Increasing the number of instructions simulated per call to `main` decreases the relative time spent looking for memoization index entries. The time to lookup an index entry does not change, but the time spent in other parts of the simulation is increased. Hence the fraction of execution time spent in index lookup is inversely proportional to the number of instructions simulated per call to `main`. But simulating more instructions per call to `main` also increases the average number of dynamic result values tested per index entry. Increasing the number of result values tested increases φ, hence it increases the number of memoization misses and the amount of work performed in miss recovery mode.

```
sem call {
    nPC2 = PC + disp30?sext(32)<<2;
    Rx(15,PC?addr?ext(64)); taken = taken + 1;
};

sem jmpl {
    nPC2 = (Rx(rs1) + SRC2)?cvt(stream)?static;
    Rx(rd,PC?addr?ext(64)); taken = taken + 1;
};

sem [                   bne     be      bg      ble     bge     bl
        bgu     bleu    bcc     bcs     bpos    bneg    bvc     bvs ] {
    if(cond) { nPC2 = PC + disp22?sext(32)<<2; taken = taken + 1; }
    else if(a) annul();
} where cond in [       i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs];

fun main(pc, npc)
{
    PC = pc; nPC = npc;
    nPC2 = 0?cvt(stream); taken = 0;

    while(taken == 0) {
        nPC2 = nPC + 4;     // default next nPC
        PC?exec();          // execute instruction
        PC = nPC; nPC = nPC2;
    }

    init = (PC, nPC);
}
```

FIGURE 6.8: Looping simulator source. The global variable `taken` is incremented whenever a call, indirect jump, or a taken conditional branch is simulated. The `main` function loops while `taken` is equal to 0.

Figure 6.8 shows how the base simulator is modified to simulate more instructions per call to `main`. The idea is to simulate instructions until a taken branch, call, or indirect jump instruction is simulated. Call this the looping simulator. Note that several non-control transfer instructions are simulated by each call to `main` before the taken variable is incriminated and `main` returns. There

are 7.2 (gcc) to 226 (mgrid) instructions simulated per call to `main`, with an average of 36.2 instruction per call to `main` across all the benchmarks.

TABLE 6.4: Looping simulator configuration.

Simulator	Args-to-main	Combining	Inlining	Insts./main
looping	pc, npc	no	no	~36.2 (1br.)[a]

a. Looping until 1 taken control transfer instruction is encountered simulates 36.2 instructions on average for the SPEC95 benchmarks.

The looping simulator design significantly improves memoized simulator performance over the base simulator. The memoized looping simulator suffers only 146 times average slowdown across all benchmarks, compared to 296 times slowdown in the memoized base simulator. Only 5.3% of fast simulation time is spent verifying index entries, compared to 28% in the base simulator. Additional performance is gained because the looping simulator design reduces the average number of actions associated with each simulated instruction by 43% over the base simulator. This effect on action's per instruction is difficult to predict, since it depends on the structure of the dynamic control flow graph produced by complex compiler analyses.

The looping simulator memoized less data than the base simulator in general (figure 6.9). This is because there are fewer index entries and actions. The exception is go: The go benchmark memoized slightly more data in the looping simulator than in the base simulator. The reason is because there are more dynamic results tests in the simulator than just testing whether a conditional branch is taken. In particular, save and restore instructions perform one or more dynamic

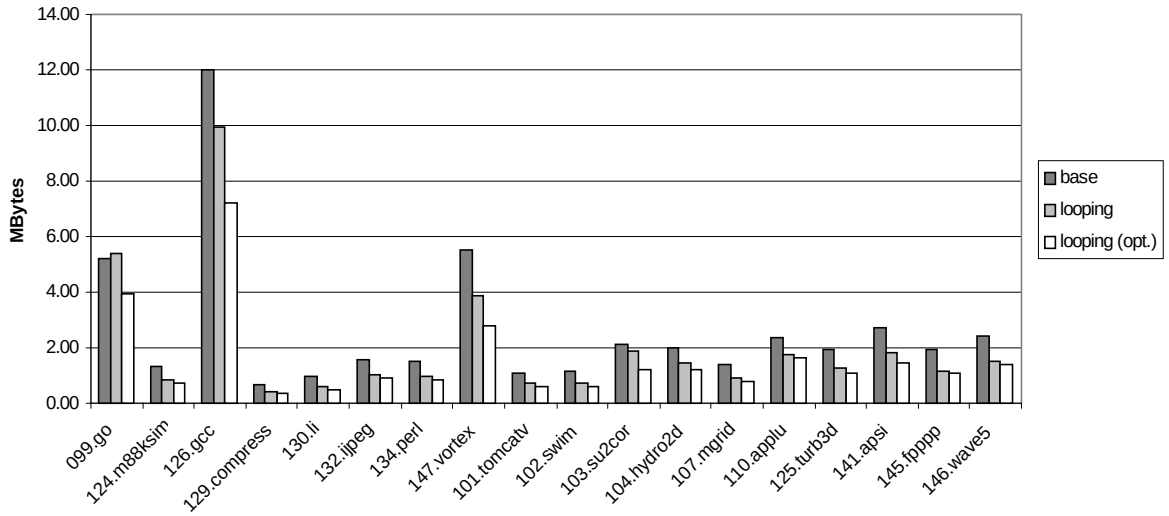

FIGURE 6.9: Looping simulator cache size. This chart shows the amount of data memoized for each benchmark in the base, looping, and optimized looping simulators.

result tests to determine whether the simulated register windows must be spilled or restored respectively. The go benchmark makes many function calls to short functions, which begin with a SAVE instruction and end with a RESTORE instruction, and has a deep call stack that results in frequent register window spills and restores. Hence the action sequences generated for go contain a higher percentage of dynamic result tests that fail than in other benchmarks. More frequent and less predictable result tests causes more memoization misses (i.e. increase φ), causing the slow simulator to run more often and memoize more data.

The looping simulator can be further optimized by reducing the number of arguments to `main`, with little or no impact on the number of index entries, actions, dynamic results, or run-time static data. The idea is to only exit from `main` if `nPC` equals `PC+4`. Then drop `npc` from the list of arguments to `main`, since it can be recomputed from the value of `pc` by adding 4.

TABLE 6.5: Looping simulator configuration (optimized)

Simulator	Args-to-main	Combining	Inlining	Insts./main
looping (opt.)	pc	no	no	~36.2 (1br.)

The optimized looping simulator runs at approximately the same speed as the un-optimized looping simulator. The difference is in memoization cache consumption (figure 6.9). The optimized looping simulator memoizes 17% less data than the un-optimized lopping simulator on average. The reason is twofold: 1) Index entries are smaller by 4 bytes each—the size of the token stream value `npc`—and 2) less work is performed by the slow simulator. The optimized simulator had 6% fewer memoization misses than the un-optimized simulator—1.2% fewer memoization misses per fast-forwarded index entry (i.e., smaller φ)—resulting in 25% percent less slow simulation work (i.e., 25% smaller W_{slow}).

By simulating even more instructions per call to `main`, the proportion of time spent looking for index entries can be reduced further. But there is a limit to how much performance can be gained this way, since index lookup only accounts for part of the overall simulation time. Increasing the instructions simulated per call to `main` also increases the number of dynamic result tests per index entry, increasing the number of memoization misses per index and the average length of action sequences that must be rolled back and replayed by the slow simulator in recovery mode.

Figure 6.10 shows how overall performance is affected by increasing the number of instructions simulated per call to `main`. At first, performance improves due to fewer index entries being

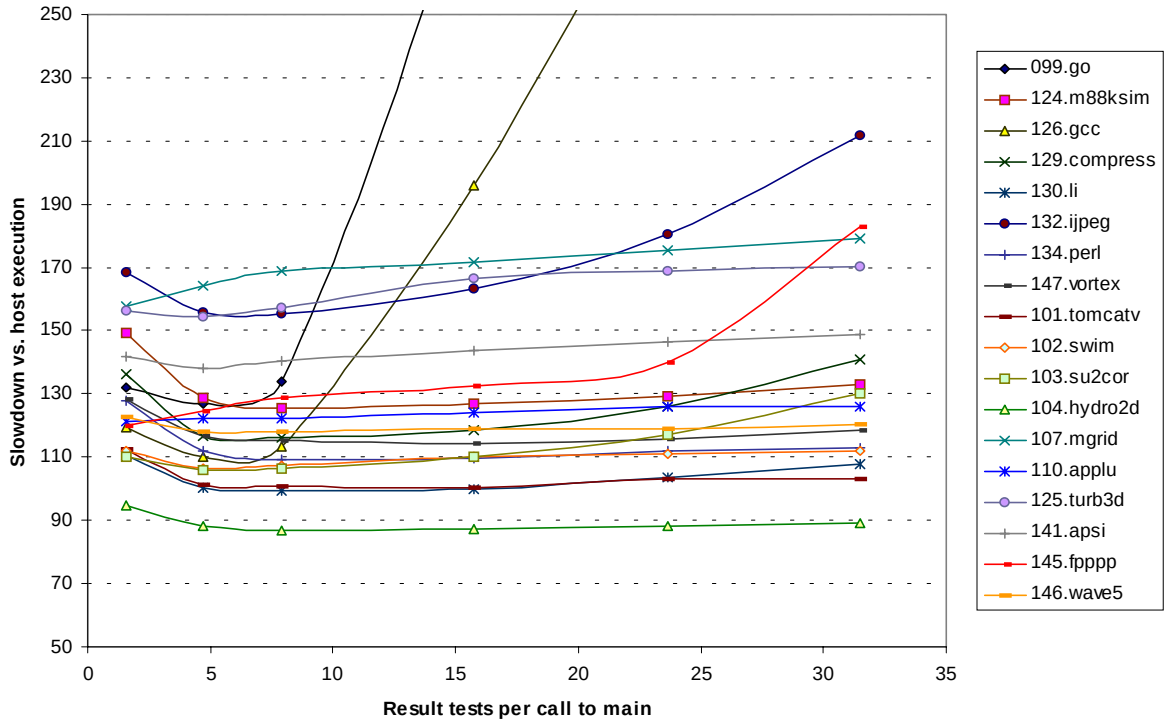

FIGURE 6.10: Performance vs. increased looping. This chart shows how memoizing simulator performance changes as looping—and the average number of dynamic result values tested per index entry—increases.

TABLE 6.6: Simulator configurations with increased looping.

Simulator	Args-to-main	Combining	Inlining	Insts./main
looping w/ 1br.	pc, cwp, cansave, canrestore	yes	yes	~34 (1br.)
looping w/ 3br.	pc, cwp, cansave, canrestore	yes	yes	~102 (3br.)
looping w/ 5br.	pc, cwp, cansave, canrestore	yes	yes	~171 (5br.)
looping w/ 10br.	pc, cwp, cansave, canrestore	yes	yes	~342 (10br.)
looping w/ 15br.	pc, cwp, cansave, canrestore	yes	yes	~512 (15br.)
looping w/ 20br.	pc, cwp, cansave, canrestore	yes	yes	~683 (20br.)

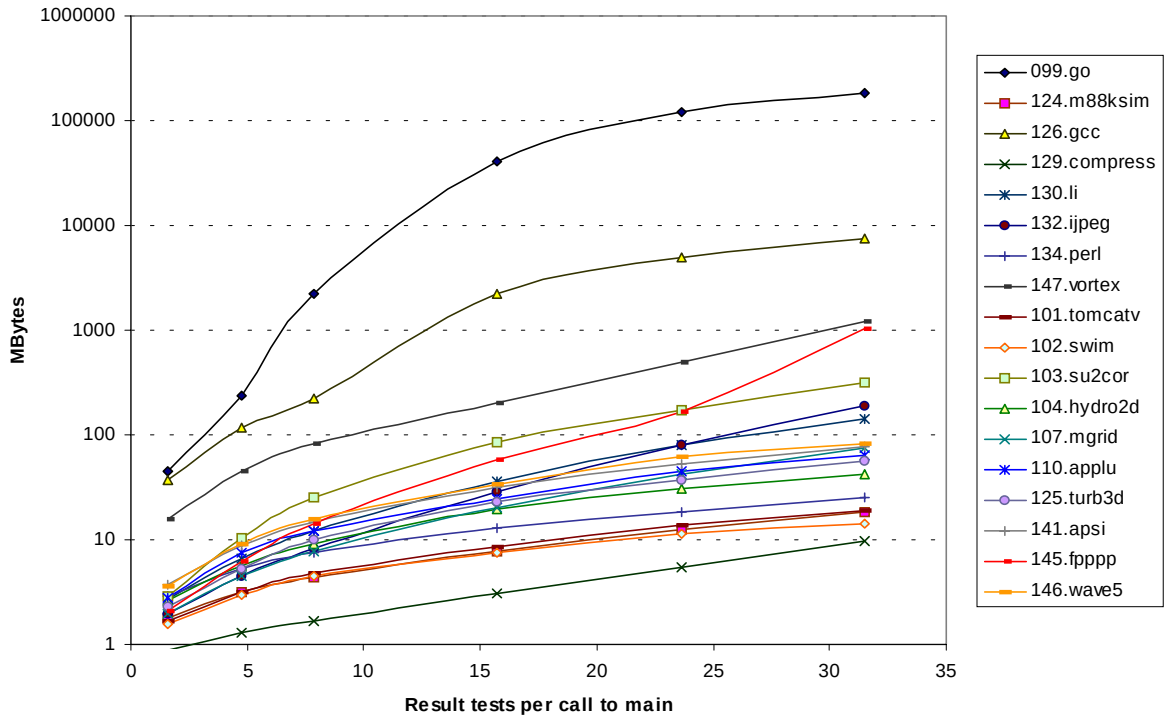

FIGURE 6.11: Cache size vs. increased looping. This chart shows the amount of data allocated in the memoization cache as looping—and the average number of dynamic result tests per index entry—increases. Note that simulations were run with a 256 MByte memoization cache. When the cache is full, it is cleared and new data is allocated into the empty cache.

generated and replayed, but index lookup is only part of the execution, so there is a limit to how much the performance can improve. Increasing the number of instructions simulated per call to `main` increases the amount of memoized data, and this affects performance in two ways: As a the simulator memoizes more data, but before it overflows the memoization cache, performance degrades slowly, because less of the working set of memoized data fits in the host processor's data cache. When a simulation overflows the memoization cache (i.e., memoizes more than 256 MBytes of data) performance drops sharply. Some benchmarks memoize more data than others, so their slowdown spikes upward earlier than those benchmarks that memoize less data.

Figure 6.11 shows how the amount of memoized data increases with increased looping. A simulator's memoized data increases exponentially until it overflows the cache. When the memoization cache overflows, the amount of memoized data increases dramatically, forming a hump in the graph of cache usage for benchmarks that exceed the 256 MByte cache size.

6.3.3. Changing the Proportion of Dynamic Code

The proportion of dynamic code in a simulator can be changed by changing the set of arguments passed to `main`. The base simulator passes `pc` and `npc` as arguments to `main`, making the decoding of target instructions run-time static. One option would be to also pass `cwp` (the current window pointer), `cansave`, and `canrestore` as arguments to `main`. This is a simple change that makes the handling of simulated register windows run-time static, but increases the size of an index entry and may increase the number of index entries in the cache. Figure 6.12 shows how the base simulator is modified to implement this option.

TABLE 6.7: Simulator configurations for run-time static windows experiments.

Simulator	Args-to-main	Combining	Inlining	Insts./main
base	pc, npc	no	no	1
base + cwp	pc, npc, cwp, cansave, canrestore	no	no	1
looping (opt.)	pc	no	no	~34.2 (1br.)
looping + cwp	pc, cwp, cansave, canrestore	no	no	~34.2 (1br.)

```
fun initialize()
{
    R4(14,system?start_sp);
    return (system?start_pc, system?start_pc + 4,
        0b0?ext(5), (NWINDOWS-2)?cvt(cwp_t), 0b0?ext(5));
}

val init = initialize();

fun main(pc, npc, cwp, cansave, canrestore)
{
    PC = pc; nPC = npc; CWP = cwp;
    CANSAVE = cansave; CANRESTORE = canrestore;

    nPC2 = nPC + 4;      // default next nPC
    PC?exec();           // execute instruction

    init = (nPC, nPC2, CWP, CANSAVE, CANRESTORE);
}
```

FIGURE 6.12: Run-time static register window source.

Figure 6.13 shows how simulator performances changes with these added arguments. Figure 6.14 shows the increase in the amount of data memoized.

Adding arguments to the base simulator hurts its performance. The base simulator spends a significant fraction of its execution time looking up index entries, so increasing the size of each index entry further increases the cost of index lookup. The amount of memoized data is also much greater, because each index entry consumes more space and more index entries are generated. There are more index entries because it is less likely that the augmented arguments will match an existing index entry (i.e., lower probability θ and higher probability φ). The increase in memoized

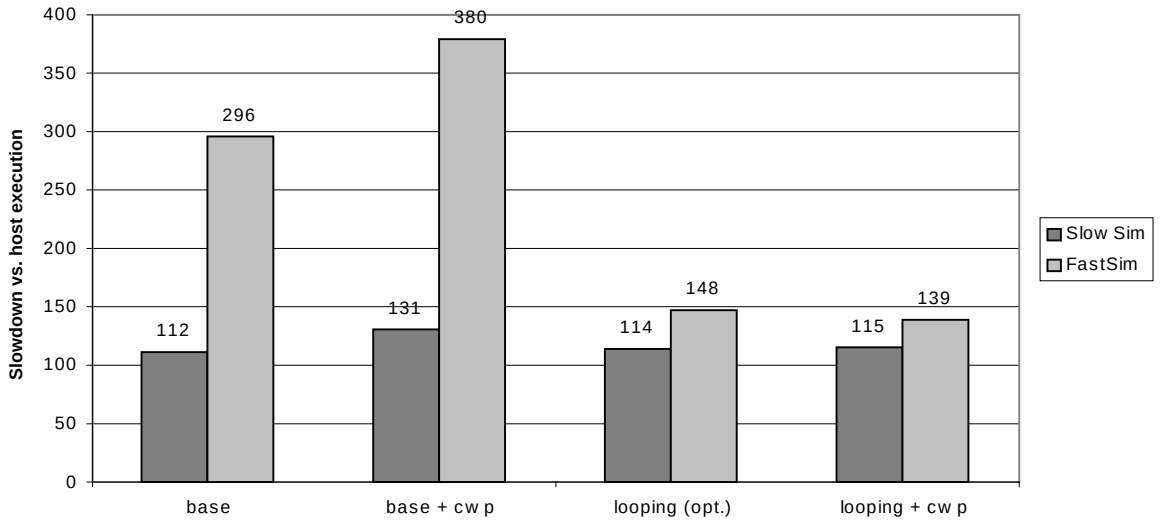

FIGURE 6.13: Performance with rt-stat windows. This chart shows the effect of adding `cwp`, `cansave`, and `canrestore` parameters to `main` on the performance of both the base and optimized looping simulators. This is an average across all SPEC95 benchmarks. Each simulator design is run without memoization (SlowSim) and with memoization (FastSim).

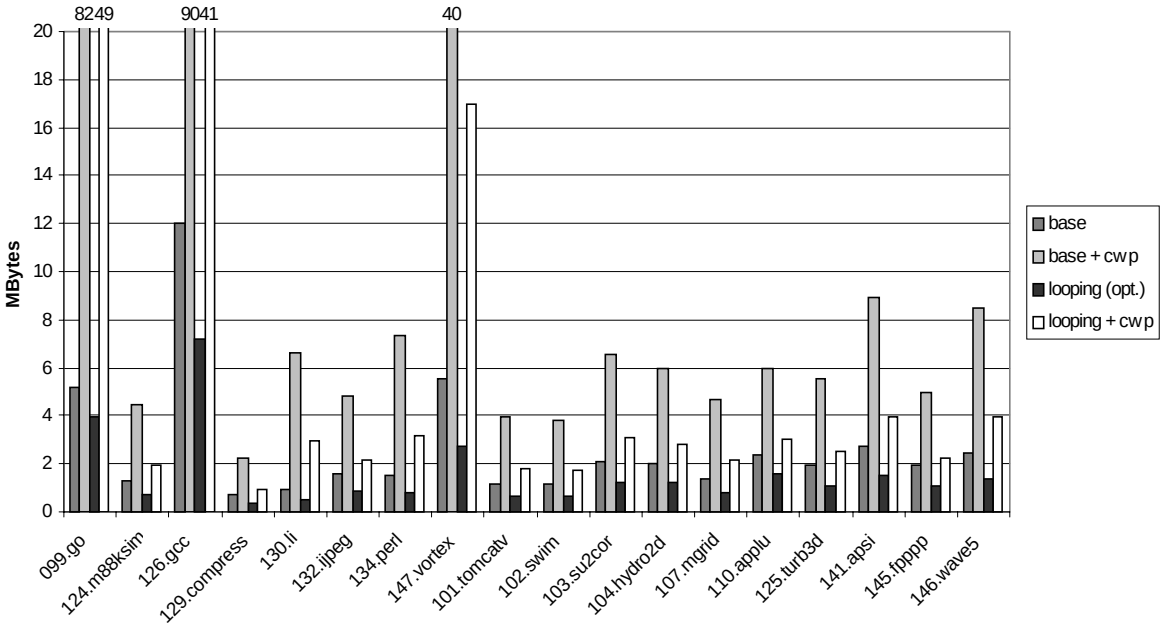

FIGURE 6.14: Cache size w/ rt-stat windows. this chart shows the effect of adding `cwp`, `cansave`, and `canrestore` parameters to `main` on the amount of data memoized by both the base and optimized looping simulators.

data also slows simulator performance by making the host processor's data cache less effective. All these effects overwhelm any benefits gained from executing less dynamic simulator code.

Adding arguments to the optimized looping simulator (from section 6.3.2) hurts performance when not using memoization, but improves performance with memoization. When not memoizing, adding arguments to `main` just adds code to copy the arguments to global variables at the start of `main` without skipping any code. When memoization is used, part of the code that accesses simulated register windows is skipped over because it is run-time static. Since several instructions are simulated per call to `main`, the fraction of simulation time spent looking up index entries and the fraction of memoization cache space consumed by index entries is much less than in the base simulator. So adding arguments to `main` in the looping simulator results in an overall improvement in memoized simulator performance.

6.3.4. Combining Run-Time Static Values

One way to reduce the amount of run-time static data written into the memoization cache is to combine two or more run-time static values into a single run-time static value before using them in dynamic code. When this kind of optimization is possible, it reduces the amount of data written into and read from the memoization cache, and may improve the execution rate of both the slow and fast simulators. It does not affect the number of index entries, actions, or dynamic results allocated or replayed, just the amount of cached run-time static data.

```
val register_windows = array(128) { 0?ext(64) };

fun Rx(i0) {    // get 64-bit register value
    val ii = i0?ext(32);
    if(ii == 0) return 0?ext(64);
    else if(ii < 8) return global_registers[ii-1];
    else {
        val win = (CWP?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); return register_windows[16*win+ii];
    }
}
```

FIGURE 6.15: Combining Run-Time Static Values. This code fragment shows how the array `register_windows` is redefined and the function `Rx` (used to read integer registers) is rewritten to combine the window number and register number before using them to lookup a register. The function to write an integer register is modified in a similar way.

Many instructions in the SPARC ISA read and write integer registers arranged in register windows. In the base simulator, these register windows are modeled as a two dimensional array, where the first dimension is the window number and the second dimension is the register number within a window. If the base simulator is modified to make the current window pointer (CWP) run-time static (as in section 6.3.3) then both the window number and register number are run-time static and are used in a dynamic expression to look up the desired register. These two run-time static values can be combined by modeling register windows as a one-dimensional array and computing the array index from the window number and register number before performing the array lookup. Figure 6.15 shows how the simulator code is changed to combine the window number and register number.

In experiments using the simulators in table 6.8, combining had no effect on the amount of data memoized. Combining, as implemented in these experiments, does reduce the number of

TABLE 6.8: Simulator configurations for value combining experiments.

Simulator	Args-to-main	Combining	Inlining	Insts./main
base + cwp[a]	pc, npc, cwp, cansave, canrestore	no	no	1
combining	pc, npc, cwp, cansave, canrestore	yes	no	1
looping + cwp	pc, cwp, cansave, canrestore	no	no	~34.2 (1br.)
looping + combining	pc, cwp, cansave, canrestore	yes	no	~34.2 (1br.)

a. This and the “looping + cwp” simulators are the same as in section 6.3.3.

run-time static data values being memoized, but alignment restrictions within the memoization cache hide any space savings from storing less data. FastSim stores data into the memoization cache at word (4 byte) aligned addresses. Hence, the combined register index consumes the same 4 bytes as the 1 byte window number and 1 byte register number do separately.

If more run-time static data values that are used in dynamic expressions could be combined into a single value, then less cache space would be consumed. Unfortunately there are few opportunities for this kind of combining in the base simulator or its variations.

6.3.5. Removing Actions Via Function Inlining

If a function that returns a dynamic value is called from more than one call site, then it generates an action into the memoization cache just after it returns. This action specifies what dynamic code (normally the code that copies the function call’s return value) to replay following the called function. If a function is only called from one call site, then an action need not be generated when the function returns, since the function can only return to one place.

Inlining a function has the effect of making a copy of the function that is only called from one call site. If the function returns a dynamic value, it does not need an extra action to identify its return site. This reduces the number of actions in an action sequence, if an action is being generated after the function call anyway to distinguish between other control flow branches in the dynamic control flow graph. Unfortunately, function inlining also increases code size, and too great an increase in code size may degrade simulator performance.

In the simple simulator in Appendix B, the function `get_src2` is used to get the second operand for most integer arithmetic instructions. The dynamic control flow graph following most calls to `get_src2` necessitates an action number in addition to the action number identifying the function's return site. Hence, inlining `get_src2` eliminates one action from the action sequences generated to simulate most integer arithmetic instructions. In Facile, all calls to a function can be inlined simply by inserting the keyword `inline` before the function's definition.

TABLE 6.9: Simulator configurations for experiments with inlining.

Simulator	Args-to-main	Combining	Inlining	Insts./main
base	pc, npc	no	no	1
base + inlining	pc, npc	no	yes	1
looping (opt.)	pc	no	no	~34.2 (1br.)
looping + inlining	pc	no	yes	~34.2 (1br.)

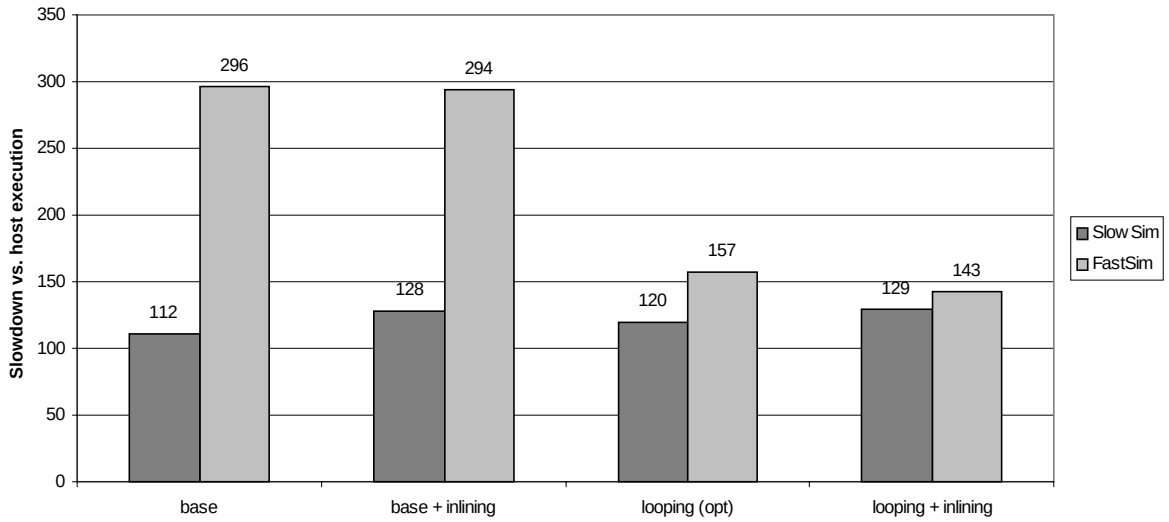

FIGURE 6.16: Performance with function inlining. This chart shows the effect of select function inlining on the performance of both the base and optimized looping simulators. This is an average across all SPEC95 benchmarks. Each simulator design is run without memoization (SlowSim) and with memoization (FastSim).

Figure 6.16 shows how function inlining affects simulator performance. When not using memoization, inlining degraded the performance of both the base and looping simulators. This is because of increased code size and because the Facile compiler does not inline functions efficiently. With memoization, inlining the selected function calls improves overall performance, despite the slowdown from increased code size and inefficient inlining. The base simulator replays 8.4% fewer actions per index entry with inlining, and the looping simulator replays 14.9% fewer actions per index entry. This results in 3.6% and 8.3% reductions in the amount of data memoized by the base and looping simulators respectively. Note that the number of index entries, number of dynamic result tests, and the amount of cached run-time static data used in dynamic expressions are all unchanged. Only the number of actions is reduced, and more simulation is per-

formed per action. The increased performance from generating fewer actions in slow simulation and from reading and interpreting fewer actions in fast simulation outweighs the slowdown from increased code size. Hence, overall simulator performance was improved by inlining.

CHAPTER VII: Future Optimizations

Several optimizations were not implemented in the current version of the Facile compiler and run-time system, either because of limited time or because their usefulness was not realized until later. Run-time code generation (RTCG) is an obvious optimization to apply to fast-forwarding simulators, since fast-forwarding works by interpreting actions stored in the memoization cache. Generating native machine instructions directly into the memoization cache in place of interpreted action numbers would improve performance the same way compiling code improves performance over interpreting it. Run-time code generation was not implemented in the current system because of its complexity, but two possible implementations are discussed in section 7.1.

Other opportunities for optimization were discovered after FastSim v.2 was mostly implemented. Some of these optimizations and the analyses they rely on were added to the implementation because performance was too poor without them. For example, initially all function calls were inlined. Selective inlining improved performance, but required enhancing several analyses (e.g., control-flow and binding-time analysis) to work inter-procedurally. Optimizations that were implemented in FastSim v.2 are discussed in Chapter V. Other optimizations and analyses were not added to FastSim v.2. These future optimization are discussed here, in section 7.2. They include partial index verification, live variable analysis, and optimizing pattern cases in switch statements.

7.1. Run-Time Code Generation (RTCG)

In the current implementation of FastSim, the fast version of a memoizing simulator interprets sequences of action numbers stored in the memoization cache. Fast simulation could be optimized further by compiling dynamic code directly into the memoization cache at run-time, and executing that code instead of interpreting action numbers. With RTCG there would be no fixed fast simulator function, since fast simulation would execute run-time generated code stored in the memoization cache. This eliminates the overhead of reading action numbers and executing a switch statement to select dynamic basic block code.

RTCG is not currently implemented in the FastSim v.2 simulation system, but I have considered two possible implementations that could be used. Section 7.1.1 describes a simple technique for generating run-time code templates and stringing them together in the memoization cache. Code templates for each dynamic basic block are written directly into the memoization cache in place of the action numbers and run-time static data records used in the current implementation. Section 7.1.2 describes a more complex run-time code generator that optimizes register allocation across several dynamic basic blocks to generate more efficient and more compact run-time code.

7.1.1. Implementation 1: Basic Block Code Templates

Consel and Nöel described a way to generate code templates for RTCG, which could be used in FastSim [19]. The basic idea is to generate dynamic basic blocks as C code, compile the C code into native host instructions, then extract compiled basic block code from the object file. Consel and Nöel used tree grammars to encode a conservative approximation of the dynamic control flow

graph of a program. The C code they generate is constructed using these tree grammars, so its control flow graph is also a conservative approximation of the source program's dynamic control flow graph. In this way, a C compiler can perform optimizations between basic block templates (e.g., inter-block register allocation and instruction scheduling) at compile time. At run-time, compiled code templates are simply strung together and populated with run-time static data to produce run-time specialized code.

FastSim v.2 could be modified to use a variation of Consel and Nöel's technique with relative ease. The Facile compiler already generates C code for dynamic basic blocks as individual cases in a switch statement within an outer loop. With minor modifications, similar C code can be compiled and turned into code templates for run-time code generation. In this implementation, compiled C code templates contain the correct instructions to access dynamic variables and call external functions, with efficient register allocation and instruction scheduling within each block. A simple analysis of the compiled code and symbol table identifies the instructions corresponding to each dynamic basic block—extracted to form a code template—and the address relocations that are needed when the template is copied into the memoization cache. More analysis would be needed to identify holes in the code templates where run-time static data is inserted. These holes are filled by the slow simulator, when a template is copied into the memoization cache and run-time static data becomes available.

With C code for each basic block being put in a separate case in a switch statement in a loop, the C compiler cannot allocate registers, schedule instructions, or perform other optimizations

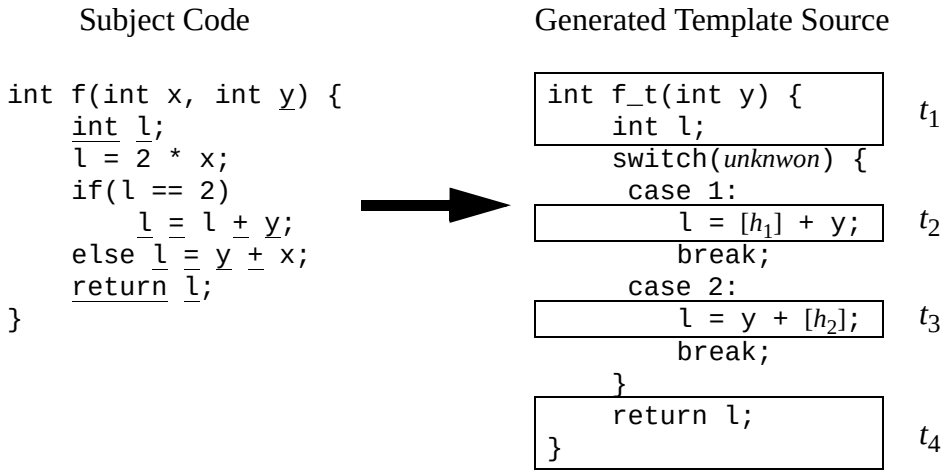

FIGURE 7.1: C code for basic block templates. This subject code and template source is taken from examples in [19]. Dynamic subject code is underlined; Other subject code is either static or run-time static. Four code templates are generated. This template source is compiled and binary code templates are extracted from the object file. h_1 and h_2 are holes where run-time static data can be inserted.

between code templates. Consel and Nöel allowed the C compiler to optimize across some block boundaries by generating C code that implements a conservative approximation of a subject program's dynamic control flow graph. Figure 7.1 demonstrates how this is accomplished. A C compiler can use the declarations of `y` and `l` in template t_1, and the result computed by t_2 or t_3 can be left in a register and used by template t_4.

A similar technique can be used for RTCG in FastSim, but there is an added limitation. Typical RTCG systems generate specialized code for an entire function, but RTCG for a fast-forwarding simulator only generates code for the control flow paths that are executed by the slow simulator. When fast simulation tests a dynamic result value, a memoization miss may occur, because code for the current value is not yet generated. Run-timed generated code must flush data

from host registers back to memory before miss recovery can commence. Hence, the C code generated for templates cannot allow any register allocation assumptions to persist beyond the end of a template that tests a dynamic result. One way to accomplish this is to generate C code for these templates that ends in a return statement. Since the fast simulator stores all its values in global variables, a return will result in all global variable values being flushed to memory.

At run-time, instead of writing action numbers, code templates for a dynamic basic block are copied to the memoization cache just before the first statement of the block is executed by the slow simulator. In addition to copying template code, relocations are applied to fix up any variable or function addresses used by the template. Then, as run-time static values that are used in dynamic expressions become available, the holes in the code template are filled in by encoding computed values as literal data into the template's instructions.

Index entries and dynamic result data would still be stored as data in the memoization cache, with a function call in the run-time generated code to verify an index or result value respectively. As in the current (non-RTCG) implementation, these functions verify that the current dynamic data matches data already in the memoization cache, then jump to the next sequence of run-time generated code. To allow recovery from memoization misses, these functions must also record the sequence of dynamic values tested since the last index entry was encountered. If there is a memoization miss, the sequence of dynamic values already tested by memoized code is needed to execute the slow simulator in recovery mode until it catches up to the failed fast simulation.

7.1.2. Implementation 2: Optimization Across Basic Blocks

The RTCG implementation discussed in the previous section uses an existing compiler to generate compiled code templates, but misses many opportunities to optimize between code templates. A more complex implementation involves compiling code at run-time from an intermediate representation of the dynamic Facile code. At run-time, intermediate code for several dynamic basic blocks is strung together, then converted to native machine instructions by the simulator. Register allocation and simple optimizations are applied to dynamic sequences of operations from several basic blocks, not just block sequences known at compile time, so more compact and efficient run-time code can be generated.

To make the run-time compiler as efficient as possible, most compilation decisions should be made by the Facile compiler before a simulator is executed. For example, specific SPARC instructions can be selected by the Facile compiler for all dynamic operations in a simulator other than loading and storing values of Facile variables. The slow simulator would emit these pre-selected instructions, plus extra instructions to load and store variable values as directed by a run-time register allocator. This register allocator is responsible for remembering which variables are already loaded into registers, and emitting extra instructions to store modified values and load values that are not already in registers. Registers to retain intermediate expression results can be allocated by the Facile compiler at compile-time, with the run-time register allocator designed to work around these previous allocation decisions. Alternatively, registers for intermediate values could be allocated at run-time, increasing the cost of run-time compilation but benefiting from a more uniform register allocation.

Consider a simulator that simulates several target instructions in each call to the Facile `main` function. If a simulated target instruction computes a value and stores it in a Facile variable and the next simulated target instruction reads that Facile variable, then run-time generated code may not have to load that variable to simulate the second target instruction. Similarly, if a variable is modified more than once, the run-time generated code may eliminate all but the last store to save the variable's final value. An ideal RTCG implementation would generate code that loads each simulated register that is read by a single call to `main` once, simulate each target instruction with a single host instruction, then store modified simulated registers with one instruction each.

The advantage of performing register allocation at run-time is that variables used in more than one dynamic basic block can be left in registers, eliminating some loads, stores, and address calculations from the run-time generated code. This differs from the inter-block register allocation performed by a C compiler described in the previous section, because the particular linear sequence of dynamic basic blocks is known—i.e. all run-time static control flow decisions have been made. The disadvantages are slower generation of run-time code and a more complex implementation.

With FastSim's memoization cache organization, all variables stored in registers have to be flushed out to memory before reaching the next memoized index entry (i.e., the next call to `main`). No register can remain allocated following an index entry, and variable values must be loaded from memory before they can be used. This is because the code that precedes an index entry changes dynamically every time the slow simulator matches the index entry's data, adds a

jump to the code associated with the matched entry, and transitions to fast simulation. The register allocation map prior to an index entry cannot be determined, so all registers are flushed to memory and new register allocations made as needed.

A similar problem occurs when dynamic results are tested against previously cached results to make them run-time static. In this case, the register allocation map prior to the dynamic test is known, but it may not be possible to store modified variable values back to memory after a memoization miss. Normally, when not testing a dynamic result, modified values are written back to memory in subsequent run-time generated code. But there may not be any more run-time generated code following a dynamic result test, so another mechanism is needed. One alternative is to generate extra run-time code to save variable values in the case of a memoization miss, but this would generate extra code into the memoization cache for every instance of a dynamic result test. Another alternative is to flush modified variable values back to memory before the dynamic test. Subsequent run-time generated code could still use loaded register values without having to reload them, but extra stores may be generated if a modified variable is modified again before the next index entry.

Other optimizations that could be performed by run-time code generation include instruction scheduling and combining operations from separate dynamic basic blocks into single host instructions. The exact linear sequence of dynamic basic blocks is known to the run-time code generator, so a run-time instruction scheduler could schedule instructions better than any compile-time code generator. Some operations in separate dynamic basic blocks that are not combined because of

unknown control-flow at compile-time, could be combined at run-time. Combining operations from separate dynamic blocks at run-time could reduce the amount of code in the memoization cache by generating two or more operations as a single host instruction.

An ideal RTCG system, applied to a simulator that only models the functional behavior of an ISA, would generate one host instruction for each target instruction plus a few extra loads and stores to access simulated registers stored in simulator variables. Applied to a simulator with additional behavior, e.g., modeling micro-architecture or collecting execution statistics, instructions for additional behavior would be intermixed (or even combined) with re-compiled target instructions. This is similar to existing simulators that use dynamic cross-compilation (e.g., Shade [17]), but allows greater flexibility in the kinds of simulation: Any simulator that can be written in Facile could be optimized with memoization and RTCG to behave like a dynamic cross-compiling simulator.

7.2. Further Optimization

The experience gained implementing and analyzing the memoizing simulators in Chapter VI revealed several compiler optimizations that could improve performance of memoizing simulators in Fastsim v.2. These optimizations were not implemented in the current version of the Facile compiler to simplify its implementation or because their importance was discovered too late.

One optimization is for the fast simulator to only verify the parts of an index entry that may contain dynamic data. Another optimization is to avoid generating extra statements to copy data

from run-time static variables to their dynamic versions, when the variable is not live. A third optimization is to generate more efficient code to decode instructions. Facile switch statements are translated into C code that is reasonably efficient, but a couple optimizations could further improve their performance.

7.2.1. Partial Index Verification

When the fast simulator reaches the end of the action sequence associated with an index entry, it looks for the next index entry, so it can continue replaying memoized simulation. The fast simulator first compares the current `init` value against the index entries listed as known successors to the completed action sequence. If none of these index entries matches `init`, then the fast simulator looks in the index for the entire memoization cache, or if this fails, it switches back to slow simulation. Either way, the new matching index entry is added to the list of known successors for the completed action sequence. In Chapter VI, verifying index entries in the base and out-of-order simulators accounted for 28.1% and 1.2% of fast simulator execution time respectively.

In the current implementation, the entire value of `init` is compared against the entire value stored in each known successor. But the only parts of `init` that can miss-match a known successor are the parts that are set with dynamic data. Run-time static data stored in `init` will always match the corresponding parts of the index entries listed as known successors. Hence, index verification in the fast simulator can be optimized by only comparing the dynamic parts of `init` to the corresponding parts of the index entries listed as known successors. Note that index lookup in

the full memoization cache index must still compare the entire value of `init`, since the context that generated these index entries is unknown.

To implement this optimization, the Facile compiler would generate a function to compare only the dynamic parts of `init` to an index entry. Then FastSim's run-time library would use this function—instead of its generic index compare function (i.e., memcmp)—to look in the list of known successors. Only if `init`'s current value is not found among the known successors, would the run-time library compute a hash value and use the generic compare function to find a matching entry in the index hash bucket.

7.2.2. Live Variable Analysis

Extra statements are generated when the fast-forwarding optimization is applied. Many of these extra statements copy run-time static variable values to the dynamic version of the same variables. Copying occurs whenever a control flow path with run-time static variables merges with another control flow path, where the same variables are dynamic. These extra statements are part of the dynamic code in a simulator, so they appear in both the slow and fast simulator versions. These statements also use run-time static data in dynamic code, so the copied data is also stored in the memoization cache. But these statements are sometimes unnecessary, because the variable's value at that point is never used by subsequent code. A more efficient implementation would not bother copying variables if their values are never used.

A variable *v* is considered live at some point in a program *p*, if *v* is read before it is written along some control flow path reachable from *p*. The set of live variables at every point in a program can be computed with a fixed-point iteration algorithm, similar to the algorithms used for BTA and to construct a dynamic control flow graph (see Chapter V, sections 5.2.2.2 and 5.2.4.1 respectively). The difference is that live variable analysis must move backwards through the control flow graph, from later statements to earlier statements. Each variable that is read is added to the set of live variables in earlier code, and each variable that is written is subtracted from the set of live variables in earlier code. Extra statements to copy run-time static variables to their dynamic versions must still be generated, but only if the variable being copied is live.

In the current implementation, non-live variables are frequently copied by extra statements. One primary example involves global variables that are assigned run-time static values at the start of `main` and remain run-time static throughout the entire `main` function. When `main` returns, these variables become dynamic, so extra statements are generated to copy all run-time static data in global variables into the corresponding dynamic variables. Most of this copying is unnecessary, since the next call to `main` writes new run-time static data into the global variables without reading them first. With live variable analysis, extra statements like these could be eliminated from the slow and fast simulators, reducing the amount of dynamic code and the amount of run-time static data stored in the memoization cache.

7.2.3. Optimizing Instruction Decode

Instruction decoding is handled by Facile's switch statement or a collection of semantic (`sem`) declarations that are interpreted as cases in an implicit switch statement. The Facile compiler transforms pattern cases in a Facile switch statement or instruction/pattern names in `sem` declarations into a collection of nested if and switch statements in C. This transformation closely follows the structure of the patterns given in the pattern cases. It tests conditions on token fields in the order the conditions are listed in the source pattern expressions, generating separate tests for each condition in an AND-list. Two ways to optimize the transformed switch would be 1) to reorder the conditions in all the tested AND-lists to perform the most important tests first, and 2) to combine some conditions in an AND-list into a single test.

A possible strategy for reordering conditions is to find a test that applies to the largest number of cases in the pattern case list, and perform that test first. For example, every SPARC instruction must test the `op` field (bits 30:31), so generate a switch statement to test this field first. Once that test is performed, and conditions on the `op` field have been removed from the AND-list of every pattern case, find the next largest number of cases that share the same test, and so on. In my SPARC simulator implementations, I have been careful to arrange the conditions in pattern expressions to produce this optimized ordering. The Facile compiler could be made to reorder conditions this way automatically, in case the programmer wrote less optimal code.

Often, several pattern cases contain conditions on the same subset of token fields. It may be more efficient to test the combined values of all these fields simultaneously, with a single test,

than to generate a separate nested test for each field. For example, a single switch could be used to test the `op2` field (bits 22:24) and the `cond` field (bits 25:28) simultaneously, when decoding SPARC branch instructions. Simply extract the `op2` and `cond` fields and concatenate them together (or just extract bits 22:28 to get both fields at once) and switch on their combined value. This is a common optimization performed in hand written C code to decode binary instructions, but is not currently performed by the Facile compiler.

A difficulty with combining field tests is that the set of fields tested for each instruction may be different. E.g., branches test the `op2` field (bits 22:24) but many other instructions test the `op3` field (bits 19:24) instead. One solution is for the compiler to combine field tests only if some subset of conditions on fields is the same in all listed pattern cases. This is unlikely to occur in the first test performed on a token stream, but may occur in nested tests that distinguish between similar instruction types. For example, to decode a SPARC instruction the Facile compiler may first generate a C switch statement to test the `op` field. Then in the `op==0` case—mostly branch instructions—it can test the combined values of the `op2` and `cond` fields to determine exactly which branch instruction should be simulated.

CHAPTER VIII: Conclusion

This work has advanced the implementation of instruction-level micro-architectural simulators with innovative techniques to improve simulator performance. My primary contribution is a form of memoization that accelerates the execution of complex micro-architecture simulators. This optimization—called fast-forwarding—produces an order-of-magnitude speedup in the simulation of out-of-order processors.

The next section (section 8.1) summarizes each of the contributions made in this dissertation. While much work has gone into this research, there are still many opportunities for further research. Section 8.2 discusses potential future work and how it relates to what I have already done.

8.1. Contributions

The contributions in this dissertation include:

- A technique, called fast-forwarding, for memoizing an out-of-order processor simulator to speed its execution.

- A technique—called speculative direct execution—for using direct execution in a speculative out-of-order processor simulator to speed its execution.

- A special purpose programming language design that both simplifies micro-architecture simulator implementation and simplifies the automatic translation and optimization of these simulators to use memoization.

- Compiler techniques for generating memoizing simulators.

- An explanation of the performance factors in a memoizing simulator, with experimental result to illustrate how simulator design changes affect performance.

- Descriptions of how run-time code generation could be implemented in future versions of my memoizing simulation system.

A prototype simulator—FastSim v.1—demonstrated that memoization is very effective at speeding the simulation of complex micro-architecture, but also that fast-forwarding is a difficult optimization to implement by hand. To overcome the complexity of implementing fast-forwarding, I developed a special purpose programming language and optimizing compiler that generates fast-forwarding simulators automatically. The language—Facile—combines syntax for architecture description with general program constructs for programming the rest of an instruction-level micro-architecture simulator. A compiler analyzes Facile simulator code and transforms it into the two cooperating simulator versions needed to implement fast-forwarding.

Facile extends previous work in architecture description languages to allow concise and flexible descriptions of instruction encodings, instruction semantics, and other architectural resources (e.g., registers and memory). Ramsey and Fernandez's New Jersey Machine Code Toolkit—upon which Facile's architecture description syntax is loosely based—could only describe the encoding of binary instructions, and translate them to and from assembly language syntax. Facile describes instruction semantics as well as instruction binary encodings, so Facile's architecture descriptions can be used to simulate target instructions. General programming language constructs in Facile—e.g., functions, loops, and arrays—allow programmers to model arbitrarily complex micro-architectures, including processors with out-of-order execution pipelines.

In addition to supporting the implementation of a micro-architecture simulator, Facile also simplifies the compiler analyses needed to produce a memoizing simulator. Analyses typically used for partial evaluation enable the compiler to identify Facile simulator code that can be skipped over by fast-forwarding, and to generate cooperating slow and fast versions of the simulator. The two primary omissions are pointers and recursion. Neither are necessary for writing simulators. The absence of pointers simplifies alias analysis and allows the compiler to better analyze a subject program's data flow. The absence of recursion simplifies compiler analyses, but is mainly needed to implement an efficient memoization miss recovery mechanism.

The very structure of a Facile simulator identifies the run-time static code and data, which is skipped over by fast-forwarding, in a natural way. The `main` function in a Facile simulator is called repeatedly by the FastSim simulation framework (i.e., by FastSim's run-time library).

Arguments to `main` are assumed to be run-time static and all computation and data that depends only on these arguments and other static data is also run-time static. Action sequences, describing the residual dynamic computation (not skipped by fast-forwarding), are written into the memoization cache by the slow simulator as it executes. The fast simulator replays these action sequences and skips over run-time static code when `main` is called with the same arguments as some previous call.

Facile simulators are optimized using techniques from partial evaluation. Binding-time analysis (BTA) is commonly used in off-line partial evaluators to determine which parts of a program can be evaluated at compile time. The Facile compiler uses a new form of polyvarient BTA to determine which parts of a simulator can be evaluated once at run-time, then skipped by subsequent execution. As in off-line partial evaluation, the Facile compiler uses annotations derived by BTA to generate a residual (fast) version of the subject simulator that only contains dynamic code. Unlike traditional partial evaluation, a slow version of the simulator that does not skip any simulator code is also generated, to run first and generate action sequences that the fast simulator can replay later.

To reduce the opportunity for programmer error, Facile is strongly typed. To reduce the size of programs and ease the burden on programmers, the Facile compiler uses a polymorphic type inference algorithm to typecheck Facile programs. Variable, parameter, and return types are inferred by the compiler and do not have to be specified by a programmer. Facile's type inference algorithm extends the well-known type inference algorithm—algorithm *W* [47]—to allow opera-

tor overloading and user defined function overloading. Functions with the same name but different types are allowed. Type inference finds all possible instantiations of overloaded operator and function names that produce legal typings of a given piece of code.

Although function inlining is not necessary for miss recovery, it can improve memoized simulator performance and is useful for compiling local function definitions from Facile into global functions in C. C does not allow local function definitions—i.e., functions defined within the scope of another function—but Facile does. All calls to local functions in Facile are inlined, and the local function definitions are removed. When fast-forwarding, function calls that contain dynamic code often increase the length of memoized action sequences, because an action number is needed to identify the function's return site. Inlining these functions eliminates some memoized actions, saving space in the memoization cache and speeding up the execution of the fast simulator. A programmer can specify which global functions to inline, potentially improving simulator performance, but care must be taken not to inline too much or performance will suffer.

Experiments with various designs of a simple simulator illustrate the effects of techniques that use memoization as a simulator optimization. Select inlining of functions reduced the number of memoized actions, combining run-time static values before using them in dynamic expressions reduced the number of values stored in into the memoization cache, and adding select parameters to `main` increased the amount of code skipped by fast-forwarding. Unfortunately, inlining increases code size and is difficult to predict its effect on compiler analyses, limited combining of run-time static values does not necessarily reduce memoization cache size, and adding to `main`'s

arguments may increase the cost of index verification more than reducing the cost of fast simulator execution.

The most effective design change to the base simulator (in Appendix B) was to simulate more target instructions per call to `main`. This reduced the number of index lookups in the slow simulator and index verifies in the fast simulator. Index lookup and index verify were responsible for a significant portion of the execution time in the memoized base simulator. Reducing them initially improved simulator performance. But, simulating more instruction's per call to `main` increases the chance of a memoization miss and the cost of memoization miss recovery. It also increases the amount of data in the memoization cache, which degrades data cache performance on the host and slows down the entire simulation. A careful balance between fewer index entries and fewer memoization miss opportunities achieved the best performance.

Fast-forwarding was proven very effective at speeding up the simulation of an out-of-order pipeline. FastSim v.1 used a hand coded implementation of the fast-forwarding optimization to achieve an order of magnitude speedup in out-of-order micro-architecture simulation over a contemporary out-of-order simulator, i.e., SimpleScalar. FastSim v.2 provides a flexible system for writing instruction-level simulators—using Facile—and automatically compiles them to use fast-forwarding. An out-of-order processor simulator written in Facile also achieved an order of magnitude speedup with fast-forwarding over the same simulator without this optimization. However, inefficiencies in the compiled Facile code made the simulator slower than the hand-coded simulator—FastSim v.1 written in C—but it still out performs SimpleScalar.

8.2. Future Work

Potential future work related to this dissertation includes, but is not limited to improvements in the Facile compiler and FastSim simulation system. The Facile compiler does a good job of adding memoization to source simulators, but additional optimizations or further tuning of existing compiler stages could produce more efficient simulator code. One significant optimization would be to use run-time code generation (RTCG) to generate specialized simulation code at run time. Run-time generated code would execute faster than interpreting action sequences, in the same way a compiled program executes faster than evaluating the same program through an interpreter. Section 8.2.1 discusses the potential for RTCG in FastSim.

Simulator fast-forwarding can be applied in other simulation environments, not just FastSim v.2. FastSim only simulates target instructions from user level execution on a single processor. Memoization has not yet been applied to simulators of parallel processor machines or multi-threaded processor micro-architectures. Section 8.2.2 comments on memoizing simulators of multi-processor/multi-threaded systems. Another direction is to combine memoized micro-architecture simulation with full system simulation—like that in SimOS and Simix—to accelerate simulation of both user-level and kernel instructions. Some issues of memoizing both user and kernel instruction simulation are discussed in section 8.2.3.

8.2.1. Memoization With Run-Time Code Generation (RTCG)

Run-time code generation (RTCG) is an important topic of research for programming language optimization. Instruction-level micro-architecture simulation is a domain in which RTCG

could be very effective. Instead of memoizing action numbers and interpreting them to replay previous simulation, specialized compiled code could be generated directly into the memoization cache. Run-time generated code would run faster that an action sequence interpreter, just as a compiled program runs faster than a program evaluated in an interpreter. Two possible implementations of RTCG in a memoizing simulator are discussed in Chapter VII section 7.1.

Instruction-level micro-architecture simulation is a good application for RTCG, because simulation is far more often replayed by the fast simulator than executed without optimization by the slow simulator. Hence, the cost of generating code at run-time can be amortized over many uses of the generated code. Reasonably efficient run-time generated code would execute faster than interpreting sequences of action numbers. Because the vast majority of simulation is replayed by the fast simulator, RTCG could easily improve overall simulator performance.

RTCG for simulator fast-forwarding is different from traditional partial evaluation and RTCG systems. Traditionally, specialized code is generated for an entire function either at compile time (partial evaluation) or at run time (RTCG). Fast-forwarding only generates code (or action sequences) for control flow paths in the simulator that actually execute. This is a necessary feature to save space in the memoization cache, since many run-time specialized versions of a function are generated. For example, the memoizing version of the simulator in Appendix B generates a specialized version of the `main` function for each different target `pc` address simulated. But actions are generated only for the control flow paths that actually execute.

8.2.2. Memoizing Multi-Processor Simulators

Multi-processor systems or multi-threaded processors would not memoize effectively in the current FastSim v.2 simulation system. If a single processor with a single thread caches simulator actions with index entries capable of N possible values, then a multi-processor system with P processors would need memoization cache index entries chosen from N^P possible values. There is no particular reason to expect multiple processors to have strongly correlated run-time micro-architecture states—i.e., a pair of processors is unlikely to repeatedly execute the same pairs of instructions. So expect the number of index entries actually generated to be $O(n^P)$, where n is the expected number of index entries generated when simulating a single processor. Even with just two processors, the blowup in memoized data would make memoizing multi-processor simulation in FastSim v.2 infeasible.

A better approach is to simulate each processor separately, but share the cache of memoized actions. When separate processors are running the same program, they are likely to call the same instructions with similar micro-architecture simulation results. By sharing a single memoization cache, each processor could generate actions for its particular micro-architecture state, and all the processors could share those actions to fast-forward their simulations. An implementation of this approach may simulate each processor in a separate host thread (or find another way to interleave the simulation of the modeled processors) with a memoization cache shared by all the threads.

Multi-threaded processor simulation shares the problems of multi-processor simulation, but may not be as easy to fix. Multi-threaded micro-architectures execute multiple target instruction

sequences (i.e., from multiple threads) within the same processor. In this case, micro-architecture behavior depends on the combination of more than one instruction sequence. The micro-architecture behavior resulting from one thread cannot be computed without considering instructions in other threads, likely generating too many index entries in the memoization cache. Other techniques are needed to reduce the amount of memoized index and result data in this case.

8.2.3. Memoizing Full System Simulators

Today's (micro-)architecture research is increasingly relying on benchmarks taken from real world applications, such as commercial databases (e.g., Oracle or Informix) and commercial user applications (e.g., Microsoft Word or Excel). Unlike simple micro-benchmarks and scientific applications historically used for benchmarking, commercial application performance is strongly affected by operating system and hardware device performance. Application performance may depend on how the operating system provides access to protected resources (e.g., databases accessing the disk), manages virtual memory, implements process scheduling, etc.

Just as memoization accelerates the simulation of a complex micro-architecture on user-level instructions, it could also accelerate the simulation of system-level instructions. One problem is dealing with changes to the instruction text in memory—e.g., resulting from mapping a new process image, changes in the mapping of virtual to physical memory, etc. FastSim v.2 assumes that instruction text cannot change, so the instruction text does not need to be included as part of the run-time static state stored in memoization index entries.

FastSim has a special case for handling instructions in the jump table for dynamically linked library calls. These instructions do change[1], so special dynamic result tests are added to the memoized action sequences verifying that instructions in the jump table have not changed. These extra result tests are only needed when memoizing actions for instruction that can change, and do not affect memoized performance when simulating instructions that cannot change (i.e., they are in read-only memory). In a full system simulator, any instruction can change (e.g., after an exec system call or from re-mapping virtual pages). A similar mechanism could be used to verify the memoized results for all simulated instructions, but this would hurt memoized simulator performance. A better mechanism would be to assume that instructions do not change, and invalidate the entire memoization cache when something happens that violates this assumption.

8.2.4. Memoizing Other Micro-Architecture Structures

Memoization is an effective technique for accelerating the execution of an out-of-order micro-architecture pipeline. How this optimization applies to other micro-architecture structures remains largely unexplored. Other micro-architecture structures include instruction and data caches, branch predictors, value prediction and instruction reuse buffers, etc. For many of these structures, the set of data needed to encode a useful memoization index entry is large and it is impractical to index a data cache simulator on the values stored in the data cache, or even on the addresses of data in the data cache.

1. The first time a DLL function is called the jump table changes to jump directly to the dynamically linked function code.

Although cache-like structures cannot be fully memoized, sometimes parts of their behavior can be memoized. For example, a non-blocking data cache simulator can have complex timing results. This behavior depends partly on whether an address is found in each level of cache, but also on the pattern of address dependencies in a sequence of cache accesses independent of the actual memory addresses. The effect of these address dependencies may be memoizable, even though looking up an address in each level of cache would still be a dynamic action. Similarly, the out-of-order simulators in Chapter III (FastSim v.1) and in Chapter VI section 6.1 memoize the register renaming map—another buffer with many possible values—by only considering the dependencies between instructions.

Memoization is not a universally applicable optimization for all micro-architecture simulation, but it is effective at accelerating some costly types of micro-architecture simulation. Creative simulator designs can memoize parts of seemingly intractable micro-architecture structures. The important property to look for is where a small set of frequently repeated micro-architecture states determines a significant part of simulated behavior. Anywhere this happens, there is an opportunity for improved performance through memoization.

APPENDIX A: FACILE LANGUAGE REFERENCE

This appendix contains a complete reference for the Facile programming language as implemented by the current Facile compiler. It describes Facile's lexical specification, architecture description features, types, and semantic statements and expressions. Note that Facile source files are passed through the C pre-processor (**cpp**) before being given to the Facile compiler, so C pre-processor directives (e.g., `#define` or `#include`) can be used in Facile source files.

TYPOGRAPHICAL CONVENTIONS. The following typographical conventions are used in this reference:

- `fixed-width` for Facile code, and for reserved words and punctuation in Facile grammar descriptions.

- *italic* for non-terminals in Facile grammar descriptions.

- **bold** for terminal symbols that can have many values (e.g., **name**, **int**, and **float**).

Lexical Specification

Facile is case sensitive. Whitespace—e.g., space, tab, newline—is ignored except where it is needed as a token separator. Comments are treated as white space and are either written as `//` to the end of line, or between `/*` and `*/` delimiters. Comments written as `/*` … `*/` do not nest.

Below are lists of all the reserved words and all punctuation used in Facile:

```
array       continue    fun         return      val
as          default     if          sem         var
break       else        in          struct      where
by          extern      inline      switch      while
case        fields      pat         token
const       for         queue       type

(   )   ,   .   ..  :   ;   =   ?   [   ]   _   {   }
$   %   &   &&  *   +   -   /   <<  >>  ^   ^^  |   ||  ~
<   <=  ==  !=  >=  >
```

Function, variable, type, and attribute names are any alpha-numeric words that are not already defined as reserved words. Names are separated by whitespace or punctuation (except for the wildcard mark '_' which can be part of a name). They can not have a numeric digit as their first character, although numeric digits can be used later in a name. I.e., names are of the form `[A-Za-z_][A-Za-z_0-9]`*.

Literal values are integers (in decimal, octal, hexadecimal, binary, or character form), floating point constants, and strings. Decimal and octal constants are written as `[1-9][0-9]`* and `0[0-7]`* respectively, and are interpreted as 32-bit unsigned values. Hexadecimal and binary constants are written as `0x[0-9A-Fa-f]`* and `0b[01]`* respectively, and are interpreted as unsigned values with a bit-widths just large enough to encode all the digits appearing in the literal. E.g., 0xFF represents the 8-bit wide integer with value 255, and 0b11001 is the 5-bit wide integer

with value 25. Character literals are interpreted as 1-byte (8-bit wide) unsigned integers, and are written as `'`*c*, where *c* is any single character or escape sequence. Escape sequences are newline `\n`, tab `\t`, double-quote `\"`, backslash `\\`, or any ASCII character number written in hexadecimal as `\[0-9A-Fa-f]`$^{\{1,2\}}$`x` (e.g., `'A`, `'\n`, `'\13x`).

Floating point values are written as `-`$^{?}$`[0-9]`*`.[0-9]`*`([Ee][-+]`$^{?}$`[0-9]`$^{+}$`)`$^{?}$ and are interpreted as 8-byte (64-bit) IEEE floating point values. There must be at least one digit before or after the decimal point. The following are some possible floating point literals: `5.0`, `.15`, `-7.`, and `1.0e-9`.

String literals are written as sequences of characters and character escapes between double-quote delimiters (`"`). The recognized character escapes are newline `\n`, tab `\t`, double-quote `\"`, backslash `\\`, and any ASCII character number written in hexadecimal as `\[0-9A-Fa-f]`$^{\{1,2\}}$`x`. Strings cannot span multiple lines. To include a newline in a string literal, use the `\n` character escape.

Architecture Description

An instruction set architecture (ISA) is described using token, field, pattern, and semantic declarations. Tokens define fixed width groupings of bits in an instruction stream (also called a token stream) and fields are contiguous sequences of bits within a token. Patterns are sets of conditions on the values stored in fields within a token stream and are used to distinguish the binary encod-

ings of instructions in an ISA. Semantic declarations map pattern names to semantic code. This semantic code usually simulates target instructions, but can be anything the programmer desires.

global_stmt	→ `token` **name** `[` **int** `]` *fields_opt* `;`	token declaration
fields_opt	→ `fields` *field_dec_list*	field declarations
	→	
fields_dec_list	→ *fields_dec_list* `,` *field_dec*	
	→ *field_dec*	
field_dec	→ **name** **int** `:` **int**	bit range field
	→ **name** **int**	single bit field

A `token` declaration statement defines a token with the given name and bit width. The token name is also defined as a field name that refers to the entire token. Additional fields are defined using an optional `fields` clause following the token definition. Field names are give in a comma separated list, and each name is associated with a single bit or a range of bits within the token. Token bits are numbered in big-endian order starting at 0 up to the token width minus 1—i.e., bit number 0 is the right most (least significant) bit and bit numbers increase to the left. For example, in a 32 bit token, the first byte (8-bits) contains bits 24:31, the second byte contains bits (16:23), etc.

global_stmt	→ `pat` *pnames* `=` *pat_exp* `;`	pattern name declaration
pnames	→ `[` *pname_list* `]`	
	→ **name**	
pname_list	→ *pname_list* *pname*	
	→ *pname*	

pname	→ **name**	
	→ _	

A `pat` declaration associates mnemonic names with patterns that describe the binary encodings of instructions. A pattern is represented as a collection of conditions on token fields in disjunctive normal form—i.e., an OR-list of AND-lists of conditions on token fields. Patterns are constructed with pattern expressions, called *pat_exp* in the grammar.

pat_exp	→ *pat_field op* **int**	single condition
	→ *pat_field* `in [` *pint_list* `]`	multiple condition
	→ *pat_exp* `||` *pat_exp*	OR two patterns
	→ *pat_exp* `&&` *pat_exp*	AND two pattern
	→ *pat_exp* `$` *pat_exp*	concatenate patterns
	→ **name**	named pattern
pat_field	→ **name** `?` **name** `(` *int_list* `)`	attributed field name
	→ **name**	field name
int_list	→ *int_list* `,` **int**	
	→ **int**	
pint_list	→ *pint_list pint*	
	→ *pint*	
pint	→ **int** `..` **int** `by` **int**	range of integers with step
	→ **int** `..` **int**	range of integers with step 1
	→ **int**	single integer

A single condition compares a token field to a single integer value using one of the comparison operators `<`, `<=`, `==`, `!=`, `>=`, or `>`. The `in` expression generates multiple comparisons OR'ed together that test if the token field is equal to any of the listed integers. A range of integers can be abbreviated using an ellipsis (`..`) with an optional step argument. For example, the pattern

expression `cond in [ 0x0 .. 0xf ]` is the same as OR'ing together 16 single comparisons, testing if the field `cond` is equal to the values 0 through 15.

Fields in a pattern expression can be referenced as field names defined in a previous `token` declaration, or as a sub-field of a named token field. The attribute operators `?bit` and `?bits` can be used in pattern expressions to select a single bit or a range of bits from the named token field. These attributes are describe more on page 259. When used in a pattern expression the `?bit` attribute operator can only be called with a literal integer operand.

The `||` operator concatenates the OR-lists of its two operands to generate a new pattern in disjunctive normal form. `&&` returns the cross product of its operands, concatenating every AND-list from the left-hand side with ever AND-list from the right-hand side. The concatenation operator (`$`) behaves like the `&&` operator, but `$` also adds an offset to all the tokens in its right-hand side operand so they follow the left-hand side tokens in a matched token stream.

If a `pat` declaration is used to define multiple pattern names, then each pattern name is mapped to one element of the OR-list described by the pattern expression. The number of pattern names being defined must be the same as the number of elements in the OR-list described by the pattern expression, or it is an error. The wildcard pattern name (_) can be used to skip an element of the OR-list without associating it to a name.

To allow multiple pattern names to be defined with more than one OR'ed AND-list, previously declared pattern names are not expanded until after the new pattern names, listed in the current `pat` declaration, have been defined. Once defined, previously declared pattern names in the pattern expression are expanded, and the patterns associated with each new pattern name are re-normalized into disjunctive normal form.

global_stmt	→ `sem` *snames scope where_opt* `;`	instruction semantic declaration
snames	→ `[` *name_list* `]`	
	→ **name**	
name_list	→ *name_list* **name**	
	→ **name**	
scope	→ `{` *stmt_list* `}`	
where_opt	→ `where` *where_bind_list*	
	→	
where_bind_list	→ *where_bind_list* `,` *where_bind*	
	→ *where_bind*	
where_bind	→ **name** `=` *exp*	
	→ **name** `in [` *wexp_list* `]`	
wexp_list	→ *wexp_list wexp*	
	→ *wexp*	
wexp	→ *w_op*	
	→ **name**	
	→ *aexp*	

A `sem` declaration associates a list of previously declared pattern names with semantic code. These are the pattern names that correspond to instructions in a target ISA. Other pattern names, those without semantics, just describe miscellaneous conditions on token streams that do not nec-

essarily correspond to instructions. Note that token field names can be used in semantic code to reference the value of bit fields within the instruction, with the same restrictions as in pattern cases of a Facile switch statement (described on page 250).

Multiple pattern names (instructions) can be associated with semantic code using a single `sem` declaration. Each pattern name is bound to a different copy of the semantic code. An optional `where` clause parameterizes the semantic code, and these parameters can be bound to different expressions for each instruction being defined. The list of expressions associated with each parameter must have the same length as the list of instruction names being defined.

Although Facile usually forbids function pointers or pointers to operators, expressions in the list of values for parameters in a `where` clause are an exception. Elements in these lists can be Facile operators (*w_op* can be one of `!`, `%`, `&`, `&&`, `*`, `+`, `-`, `/`, `<<`, `>>`, `^`, `^^`, `|`, `||`, `~`, `<`, `<=`, `==`, `!=`, `>=`, `>`), function names, or an atomic expressions (*aexp* is defined on page 253). More complex expressions can be listed, but must be enclosed in parentheses, as per the definition of *aexp*.

Facile Types

Facile is a strongly typed language with a type system inspired by the programming language ML. As in ML, functions can be defined with incomplete—i.e., polymorphic—types. Polymorphic functions can be instantiated with different concrete types at different call sites. For example, the identity function (takes a single argument and returns it) has type `(float)` → `float` when called with a floating-point argument, and type `(ulong)` → `ulong` when applied to a 32-bit

unsigned integer[1]. Unlike ML, Facile also allows overloading. An overloaded function name can be bound to two or more function bodies, so long as the overloaded versions can be distinguished by the types of their parameters or return value.

Type expressions can optionally be associated with any semantic expression, variable declaration, or function parameter or return value. Normally no types need to be specified, and the Facile compiler infers complete type information from clues in the code. For example, literal constant `5` automatically has type `ulong`, and in the variable binding `val x = 5` the variable `x` also has type `ulong`. Occasionally though, Facile's type inference algorithm needs some help determining types and disambiguating overloaded functions and operators. When disambiguation is needed, types are specified by following an expression, declared variable name, or function parameter with a colon (`:`) and the appropriate type expression.

bind_stmt → `type` *type_param_list_opt* **name** *length_param_list_opt* `=` *type* `;`

type_param_list_opt → *type_param_list_opt* *type_param*
→

length_param_list_opt → `[` *lenght_param_list* `]`
→

length_param_list → *length_param_list* `,` *type_param*
→ *type_param*

type_param → **name**
→ `_`

1. Facile has no explicit function types. Where function types are need in this reference they are written as (*arg-type-list*) → *return-type* .

A `type` declaration defines a new type name. The type name can optionally be parameterized by one or more type parameters that stand for any type, or by length parameters that can get instantiated with literal integer values. An example of a `type` declaration without any parameters is `type cwp_t = unsigned[5]`, where the type `cwp_t` is defined to be the 5-bit unsigned integer type. An example of a parametrized type name declaration is `type T pair = (T,T)`, so the type `ulong pair` would be the same as type `(ulong,ulong)`.

Length parameters are useful in types built from arrays (to specify the array length) or from integer or floating point types (to specify the width). For example, to declare a type name for arrays of unsigned integers, the following declaration could be used:

```
type uarray[dim,width] = unsigned[width] array[dim];
```

The wildcard name (_) can be used in place of a type or length parameter that is not actually used in the type expression. For example, `type T my_queue[_] = T queue` defines `my_queue` so that it takes a length argument even though the length argument is not needed with Facile's built-in `queue` datatype.

type → *type_arg_list* **name** *type_lengths_opt*
→ *type_arg_list* `array` *type_lengths_opt*
→ *type_arg_list* `queue`
→ *atype*

atype	→ **name** *type_lengths_opt*	
	→ `struct {` *field_type_list_opt ellipsis_opt* `}`	structure type
	→ `(` *type* `,` *type_list* `)`	tuple type
	→ `(` *type* `)`	
	→ _	wildcard (polymorphic)
type_arg_list	→ *type_arg_list atype*	
	→ *atype*	
type_lengths_opt	→ `[` *type_length_list* `]`	
	→	
type_length_list	→ *type_length_list* `,` *type_length*	
	→ *type_length*	
type_length	→ **name**	
	→ **int**	
	→ _	

Several base types are predefined: The types `char`, `short`, `long`, and `llong` are all signed integer types with widths (in bits) 8, 16, 32, and 64 respectively. Types `bool`, `uchar`, `ushort`, `ulong`, and `ullong` are unsigned integers with widths (in bits) 1, 8, 16, 32, and 64 respectively. Types `float`, `double`, and `quad` are IEEE floating point types with widths (in bits) 32, 64, and 128 respectively. Other predefined type names are `void`, `string`, `stream` (token streams), `system` (the type of the system variable, described on page 254), `cc` (condition codes), and `elf` (ELF file descriptors[1]).

Parameterized type names are instantiated by listing the type arguments before the parameterized name and listing length arguments in square brackets after the type name. All type and length parameters of a parameterized type must be given values when the type is instantiated. Array

1. ELF file descriptors are defined in the system C header file <libelf.h>.

types are specified with the `array` reserved word, given one type argument (the element type) and one length argument (the array length). Multi-dimensional array types are written as arrays of arrays. Facile also supports a double-ended dynamic queue datatype, that grows and shrinks as needed at run-time. Queue types are specified using the `queue` reserved word preceded by one type argument specifying the queue element type. The queue type does not have any length arguments because a queue length can change at run-time.

There are two other predefined parameterized type names: The names `signed` and `unsigned` stand for signed and unsigned integers respectively, and each takes one length argument specifying the width (in bits) of the integer. Signed integers must have width 8, 16, 32, or 64. Unsigned integers can have any width from 1 to 64 inclusive.

The wildcard name (_) can be used to describe polymorphic types. The wildcard stands for an unknown type that will be instantiated later by type inference. For example, `unsigned[_]` is an unsigned type of any width.

atype	→ `struct {` *field_type_list_opt ellipsis_opt* `}`	structure type
	→ `(` *type* `,` *type_list* `)`	tuple type
field_type_list_opt	→ *field_type_list_opt* `,` *field_type*	
	→	
field_type	→ **name** `:` *type*	
type_list	→ *type_list* `,` *type*	
	→ *type*	

Array and queue types are examples of complex types. Other complex types can be constructed using Facile's tuple and structure type constructors. Tuple types are specified as comma separated lists of types in parentheses, with at least two elements in the list. Structures are specified using the `struct` reserved word followed by a list of field name/type pairs in curly brackets. Incomplete/polymorphic structure types are specified by including an ellipsis (`..`) at the end of the field list. Incomplete structure types are not the same as subtypes: Once an incomplete structure type is instantiated to be a complete type, the type is fixed and no subtyping is allowed.

Semantic Statements

Semantic statements include variable, function, and type name declarations, and a variety of control flow statements. Semantic declarations can be used in any scope—global or local—and define a name that can be used anywhere after the declaration statement within the current scope or in any subsequent sub-scope. Other statements are only allowed within a function body or in instruction semantic code. Any expression (described in the next section) can also be written as its own statement to ignore the expression's result value. Multiple statements can be grouped within curly brackets (`{}`), which delimit a new sub-scope. Variable names declared with a sub-scope or in any enclosing scope out to and including the global scope can be accessed.

global_stmt → *bind_stmt*

bind_stmt → `type` *type_arg_list_opt* **name** *length_arg_list_opt* `=` *type* `;`
→ *inline_opt* `fun` **name** `(` *arg_list_opt* `)` *type_restrict_opt scope*
→ *const_opt* `val` **name** *type_restrict_opt bind_exp_opt* `;`
→ `var` **name** *type_restrict_opt* `=` *exp* `;` reference variable

arg_list_opt → *arg_list* `,` *arg*
→

arg → `var` **name** *type_restrict_opt*
→ **name** *type_restrict_opt*

scope → `{` *stmt_list* `}`

stmt_list → *stmt_list* *stmt*
→ *stmt*

bind_exp_opt → `=` *exp*
→

inline_opt → `inline`
→

const_opt → `const`
→

type_restrict_opt → `:` *type*
→

Binding statements can appear in both the global scope and in local scopes, i.e., inside a function body or in the semantic code of a `sem` declaration. Type binding declarations are described on page 242 in the section on Facile types. The remaining binding statements define function and variable names. A `fun` declaration defines a function, a `val` declaration defines a variable, and a `var` declaration defines a reference to any lvalue[1]. Although functions and variable names must be declared before they are used, specifying types is optional if the type can be derived from the context in which the function or variable is used.

1. As in C, an lvalue is an expression that identifies a particular storage location. For example, an array lookup `A[5]` is an lvalue, but a computed value `x+1` is not.

A function binding includes a function name, a list of parameter names, and a function body. Function parameters can be passed by value or passed by reference. Reference parameters are preceded with the `var` reserved word. An optional type specification is allowed after each function parameter to specify the parameter's type, and between the parameter list and the function body to specify the function return type. If a function declaration is preceded by the `inline` reserved word, then all calls to the function are inlined. Regardless of the presence or absence of the `inline` flag, all calls to local functions (i.e., functions declared within a scope other than the global scope) are inlined.

Non-reference variables are defined using `val` declarations. Once declared, a variable can be used in subsequent statements of the same scope or sub-scopes. An optional initial value can be given in the declaration. A `val` declaration preceded by the `const` reserved word cannot be changed by subsequent assignments. Constant variables must have an initial value given in their declaration. Reference variables are defined using `var` declarations, and are interpreted as an alias to the value they are a reference for. A reference variable must be initialized with an lvalue expression.

stmt → *bind_stmt*
→ `if (` *exp* `)` *stmt*
→ `if (` *exp* `)` *stmt* `else` *stmt*
→ `while (` *exp* `)` *stmt*
→ `for (` **name** *type_restrict_opt* `in` *exp* `)` *scope*
→ `switch (` *exp* `) {` *case_list* `}`
→ `continue` *int_opt* `;`
→ `break` *int_opt* `;`
→ `return` *exp_opt* `;`
→ *exp* `;`
→ *scope*
→ `;`

int_opt → **int**
→

exp_opt → *exp*
→

Facile's control flow statements are `if`, `while`, `for`, `switch`, `continue`, `break`, and `return`. An `if` statement switches between two possible control flow paths based on the result of an expression with type `bool`. The `else` clause is optional. A `while` statements test a conditional expression of type `bool`, and loops until the condition evaluates to `false`. A `for` loop iterates through the elements of an array or queue. The variable named in a `for` loop is only defined within the associated scope, and it refers to the elements of the given array or queue, one element per iteration in ascending index order.

A `continue` statement causes the control flow to jump to the next iteration of an enclosing `while` or `for` loop. A `break` statement jumps to just after an enclosing loop. An optional integer literal can follow a `continue` or `break`, specifying which enclosing loop to continue or break from. A value of 1 specifies the innermost loop, 2 is the second innermost loop, etc. If omit-

ted, the default is to continue or break from the innermost loop. A `return` statement returns from the current function. If the function return type is anything other than `void`, then a return value must be given.

An expression can be written as a statement. This is useful for expressions with a `void` result type (e.g., assignment expressions and function calls with no return value). A statement can also be empty (just a `;` with no other code), or multiple statements grouped within curly brackets (`{}`). Every group of statements within curly brackets is a sub-scope. All variable, function, and type names defined in enclosing scopes can be used in a sub-scope provided they are not masked by other variable, function, or type names. Note that function and variable names can be overloaded, so a function name may not get masked by later bindings, unless the type of the new binding masks the type of bindings for the same function name in enclosing scopes.

stmt	→ `switch (` *exp* `) {` *case_list* `}`	
case_list	→ *case_list case_clause*	
	→ *case_clause*	
case_clause	→ `case` *match1* `:` *stmt_list*	normal case
	→ `pat` *pat_exp* `:` *stmt_list*	pattern case
	→ `default :` *stmt_list*	default case
match	→ *match* `:` *type*	type specification
	→ **name** `as` *match*	declare alias name
	→ `pat` *pat_exp*	pattern sub-case
	→ *match1*	

match1	→ **name**	declare variable or reference
	→ *int_match*	match an integer
	→ `(` *match* `,` *match_list* `)`	match a tuple
	→ `{` *match_field_list* *ellipsis_opt* `}`	match a record
	→ `(` *match* `)`	grouping
	→ `_`	wildcard
int_match	→ `-` **int**	negative signed integer
	→ `+` **int**	positive signed integer
	→ **int**	unsigned integer
match_list	→ *match_list* `,` *match*	
	→ *match*	
match_field_list	→ *match_field_list* `,` *match_field*	
	→ *match_field*	
match_field	→ `name =` *match*	
ellipsis_opt	→ `..`	
	→	

Facile `switch` statements are similar to case statement in ML, because they can extract individual elements of complex data structures and bind them to variable names, in addition to selecting between multiple cases based on data values. A Facile `switch` statement can also select between multiple patterns in a token stream. For example, instructions in an ISA could be decoded using a `switch` statement with pattern cases, and the Facile compiler internally transforms instructions defined with `pat` and `sem` declarations into a Facile `switch`.

Following the `switch` reserved word and the condition expression is a list of case clauses. Logically, a `switch` statement is evaluated by first evaluating its condition expression, then stepping through the list of case clauses in program order until one of the clauses matches the value of

the condition. Then the statements associated with the matching case clause are evaluated. The statements associated with a case clause do not fall through to the next case, as in C.

A case clause with the `case` reserved word is used to match data with types other than `stream` (the token stream type). These clauses can match simple signed and unsigned integer values, or deconstruct complex types to match the values of one or more fields. If variable names are given in a case clause's match expression and the case clause is matched when evaluating the switch statement, then the named variables are bound to their corresponding component of the condition expression. These variable bindings are only valid within the scope of the case clause. If the condition expression is an lvalue, then variable names are bound as reference variables, otherwise the variable names are bound to a copy of the selected data. The wildcard name (_) can be used in place of a variable name to match any value without binding it to a name.

A case clause with the `pat` reserved word matches data of type `stream`. The pattern expressions used here are the same as the pattern expressions used in `pat` declarations, described on page 237. The statements associated with a pattern case can use token field names to access bit fields within the matched token stream. Token field names are defined for the case clauses statements if the token is in the matched pattern and the field name uniquely identifies a sequence of bits in the token stream. Field names for token not in a matched pattern and field names that cannot be uniquely determined are not defined. Patterns in a token stream can also be matched as part of the match expression in a case clause with the `case` reserved word, but no field names will be defined.

A case clause with the `default` reserved word is the same as a case clause with the `case` reserved word and a wildcard (_) match expression (i.e., `case _ :` *stmt-list*). Case clauses following the default case are never matched. In general, case clauses are tested in the same order as they appear in the switch statement. Earlier case clauses may mask later case clauses if every value that would match in a later case is matched by earlier cases.

Semantic Expressions

Adding two values, calling a function, and assigning to a variable are all examples of expressions. All expressions that have type other than `void` produce one result value. `void` typed expressions have no result value.

exp	→ *exp* `:` *type*	type restriction
	→ *exp* *op2* *exp*	binary operator
	→ *aexp1*	
aexp1	→ *op1* *aexp1*	unary operator
	→ **name** `(` *exp_list_opt* `)`	function call
	→ *aexp1* `?` **name** *arg_list_opt*	attribute
	→ *aexp*	
aexp	→ **name**	variable name
	→ **int**	literal unsigned integer
	→ **float**	literal 64-bit float
	→ **string**	literal string
	→ `array` *length_arg_opt* `{` *exp_list* `}`	array
	→ `queue` *length_arg_opt* `{` *exp_list_opt* `}`	queue
	→ `struct {` *struct_field_list* `}`	structure
	→ `(` *exp* `,` *exp_list* `)`	tuple
	→ *aexp* `[` *exp* `]`	array/queue lookup
	→ *aexp* `.` **name**	record/tuple field selection
	→ `(` *exp* `)`	grouping

arg_list_opt → `(` *exp_list_opt* `)`
→

exp_list_opt → *exp_list*
→

exp_list → *exp_list* `,` *exp*
→ *exp*

length_arg_opt → `(` *exp* `)`
→

struct_field_list → *struct_field_list* `,` *struct_field*
→ *struct_field*

struct_field → **name** `=` *exp*

At the leaves of an expression tree are variable names and literal integer, floating-point, and string values. Note that literal integers are all unsigned and literal floating-point values are all 64-bits wide (type `double`). Literal values can be cast or converted to other types as needed. The names `true` and `false` are predefined to be the constant 1-bit unsigned integers 1 and 0 respectively. The name `system` is predefined as a value of type `system`, and provides access to various system related attributes, including getting/setting memory values and accessing the raw ELF file descriptor for a target executable.

An un-named array or queue value is constructed using the `array` or `queue` reserved word respectively, followed by an optional length argument and an initializer list. When constructing an array, the array length (if omitted) is inferred from the length of the initializer list. If an array length is given and the initializer list has exactly one element, then the array is initialized with a copy of the initializer value in every array element. Otherwise the array length and the initializer

list length must be the same. When constructing a queue, the length argument is only a hint to the compiler, and a queue's initial length is determined entirely from the length of its initializer list. The length argument in a queue constructor should be the expected maximum number of elements that will be stored in the queue, so Facile's compiler can generate efficient queue allocation code. Both array and queue elements are accessed by following an expression that evaluates to an array or queue with an index enclosed in square brackets. In array lookup expressions, the index value is an unsigned integer. In queue lookup expressions, the index value is a signed integer, and negative values index backward from the back end of the queue (e.g., `Q[-1]` is the last element in queue `Q`).

Structures are constructed with the `struct` reserved word followed by a list of field name/value pairs enclosed in curly brackets (`{}`). Unlike structure types and structure matching expressions in `switch` statements, structure values must be fully defined (i.e., no ellipsis). Tuple values are constructed by enclosing a list of two or more expressions in parentheses. Structure fields are accessed by following an expression that evaluates to a structure with a period (`.`) and a field name. Tuples fields are access in the same way but an unsigned integer from 1 to the tuple length is used, since tuples have no field names (e.g., `(10,20).1` evaluates to `10`).

Facile has several infix binary and prefix unary operators. Unary operators are `!` (boolean not), `~` (bit inversion), `-` (negation), and + (unsigned to signed type casting). The `!` operator can only be applied to values of type `bool` and returns `bool` values. The `~` operator can be applied to any unsigned integer and returns a value of the same type. The unary `-` operator can be applied to

any signed integer, unsigned integer, or floating-point value and returns a value with the same width. Negating an unsigned integer produces a signed integer of the same width. The unary + operator can only be applied to unsigned integers and casts its operand to a signed value with the same bit-width. Facile also pre-defines the function `sqrt` (called as a function, not as an operator) that takes one floating point argument and returns its square root.

The following table lists all of Facile's boolean operators, and their (possibly overloaded) types. These operators are listed in order of increasing precedence, with operators listed on the same line having the same precedence. All binary operators are left associative.

Operators	**Type(s)**
`=`	(α, α) → `void`
`&& || ^^`	(`bool`, `bool`) → `bool`
`< <= == != >= >`	(`unsigned[`α`]`, `unsigned[`α`]`) → `bool` (`signed[`α`]`, `signed[`α`]`) → `bool` (*fp*`[`α`]`, *fp*`[`α`]`) → `bool`[a]
`+ -`	(`stream`, `unsigned[`α`]`) → `stream` (`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]` (`signed[`α`]`, `signed[`α`]`) → `signed[`α`]` (*fp*`[`α`]`, *fp*`[`α`]`) → *fp*`[`α`]`
`* / %`	(`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]` (`signed[`α`]`, `signed[`α`]`) → `signed[`α`]` (*fp*`[`α`]`, *fp*`[`α`]`) → *fp*`[`α`]` (`*` and `/` only)
`& | ^`	(`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]`
`<< >>`	(`unsigned[`α`]`, `unsigned[_]`) → `unsigned[`α`]` (`signed[`α`]`, `unsigned[_]`) → `signed[`α`]` (`>>` only)

a. The name *fp* is used to describe a floating-point types with a width parameter. A special name is needed here, since no such name is predefined in Facile.

In addition to binary and unary operators, Facile also has attributes and attribute calls. Attributes follow an expression and are written as `?` and the attribute name. Attribute calls also have a list of arguments in parentheses. Attributes perform special operations that either support Facile's built-in data types (e.g., queues and condition codes) or require non-standard syntax (e.g., type casting and bit selection). Facile makes no distinction between attributes and attribute calls with no arguments. For example `PC?exec` is the same as `PC?exec()`. Below is a comprehensive list of all the attributes and attribute calls recognized by Facile:

- *exp*`?length` returns the length of an array or queue.

- *exp*`?width` returns the bit-width of a signed, unsigned, or floating-point value.

- *exp*`?start_pc` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the start address of the target executable.

- *exp*`?start_sp` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the initial stack pointer for the target executable.

- *exp*`?elf` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the ELF file descriptor for the target executable.

- *exp*`?exec()` decodes the first instruction in the given token stream and evaluates its semantic code. Instruction patterns and semantics are declared using `pat` and `sem` declarations. All `sem` declarations must occur before the first call to `?exec` in a Facile source file.

- *exp*`?addr` returns the target byte address of the beginning of a token stream.

- *exp*`?cast(`*type*`)` changes the type of a signed integer, unsigned integer, or floating-point value to a signed integer, unsigned integer, or floating point type with the same width. The bits of a value are not changed, only its type.

- *exp*`?cvt(`*type*`)` converts a value of any signed integer, unsigned integer, or floating-point type to a value of any other signed integer, unsigned integer, or floating point type. Note that the bit representation may be changed, although the represented value is preserved up to the resolution of the new representation type.

- *exp*`?ext(`**int**`)` extends a signed or unsigned integer to the bit-width specified in the attribute argument by padding the most significant bits with 0s. The given width must be equal to or greater than the width of the integer being extended. The extended result is unsigned.

- *exp*`?sext(`**int**`)` sign extends a signed or unsigned integer to the bit-width specified in the attribute argument. The sign of the value being extended is inferred from its most significant bit (as in a 2s complement representation), even if the value has an unsigned type. The given

width must be equal to or greater than the width of the integer being extended. The extended result is unsigned.

- *exp*`?memory(`**int**`,`*exp*`)` applies to expressions of type `system` (normally only the pre-defined variable `system`) and provides access to a target executable's memory. The first attribute argument is a literal integer specifying the access width in bytes[1], and the second argument is the target virtual address. Note that the result of a `?memory` attribute call is an lvalue, so memory values can be both read and written.[2]

- `(`*exp op2 exp*`)?cc(`*exp*`)` computes the condition codes associated with a boolean operator expression. The normal expression value is returned, but condition codes are stored into the attribute call argument as a side effect. The attribute call argument must be an lvalue of type `cc`. Facile can compute condition codes for the floating-point subtraction operator (`-`) and the following integer operators: `+`, `-`, `&&`, `||`, `^^`, `&`, `|`, and `^`.

- *exp*`?bit(`*exp*`)` selects one bit from any signed, unsigned, floating-point, or condition code value. Which bit is selected is specified by the attribute call argument.

- *exp*`?bits(`**int**`,`**int**`)` and *exp*`?bits(`**int**`)` select one or more bits from any signed, unsigned, floating-point, or condition code value. The first version of this attribute call is called with the

1. FastSim's current host processor is a SPARC, so unaligned memory accesses produce a bus error signal (SIGBUS).
2. Some target memory is not mapped or is mapped read-only, as it would be in a real execution outside the simulator. Attempting any access to an address that is not mapped, or assigning to read-only memory produce a segmentation violation signal (SIGSEG).

index of the first and last bits to select. The second version selects a given number of bits starting from bit 0, the least significant bit. The attribute arguments are integer literals because the number of bits selected must be known by the compiler to determine the result type. Results are unsigned.

- *exp*`?push_back(`*exp*`)` and *exp*`?push_back()` push one new element onto the back end of a queue. If an attribute argument is given, then it is used to initialize the new queue element. This attribute call has no return value (return type `void`).

- *exp*`?push_front(`*exp*`)` and *exp*`?push_front()` push one new element onto the front end of a queue. If an attribute argument is given, then it is used to initialize the new queue element. This attribute call has no return value (return type `void`).

- *exp*`?pop_back(`*exp*`)` and *exp*`?pop_back()` pop element(s) off the back end of a queue. The first version takes an unsigned integer as its attribute call argument, and pops the given number of entries off the queue. This version has no return value (return type `void`). The second version pops one element and returns it. Attempting to pop elements from an empty queue produces a run-time error.

- *exp*`?pop_front(`*exp*`)` and *exp*`?pop_front()` pop element(s) off the front end of a queue. The first version takes an unsigned integer as its attribute call argument, and pops the given number of entries off the queue. This version has no return value (return type `void`).

The second version pops one element and returns it. Attempting to pop elements from an empty queue produces a run-time error.

- *exp*`?clear()` removes all elements from a queue. No value is returned (return type `void`).

- *exp*`?static` causes a fast-forwarding simulator to make the attributed expression result value run-time static. If the attributed value is an lvalue, then the `?static` attribute also makes the lvalue storage location run-time static. Note that, if a value is already static or run-time static, or if the simulator is not compiled with the fast-forwarding optimization, then this attribute is a no-op.

Facile pre-defines the function `assert`. `assert` takes a single argument of type bool. If the argument evaluates to false at run-time, then the simulator exits with an assertion failed error.

External Code

Facile code can access variables and call functions written in other languages, or just linked from separate object files. Conversely, external code can access facile variables and call facile functions. A Facile file's external interface is specified using external variable, function, and type declarations.

bind_stmt	→ `extern` **name** `:` *type* `;`	external variable
	→ `extern` **name** `(` *arg_type_list_opt* `)` `:` *type* `;`	external function
	→ `extern type` **name** *type_assign_opt* `;`	external type

arg_type_list_opt → *arg_type_list* `,` *targ*
→

targ → `var` *type*
→ *type*

type_assign_opt → `=` *type*
→

Variable and function names that either refer to symbols in another file or provide external access to values and code in a Facile file must be declared with an external variable or function declaration respectively. These declarations define the type of an external variable or the types of an external function's parameters and return value. To export a global variable or function defined in a Facile file, first declare the name with one of these external declaration, then bind the name with a `var` or `fun` declaration. Note that only global variables and functions can be exported. External function names cannot be overloaded or polymorphic.

External type declarations either provide a name for an external pointer type that can be stored in Facile variables but cannot be not dereferenced, or a Facile type that may be used by external code. To declare a type name for an external pointer type, declare an extern type name but omit the optional type assignment. Values of this type can then be stored in Facile variables, but they cannot be dereferenced and are always treated as dynamic values when fast-forwarding. To export a Facile type, declare an external type name and assign it to a Facile type. Exported Facile types are put in a C header file generated by the Facile compiler, and can be `#include`'ed into other C source files.

Simulator Layout

A Facile simulator must have a certain structure to work in the FastSim run-time environment. Primarily, a simulator must define a global variable called `init` and a global function called `main`. The `main` function is called repeatedly by the FastSim run-time environment. Each time `main` returns, FastSim calls it again. `main` must have one or more parameters, with any types except named external pointer types (since they cannot be memoized). The value of `main`'s arguments are used to index the memoization cache when fast-forwarding.

The argument values for each call to `main` are stored in `init`. Typically each call to `main` will update the `init` variable to set up arguments for the next call to `main`. If `main` has one argument, then `init` has the same type as `main`'s one argument. If `main` has multiple arguments, then `init` is a tuple, where each tuple field corresponds to one argument.

Call-by-reference parameters (declared with the `var` reserved word) can be used in the declaration of `main`. A reference parameter refers to the corresponding tuple field in the `init` variable. Hence the value in `init` can be modified by either assigning directly to `init` or by assigning to one of `main`'s reference parameters.

APPENDIX B: A COMPLETE SIMULATOR IN FACILE

A simple simulator for the SPARC-V9 instruction set is expressed in 701 lines of Facile code. A complete listing of this simulator is given below. This example is intended to provide a context for better understanding the Facile code examples used in this dissertation.

This simulator consists of four files: `simple.fs` (top-level simulation code), `sparc_v9_enc.fs` (instruction encodings), `sparc_v9_reg.fs` (architectural registers), and `sparc_v9.fs` (instruction semantics).

```
/************************************************************************
** FILE: simple.fs
** Basic sparc istruction simulator (emulation only).
*/

#include "sparc_v9.fs"

fun initialize()
{
    R4(14,system?start_sp);                              // set initial $sp
    return (system?start_pc, system?start_pc + 4);       // initiail $pc, $npc
}

// Init is the source of arguments for main. Declare and set its initial value.
val init = initialize();

// Main simulator function
fun main(pc, npc)
{
    // Copy arguments to global variables
    PC = pc; nPC = npc;

    nPC2 = nPC + 4;     // default next nPC (may get reset by branch insts)
    PC?exec();          // execute 1 instruction

    // Put next args-to-main in init
    init = (nPC, nPC2);
}
```

```
/*****************************************************************************
** FILE: sparc_v9_enc.fs
** Facile description file for the SPARC v.9 instruction encoding.
*/

//
// Declare token "instruction" and various token fields.

token instruction[32] fields
    op 30:31, op2 22:24, op3 19:24, opf 5:13, rd 25:29, rs1 14:18, rs2 0:4,
    opf_cc 11:13, opf_low 5:10, cond 25:28, mcond 14:17, rcond 10:12,
    disp30 0:29, disp22 0:21, disp19 0:18, d16hi 20:21, d16lo 0:13,
    imm22 0:21, simm13 0:12, simm11 0:10, simm10 0:9, imm_asi 5:12,
    shcnt32 0:4, shcnt64 0:5, sw_trap 0:6,
    movcc2 18:18, movccr 11:12, movcc1 12:12, movcc0 11:11,
    bpccr 20:21, bpcc1 21:21, bpcc0 20:20,
    a 29:29, p 19:19, i 13:13, x 12:12;

//
// Here are some patterns that will be useful in defining instructions later.

pat reg_imm13 = (i==0 && instruction?bits(5,12)==0) || i==1;
pat reg_imm11 = (i==0 && instruction?bits(5,10)==0) || i==1;
pat reg_imm10 = (i==0 && instruction?bits(5,9)==0) || i==1;

pat reg_shcnt64 =   (i==0 && instruction?bits(5,11)==0) ||
                    (i==1 && instruction?bits(6,11)==0);

pat reg_trap =  (i==0 && instruction?bits(5,10)==0) ||
                (i==1 && instruction?bits(7,10)==0);

pat reg_imm13_inv = (i==1 && instruction?bits(5,12)==0) || i==0;

//
// Op0 instructions:

pat [   _trap   _bpcc   bicc    bpr     sethi   fbpfcc  fbfcc   _       ]
    = op==0 && op2 in [0..7];

pat illtrap = _trap && rd==0;

pat [   _       brz     brlez   brlz    _       brnz    brgz    brgez   ]
    = bpr && cond in [0x0..0x7];

pat [   fbn     fbne    fblg    fbul    fbl     fbug    fbg     fbu
        fba     fbe     fbue    fbge    fbuge   fble    fbule   fbo     ]
    = fbfcc && cond in [0x0..0xf];

pat [   fbpn    fbpne   fbplg   fbpul   fbpl    fbpug   fbpg    fbpu
        fbpa    fbpe    fbpue   fbpge   fbpuge  fbple   fbpule  fbpo    ]
    = fbpfcc && cond in [0x0..0xf];
```

```
pat [   bn      be      ble     bl      bleu    bcs     bneg    bvs
        ba      bne     bg      bge     bgu     bcc     bpos    bvc     ]
    = bicc && cond in [0x0..0xf];

pat [   bpn     bpe     bple    bpl     bpleu   bpcs    bpneg   bpvs
        bpa     bpne    bpg     bpge    bpgu    bpcc    bppos   bpvc    ]
    = _bpcc && cond in [0x0..0xf] && bpcc0==0;

//
// Op2 instruction:

pat [   add     and     or      xor     sub     andn    orn     xnor
        addc    mulx    umul    smul    subc    udivx   udiv    sdiv
        addcc   andcc   orcc    xorcc   subcc   andncc  orncc   xnorcc
        addccc  _       umulcc  smulcc  subccc  _       udivcc  sdivcc
        taddcc  tsubcc  taddcctv tsubcctv mulscc _      _       _
        _       _       _       _       _       sdivx   _popc   _
        wr      _       _       _       _       _       _       _
        jmpl    _retrn  _       _flush  save    restore _       _       ]
    = op==2 && op3 in [0x0..0x3f] && reg_imm13;

pat flush = _flush && rd==0;
pat flushw = op==2 && op3==0x2b && rd==0 && instruction?bits(0,18)==0;

pat impdep1 = op==2 && op3==0x36;
pat impdep2 = op==2 && op3==0x37;

pat [   movfn   movfne  movflg  movful  movfl   movfug  movfg   movfu
        movfa   movfe   movfue  movfge  movfuge movfle  movfule movfo   ]
    = op==2 && op3==0x2c && mcond in [0x0..0xf] && i==0 && reg_imm11;

pat [   movn    move    movle   movl    movleu  movcs   movneg  movvs
        mova    movne   movg    movge   movgu   movcc   movpos  movvc   ]
    = op==2 && op3==0x2c && mcond in [0x0..0xf] && i==1 && reg_imm11;

pat [   _       movrz   movrlez movrlz  _       movrnz  movrgz  movrgez ]
    = op==2 && op3==0x2f && rcond in [0..7] && reg_imm10;

pat popc = _popc && rs1==0;

pat rd = op==2 && op3==0x28 && instruction?bits(0,13)==0;

pat retrn = _retrn && rd==0;

pat [ sll srl sra ]
    = op==2 && op3 in [0x25..0x27] && x==0 && instruction?bits(5,11)==0;
pat [ sllx srlx srax ] = op==2 && op3 in [0x25..0x27] && x==1 && reg_shcnt64;

pat stbar = op==2 && op3==0x28 && rd==0 && rs1==0x0f &&
            instruction?bits(0,13)==0;
```

```
pat [   tn      te      tle     tl      tleu    tcs     tneg    tvs
        ta      tne     tg      tge     tgu     tcc     tpos    tvc     ]
    = op==2 && op3==0x3a && cond in [0x0..0xf] && a==0 && reg_trap;

//
// Floating-point instructions (also Op2):

pat fpop1 = op==2 && op3==0b110100;
pat fpop2 = op==2 && op3==0b110101;

pat [   _       fadds   faddd   faddq   _       fsubs   fsubd   fsubq   ]
    = fpop1 && opf in [0x40..0x47];

pat [   _       fcmps   fcmpd   fcmpq   _       fcmpes  fcmped  fcmpeq  ]
    = fpop2 && opf in [0x50..0x57] && instruction?bits(27,29)==0;

pat [   fstox   fdtox   fqtox   fstoi   fdtoi   fqtoi
        fstod   fstoq   fdtos   fdtoq   fqtos   fqtod
        fxtos   fxtod   fxtoq   fitos   fitod   fitoq   ]
    = fpop1 && rs1==0 && opf in
    [   0x81    0x82    0x83    0xd1    0xd2    0xd3
        0xc9    0xcd    0xc6    0xce    0xc7    0xcb
        0x84    0x88    0x8c    0xc4    0xc8    0xcc    ];

pat [   _       fmovs   fmovd   fmovq
        _       fnegs   fnegd   fnegq
        _       fabss   fabsd   fabsq   ]
    = fpop1 && opf in [0x0..0xb] && rs1==0;

pat [ fmuls fmuld fmulq fsmuld fdmulq fdivs fdivd fdivq ]
    = fpop1 && opf in [0x49 0x4a 0x4b 0x69 0x6e 0x4d 0x4e 0x4f];

pat [   _       fsqrts  fsqrtd  fsqrtq  ]
    = fpop1 && opf in [0x28..0x2b] && rs1==0;

pat [ fmovfsn fmovfsne fmovfslg fmovfsul fmovfsl fmovfsug fmovfsg fmovfsu
      fmovfsa fmovfse fmovfsue fmovfsge fmovfsuge fmovfsle fmovfsule fmovfso
      fmovfdn fmovfdne fmovfdlg fmovfdul fmovfdl fmovfdug fmovfdg fmovfdu
      fmovfda fmovfde fmovfdue fmovfdge fmovfduge fmovfdle fmovfdule fmovfdo
      fmovfqn fmovfqne fmovfqlg fmovfqul fmovfql fmovfqug fmovfqg fmovfqu
      fmovfqa fmovfqe fmovfque fmovfqge fmovfquge fmovfqle fmovfqule fmovfqo
    ] = fpop2 && opf_low in [1 2 3] && rs1 in [0x0..0xf] && i==0;

pat [   movsn   fmovse  fmovsle fmovsl  fmovsleu fmovscs fmovsneg fmovsvs
        fmovsa  fmovsne fmovsg  fmovsge fmovsgu fmovscc fmovspos fmovsvc
        fmovdn  fmovde  fmovdle fmovdl  fmovdleu fmovdcs fmovdneg fmovdvs
        fmovda  fmovdne fmovdg  fmovdge fmovdgu fmovdcc fmovdpos fmovdvc
        fmovqn  fmovqe  fmovqle fmovql  fmovqleu fmovqcs fmovqneg fmovqvs
        fmovqa  fmovqne fmovqg  fmovqge fmovqgu fmovqcc fmovqpos fmovqvc ]
    = fpop2 && opf_low in [1 2 3] && rs1 in [0x0..0xf] && i==1 && movcc0==0;
```

```
pat [   _       fmovrsz fmovrslez fmovrslz _ fmovrsnz fmovrsgz fmovrsgez
        _       fmovrdz fmovrdlez fmovrdlz _ fmovrdnz fmovrdgz fmovrdgez
        _       fmovrqz fmovrqlez fmovrqlz _ fmovrqnz fmovrqgz fmovrqgez ]
    = fpop2 && opf_low in [0x5..0x7] && rcond in [0..7] && i==0;

//
// Op1 instruction: CALL

pat call = op==1;

//
// Op3 instructions:

pat [ casa casxa ] = op==3 && op3 in [0x3c 0x3e] && reg_imm13_inv;

pat [   ldf     _ldfsr  _ldqf   lddf    stf     _stfsr  _stqf   stdf
        _       _       _       _       _       prefetch _      _       ]
    = op==3 && op3 in [0x20..0x2f] && reg_imm13;

pat [ ldfsr ldxfsr ] = _ldfsr && rd in [0 1];
pat [ stfsr stxfsr ] = _stfsr && rd in [0 1];
pat ldqf = _ldqf && rd?bit(1)==0;
pat stqf = _stqf && rd?bit(1)==0;

pat [   ldfa    _       _ldqfa  lddfa   stfa    _       _stqfa  stdfa
        _       _       _       _       _       prefetcha _     _       ]
    = op==3 && op3 in [0x30..0x3f];

pat ldqfa = _ldqfa && rd?bit(0)==0;
pat stqfa = _stqfa && rd?bit(0)==0;

pat [   lduw    ldub    lduh    _ldd    stw     stb     sth     _std
        ldsw    ldsb    ldsh    ldx     _       ldstub  stx     swap    ]
    = op==3 && op3 in [0x00..0x0f] && reg_imm13;

pat ldd = _ldd && rd?bit(0)==0;
pat std = _std && rd?bit(0)==0;

pat [   lduwa   lduba   lduha   _ldda   stwa    stba    stha    _stda
        ldswa   ldsba   ldsha   ldxa    _       ldstuba stxa    swapa   ]
    = op==3 && op3 in [0x00..0x0f];

pat ldda = _ldda && rd?bit(0)==0;
pat stda = _stda && rd?bit(0)==0;

//
// UltraSPARC specific

pat alignaddress = impdep1 && opf==0x18;
```

```
/*****************************************************************************
** FILE: sparc_v9_reg.fs
** Facile description file for the SPARC v.9 registers.
*/

#define NWINDOWS 8

val PC = 0?cvt(stream);     // program counter
val nPC = 0?cvt(stream);    // next program counter
val nPC2 = 0?cvt(stream);   // next next program counter

val CCR = 0?cvt(cc);                // integer condition codes
val fcc = array(4) { 0?cvt(cc) };   // FP condition codes

val Y : ulong = 0;      // Y register for multiply & divide

// Floating status register (implemented externally)
extern get_FSR4() : unsigned[32];
extern get_FSR8() : unsigned[64];
extern set_FSR4(unsigned[32]) : void;
extern set_FSR8(unsigned[64]) : void;

//
// integer registers:

val global_registers = array(7) { 0?ext(64) };
val register_windows = array(NWINDOWS) { array(16) { 0?ext(64) } };

type cwp_t = unsigned[5];

val CWP         = 0?bits(5);    // current window pointer
val CANSAVE     = (NWINDOWS-2)?bits(5);
val CANRESTORE  = 0?bits(5);
val OTHERWIN    = 0?bits(5);
val CLEANWIN    = (NWINDOWS-1)?bits(5);

fun Rx(i0) {        // get 64-bit register value
    val ii = i0?ext(32);
    if(ii == 0) return 0?ext(64);
    else if(ii < 8) return global_registers[ii-1];
    else {
        val win = (CWP?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); return register_windows[win][ii];
    }
}

fun Rx(i0,vv) {     // set 64-bit register value
    val ii = i0?ext(32);
    if(ii >= 8) {
        val win = (CWP?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); register_windows[win][ii] = vv;
```

```
    } else if(ii > 0) {
        global_registers[ii-1] = vv;
    }
}

fun R4(ii) { return Rx(ii)?bits(32); }  // get 32-bit register value
fun R4(ii,vv) { Rx(ii, vv?sext(64)); }  // set 32-bit register value

fun R8(i0) {    // make a 64-bit value from two 32-bit register
    val ii = i0?ext(32);
    val xx = (Rx(ii) & 0xffffffff?ext(64)) << 32;
    return xx | (Rx(ii+1) & 0xffffffff?ext(64));
}

fun R8(i0,vv) { // set two 32-bit registers from a 64 bit value
    val ii = i0?ext(32);
    Rx(ii, vv >> 32);
    Rx(ii+1, vv & 0xffffffff?ext(64));
}

//
// floating-point registers:

val fregs = array(64) { 0?ext(32) };

fun F4(ii) { return fregs[ii]?cast(float); }
fun F4(ii,vv) { fregs[ii] = vv?cast(unsigned[_])?cvt(ulong); }

fun F8(i0) {    // get concatination of two 32-bit fp-registers
    val ii = i0?ext(32)&(~1) | (i0?bit(0)?ext(32)<<5);
    val xx = fregs[ii]?ext(64) << 32;
    return (xx | fregs[ii+1]?ext(64))?cast(double);
}

fun F8(i0,vv) { // set two 32-bit fp-registers
    val ii = i0?ext(32)&(~1) | (i0?bit(0)?ext(32)<<5);
    fregs[ii] = vv?cast(unsigned[64])?bits(32,63);
    fregs[ii+1] = vv?cast(unsigned[64])?bits(32);
}

//
// Memory access:

fun M1(a) { return system?memory(1,a)?ext(64); }
fun M1s(a) { return system?memory(1,a)?sext(64); }
fun M1(a,vv) { system?memory(1,a) = vv?cast(unsigned[_])?bits(8); }

fun M2(a) { return system?memory(2,a)?ext(64); }
fun M2s(a) { return system?memory(2,a)?sext(64); }
fun M2(a,vv) { system?memory(2,a) = vv?cast(unsigned[_])?bits(16); }
```

```
fun M4(a) { return system?memory(4,a)?ext(64); }
fun M4s(a) { return system?memory(4,a)?sext(64); }
fun M4(a,vv) { system?memory(4,a) = vv?cast(unsigned[_])?bits(32); }

fun M8(a) { return system?memory(8,a); }
fun M8(a,vv) { system?memory(8,a) = vv?cast(unsigned[64]); }

////////////////////////////////////////////////////////////////////////
// External functions providing access to Facile data structures.
//

extern get_CCR() : cc;
extern set_CCR(cc) : void;
fun get_CCR() : cc { return CCR; }
fun set_CCR(CCR1 : cc) : void { CCR = CCR1; }

extern get_fcc(ulong) : cc;
extern set_fcc(ulong,cc) : void;
fun get_fcc(ii : ulong) : cc { return fcc[ii]; }
fun set_fcc(ii : ulong, CC1 : cc) : void { fcc[ii] = CC1; }

extern get_R4(cwp_t,ulong) : ulong;
extern set_R4(cwp_t,ulong,ulong) : void;

fun get_R4(cwp : cwp_t, i0 : ulong) {
    val ii = i0?ext(32);
    if(ii == 0) return 0?ext(32);
    else if(ii < 8) return global_registers[ii-1]?bits(32);
    else {
        val win = (cwp?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); return register_windows[win][ii]?bits(32);
    }
}

fun set_R4(cwp : cwp_t, i0 : ulong, vv : ulong) {
    val ii = i0?ext(32);
    if(ii >= 8) {
        val win = (cwp?cvt(ulong) - (ii / 16) + NWINDOWS) % NWINDOWS;
        ii = ii & 0xf?ext(32); register_windows[win][ii] = vv?sext(64);
    } else if(ii > 0) {
        global_registers[ii-1] = vv?sext(64);
    }
}
```

```
/*****************************************************************************
** FILE: sparc_v9.fs
** Facile description file for the SPARC v.9 instruction semantics.
** This file includes the SPARC v.9 encoding and register desriptions.
*/

#include "sparc_v9_enc.fs"
#include "sparc_v9_reg.fs"

// External routines to save and restore register windowsto the stack
extern save_regs(cwp_t,cwp_t) : void;
extern restore_regs(cwp_t) : void;

// Routines to flush all register windows to the stack
extern flush_windows(cwp_t,cwp_t) : void;
fun _flushw() {
    flush_windows(CWP,CANRESTORE);
    CANSAVE = (NWINDOWS - 2)?cvt(cwp_t);
    CANRESTORE = 0?cvt(cwp_t);
}
#define FLUSHW _flushw()

// routines to simulate SPARC/Solaris system calls
extern trap_sparc(ulong,cwp_t,cwp_t): void;
fun _trap(iflag,rs2,imm7) {
    val tnum;
    if(iflag) tnum = imm7?ext(32);
    else tnum = Rx(rs2)?cvt(ulong) & 0x7f?ext(32);

    if(tnum == 3) FLUSHW;
    else trap_sparc(tnum+256,CWP,CANRESTORE);
}
#define TRAP _trap(i,rs2,sw_trap)

#define TAG_OVERFLOW    0x23?ext(32)    // tag-overflow trap number
#define INVALID         0x21?ext(32)    // trap invalid trap number

fun annul() {   // skip next instruction
    nPC = nPC2;
    nPC2 = nPC + 4;
}

// Get second source operand (used for most integer instructions)
fun get_src2(iflag,rs2,simm) {
    if(iflag) return simm?sext(64);
    else return Rx(rs2);
}
#define SRC2 get_src2(i,rs2,simm13)

//
// branch/call instructions
```

```
sem call {
    nPC2 = PC + disp30?sext(32)<<2;
    Rx(15,PC?addr?ext(64));
};

sem jmpl {
    nPC2 = (Rx(rs1) + SRC2)?cvt(stream);
    Rx(rd,PC?addr?ext(64));
};

sem retrn {
    nPC2 = (Rx(rs1) + SRC2)?cvt(stream);
    if(CANRESTORE?cvt(ulong) > 0) {
        CANSAVE = (CANSAVE?cvt(ulong) + 1)?cvt(cwp_t);
        CANRESTORE = (CANRESTORE?cvt(ulong) - 1)?cvt(cwp_t);
    } else {
        restore_regs(CWP);
        if(CANSAVE?cvt(ulong) < NWINDOWS-2)
            CANSAVE = (CANSAVE?cvt(ulong) + 1)?cvt(cwp_t);
    }
    CWP = ((CWP?cvt(ulong) + NWINDOWS - 1) % NWINDOWS)?bits(5);
};

sem [ brz brlez brlz brnz brgz brgez ] {
    if(test(Rx(rs1),0?cvt(ullong)))
        nPC2 = PC + ((d16hi?sext(32)<<16) | (d16lo?ext(32)<<2));
    else if(a) annul();
} where test in [ == <= < != > >= ];

fun f_u(nn) { return fcc[nn]?bits(2) == 0b11; }
fun f_g(nn) { return fcc[nn]?bits(2) == 0b10; }
fun f_ug(nn) { return fcc[nn]?bit(1); }
fun f_l(nn) { return fcc[nn]?bits(2) == 0b01; }
fun f_ul(nn) { return fcc[nn]?bit(0); }
fun f_lg(nn) { val xx = fcc[nn]?bits(2); return xx==0b01 || xx==0b10; }
fun f_ne(nn) { return fcc[nn]?bits(2) != 0b00; }
fun f_e(nn) { return fcc[nn]?bits(2) == 0b00; }
fun f_ue(nn) { val xx = fcc[nn]?bits(2); return xx==0b00 || xx==0b11; }
fun f_ge(nn) { return !fcc[nn]?bit(0); }
fun f_uge(nn) { return fcc[nn]?bits(2) != 0b01; }
fun f_le(nn) { return !fcc[nn]?bit(1); }
fun f_ule(nn) { return fcc[nn]?bits(2) != 0b10; }
fun f_o(nn) { return fcc[nn]?bits(2) != 0b11; }

sem [ fba ba ] {
    nPC2 = PC + disp22?sext(32)<<2;
    if(a) annul();
};

sem [ fbpa bpa ] {
```

```
    nPC2 = PC + disp19?sext(32)<<2;
    if(a) annul();
};

sem [ fbn fbpn bn bpn ] { if(a) annul(); };

sem [                   fbu     fbg     fbug    fbl     fbul    fblg
        fbne    fbe     fbue    fbge    fbuge   fble    fbule   fbo     ] {
    if(cond(0)) nPC2 = PC + disp22?sext(32)<<2;
    else if(a) annul();
} where cond in [       f_u     f_g     f_ug    f_l     f_ul    f_lg
        f_ne    f_e     f_ue    f_ge    f_uge   f_le    f_ule   f_o     ];

sem [                   fbpu    fbpg    fbpug   fbpl    fbpul   fbplg
        fbpne   fbpe    fbpue   fbpge   fbpuge  fbple   fbpule  fbpo    ] {
    if(cond(bpccr)) nPC2 = PC + disp19?sext(32)<<2;
    else if(a) annul();
} where cond in [       f_u     f_g     f_ug    f_l     f_ul    f_lg
        f_ne    f_e     f_ue    f_ge    f_uge   f_le    f_ule   f_o     ];

// Access individual flags from the integer condition codes register (CCR)
#define C   CCR?bit(0)
#define V   CCR?bit(1)
#define Z   CCR?bit(2)
#define N   CCR?bit(3)
#define Cx  CCR?bit(4)
#define Vx  CCR?bit(5)
#define Zx  CCR?bit(6)
#define Nx  CCR?bit(7)

// Integer branch conditions
#define i_ne    (!Z)
#define i_e     (Z)
#define i_g     (!(Z|(N^V)))
#define i_le    (Z|(N^V))
#define i_ge    (!(N^V))
#define i_l     (N^V)
#define i_gu    (!(C|Z))
#define i_leu   (C|Z)
#define i_cc    (!C)
#define i_cs    (C)
#define i_pos   (!N)
#define i_neg   (N)
#define i_vc    (!V)
#define i_vs    (V)

// Integer conditions for 64-bit comparesons
#define x_ne    (!Zx)
#define x_e     (Zx)
#define x_g     (!(Zx|(Nx^Vx)))
#define x_le    (Zx|(Nx^Vx))
```

```
#define x_ge    (!(Nx^Vx))
#define x_l     (Nx^Vx)
#define x_gu    (!(Cx|Zx))
#define x_leu   (Cx|Zx)
#define x_cc    (!Cx)
#define x_cs    (Cx)
#define x_pos   (!Nx)
#define x_neg   (Nx)
#define x_vc    (!Vx)
#define x_vs    (Vx)

sem [                   bne     be      bg      ble     bge     bl
        bgu     bleu    bcc     bcs     bpos    bneg    bvc     bvs     ] {
    if(cond) nPC2 = PC + disp22?sext(32)<<2;
    else if(a) annul();
} where cond in [       i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

sem [                   bpne    bpe     bpg     bple    bpge    bpl
        bpgu    bpleu   bpcc    bpcs    bppos   bpneg   bpvc    bpvs    ] {
    if(bpcc1) {
        if(cond_xcc) nPC2 = PC + disp19?sext(32)<<2;
        else if(a) annul();
    } else if(cond_icc) nPC2 = PC + disp19?sext(32)<<2;
    else if(a) annul();
} where cond_xcc in [   x_ne    x_e     x_g     x_le    x_ge    x_l
        x_gu    x_leu   x_cc    x_cs    x_pos   x_neg   x_vc    x_vs    ],
        cond_icc in [   i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

//
// Conditional moves:

sem [ fmovfsn fmovfdn fmovsn fmovdn] {};
sem [ fmovfsa fmovfda fmovsa fmovda ] { fd(rd,fs(rs2)); }
where fs in [ F4 F8 F4 F8 ], fd in [ F4 F8 F4 F8 ];

sem [                                   fmovsne         fmovse
        fmovsg          fmovsle         fmovsge         fmovsl
        fmovsgu         fmovsleu        fmovscc         fmovscs
        fmovspos        fmovsneg        fmovsvc         fmovsvs         ] {
    if(bpcc1) { if(cond_xcc) F4(rd,F4(rs2)); }
    else if(cond_icc) F4(rd,F4(rs2));
} where cond_xcc in [   x_ne    x_e     x_g     x_le    x_ge    x_l
        x_gu    x_leu   x_cc    x_cs    x_pos   x_neg   x_vc    x_vs    ],
        cond_icc in [   i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

sem [                                   fmovdne         fmovde
        fmovdg          fmovdle         fmovdge         fmovdl
        fmovdgu         fmovdleu        fmovdcc         fmovdcs
```

```
        fmovdpos        fmovdneg        fmovdvc         fmovdvs         ] {
    if(bpcc1) { if(cond_xcc) F8(rd,F8(rs2)); }
    else if(cond_icc) F8(rd,F8(rs2));
} where cond_xcc in [   x_ne    x_e     x_g     x_le    x_ge    x_l
        x_gu    x_leu   x_cc    x_cs    x_pos   x_neg   x_vc    x_vs    ],
        cond_icc in [   i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

sem [                                   fmovfsu         fmovfsg
        fmovfsug        fmovfsl         fmovfsul        fmovfslg
        fmovfsne        fmovfse         fmovfsue        fmovfsge
        fmovfsuge       fmovfsle        fmovfsule       fmovfso         ] {
    if(cond(bpccr)) F4(rd,F4(rs2));
} where cond in [       f_u     f_g     f_ug    f_l     f_ul    f_lg
        f_ne    f_e     f_ue    f_ge    f_uge   f_le    f_ule   f_o     ];

sem [                                   fmovfdu         fmovfdg
        fmovfdug        fmovfdl         fmovfdul        fmovfdlg
        fmovfdne        fmovfde         fmovfdue        fmovfdge
        fmovfduge       fmovfdle        fmovfdule       fmovfdo         ] {
    if(cond(bpccr)) F8(rd,F8(rs2));
} where cond in [       f_u     f_g     f_ug    f_l     f_ul    f_lg
        f_ne    f_e     f_ue    f_ge    f_uge   f_le    f_ule   f_o     ];

sem [ fmovrsz fmovrslez fmovrslz fmovrsnz fmovrsgz fmovrsgez ]
{ if(test(Rx(rs1),0?cvt(ullong))) F4(rd,F4(rs2)); }
where test in [ == <= < != > >= ];

sem [ fmovrdz fmovrdlez fmovrdlz fmovrdnz fmovrdgz fmovrdgez ]
{ if(test(Rx(rs1),0?cvt(ullong))) F8(rd,F8(rs2)); }
where test in [ == <= < != > >= ];

sem [ mova movfa ] { Rx(rd,Rx(rs2)); };
sem [ movn movfn ] {};

sem [                   movne   move    movg    movle   movge   movl
        movgu   movleu  movcc   movcs   movpos  movneg  movvc   movvs   ] {
    if(bpcc1) { if(cond_xcc) Rx(rd,Rx(rs2)); }
    else if(cond_icc) Rx(rd,Rx(rs2));
} where cond_xcc in [   x_ne    x_e     x_g     x_le    x_ge    x_l
        x_gu    x_leu   x_cc    x_cs    x_pos   x_neg   x_vc    x_vs    ],
        cond_icc in [   i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

sem [                   movfu   movfg   movfug  movfl   movful  movflg
        movfne  movfe   movfue  movfge  movfuge movfle  movfule movfo   ] {
    if(cond(bpccr)) Rx(rd,Rx(rs2));
} where cond in [       f_u     f_g     f_ug    f_l     f_ul    f_lg
        f_ne    f_e     f_ue    f_ge    f_uge   f_le    f_ule   f_o     ];

sem [ movrz movrlez movrlz movrnz movrgz movrgez ]
```

```
{ if(test(Rx(rs1),0?cvt(ullong))) Rx(rd,Rx(rs2)); }
where test in [ == <= < != > >= ];

//
// System traps:

sem ta { TRAP; };
sem tn {};

sem [                   tne     te      tg      tle     tge     tl
        tgu     tleu    tcc     tcs     tpos    tneg    tvc     tvs     ] {
    if(bpcc1) { if(cond_xcc) TRAP; }
    else if(cond_icc) TRAP;
} where cond_xcc in [   x_ne    x_e     x_g     x_le    x_ge    x_l
        x_gu    x_leu   x_cc    x_cs    x_pos   x_neg   x_vc    x_vs    ],
        cond_icc in [   i_ne    i_e     i_g     i_le    i_ge    i_l
        i_gu    i_leu   i_cc    i_cs    i_pos   i_neg   i_vc    i_vs    ];

//
// Arithmetic ops:

#define C64 (CCR?bit(0)?ext(64))

sem [ add sub and or xor ]
{ Rx(rd, op(Rx(rs1),SRC2)); }
where op in [ + - & | ^ ];

sem [ addcc subcc andcc orcc xorcc ]
{ Rx(rd, op(Rx(rs1),SRC2)?cc(CCR)); }
where op in [ + - & | ^ ];

sem addc { Rx(rd, Rx(rs1) + SRC2 + C64); };
sem subc { Rx(rd, Rx(rs1) - SRC2 - C64); };

sem addccc {
    val ccr = 0x00?cvt(cc);
    val x1 = Rx(rs1); val x2 = SRC2;
    val xx = ((x1 + x2)?cc(ccr) + C64)?cc(CCR);
    CCR = ((CCR?bits(8) & 0b11011101) | (ccr?bits(8) & 0x11) |
          (((x1?bit(31)==x2?bit(31))&&(xx?bit(31)!=x1?bit(31)))?ext(8)<<1) |
          (((x1?bit(63)==x2?bit(63))&&(xx?bit(63)!=x1?bit(63)))?ext(8)<<5))
        ? cvt(cc);
    Rx(rd, xx);
};

sem subccc {
    val ccr = 0x00?cvt(cc);
    val x1 = Rx(rs1); val x2 = SRC2;
    val xx = ((x1 - x2)?cc(ccr) - C64)?cc(CCR);
    CCR = ((CCR?bits(8) & 0b11011101) | (ccr?bits(8) & 0x11) |
          (((x1?bit(31)!=x2?bit(31))&&(xx?bit(31)!=x1?bit(31)))?ext(8)<<1) |
```

```
          (((x1?bit(63)!=x2?bit(63))&&(xx?bit(63)!=x1?bit(63)))?ext(8)<<5))
        ? cvt(cc);
    Rx(rd, xx);
};

sem [ andn orn xnor ] { Rx(rd, op(Rx(rs1),~SRC2)); } where op in [ & | ^ ];

sem [ andncc orncc xnorcc ]
{ Rx(rd, op(Rx(rs1),~SRC2)?cc(CCR)); }
where op in [ & | ^ ];

sem [ mulx udivx ] { Rx(rd,  op(Rx(rs1),SRC2)); } where op in [ * / ];
sem sdivx { Rx(rd,(+Rx(rs1) / +SRC2)?cast(ullong)); };

sem sll  { Rx(rd,  Rx(rs1) << SRC2?bits(5)); };
sem srl  { R4(rd,  R4(rs1) >> SRC2?bits(5)); };
sem sra  { Rx(rd,(+R4(rs1) >> SRC2?bits(5))?sext(64)); };
sem sllx { Rx(rd,  Rx(rs1) << SRC2?bits(6)); };
sem srlx { Rx(rd,  Rx(rs1) >> SRC2?bits(6)); };
sem srax { Rx(rd,(+Rx(rs1) >> SRC2?bits(6))?cast(ullong)); };

sem taddcc {
    val x1 = Rx(rs1); val x2 = SRC2;
    Rx(rd, (x1 + x2)?cc(CCR));
    if(x1?bits(2)!=0b00 || x2?bits(2)!=0b00)
        CCR = (CCR?bits(8) | 0x02)?cvt(cc);
};

sem taddcctv {
    val x1 = Rx(rs1); val x2 = SRC2;
    Rx(rd, (x1 + x2)?cc(CCR));
    if(x1?bits(2)!=0b00 || x2?bits(2)!=0b00)
        CCR = (CCR?bits(8) | 0x02)?cvt(cc);
    if(CCR?bit(1)) trap_sparc(TAG_OVERFLOW,CWP,CANRESTORE);
};

sem tsubcc {
    val x1 = Rx(rs1); val x2 = SRC2;
    Rx(rd, (x1 - x2)?cc(CCR));
    if(x1?bits(2)!=0b00 || x2?bits(2)!=0b00)
    CCR = (CCR?bits(8) | 0x02)?cvt(cc);
};

sem tsubcctv {
    val x1 = Rx(rs1); val x2 = SRC2;
    Rx(rd, (x1 - x2)?cc(CCR));
    if(x1?bits(2)!=0b00 || x2?bits(2)!=0b00)
        CCR = (CCR?bits(8) | 0x02)?cvt(cc);
    if(CCR?bit(1)) trap_sparc(TAG_OVERFLOW,CWP,CANRESTORE);
};
```

```
fun u_div32(x1,x2,var ccr) {
    val xx = x1 / x2?ext(64);
    if(xx > 1?ext(64)<<32) { xx = 0xffffffff?ext(64); ccr = ccr | 0x02; }
    return xx;
}

fun s_div32(x1,x2,var ccr) {
    val xx = (+x1 / +x2?sext(64))?cast(ullong);
    if(+xx > +(1?ext(64)<<31)) { xx = 0x7fffffff?ext(64); ccr = ccr | 0x02; }
    else if(+xx < +((-1)?sext(64)<<31)) {
        xx = 0x80000000?sext(64);
        ccr = ccr | 0x02;
    }
    return xx;
}

sem [ udiv sdiv ] {
    val ccr = 0x00;
    val xx = ((Y?ext(64)<<32) | R4(rs1)?ext(64));
    if(i) xx = _div(xx,simm13?sext(32)?ext(64),ccr);
    else xx = _div(xx,R4(rs2)?ext(64),ccr);
    Rx(rd,xx);
} where _div in [u_div32 s_div32];

sem umul {
    val xx = (Rx(rs1)&0xffffffff?ext(64)) * (SRC2&0xffffffff?ext(64));
    Y = (xx >> 32)?bits(32); R4(rd,xx?bits(32));
};

sem smul {
    val xx = +Rx(rs1)?bits(32)?sext(64) * +SRC2?bits(32)?sext(64);
    Y = (xx >> 32)?bits(32); R4(rd,xx?bits(32));
};

fun get_div_mul_cc(xx,ccr0) {
    val ccr = ccr0 | xx?bit(63)?ext(8) << 7;
    ccr = ccr | (xx==0?ext(64))?ext(8) << 6;
    ccr = ccr | xx?bit(31)?ext(8) << 3;
    return ccr | (xx?bits(32)==0)?ext(8) << 2;
}

sem [ udivcc sdivcc ] {
    val ccr = 0x00;
    val xx = ((Y?ext(64)<<32) | R4(rs1)?ext(64));
    if(i) xx = _div(xx,simm13?sext(32),ccr);
    else xx = _div(xx,R4(rs2),ccr);
    CCR = get_div_mul_cc(xx,ccr)?cvt(cc);
    Rx(rd,xx);
} where _div in [u_div32 s_div32];

sem umulcc {
```

```
    val xx = (Rx(rs1)&0xffffffff?ext(64)) * (SRC2&0xffffffff?ext(64));
    CCR = get_div_mul_cc(xx,0x00)?cvt(cc);
    Y = (xx >> 32)?bits(32); R4(rd,xx?bits(32));
};

sem smulcc {
    val xx = (+Rx(rs1)?bits(32)?sext(64) *
          +SRC2?bits(32)?sext(64))?cast(ullong);
    CCR = get_div_mul_cc(xx,0x00)?cvt(cc);
    Y = (xx >> 32)?bits(32); R4(rd,xx?bits(32));
};

sem mulscc {
    val y0 = Y?bit(0); Y = (Y>>1) | (R4(rs1)?bit(0)?ext(32)<<31);
    val xx = (R4(rs1)>>1) | ((CCR?bit(3)^CCR?bit(1))?ext(32)<<31);
    if(y0) R4(rd, (xx + SRC2?bits(32))?cc(CCR));
    else R4(rd,(xx+0)?cc(CCR));
};

sem popc {
    val xx = Rx(rs2);
    val ii=0; val count=0?ext(64);
    while(ii < 64) {
        count = count + ((xx >> ii) & 0x1?ext(64));
        ii = ii + 1;
    }
    Rx(rd,count);
};

sem flush {};
sem flushw { FLUSHW; };

sem rd {
    switch(rs1?ext(32)) {
     case 0:    Rx(rd,Y?ext(64));
     case 2:    Rx(rd,CCR?bits(8)?ext(64));
     case 5:    Rx(rd,PC?addr?ext(64));
     case 15:   assert(rd?ext(32)==0);
     default:
        /* not implemented */
        assert(false);
    }
};

sem wr {
    switch(rd?ext(32)) {
     case 0:    Y = (Rx(rs1) ^ SRC2)?bits(32);
     case 2:        CCR = (Rx(rs1) ^ SRC2)?cvt(cc);
     default:
        /* not implemented */
        assert(false);
```

```
    }
};

sem save {
    val xx = Rx(rs1) + SRC2;
    if(CANSAVE?cvt(ulong) <= 0) {
        save_regs(CWP,CANRESTORE);
        CANSAVE = CANRESTORE; CANRESTORE = 0?cvt(cwp_t);
    }
    CWP = ((CWP?cvt(ulong) + 1) % NWINDOWS)?bits(5);
    CANSAVE = (CANSAVE?cvt(ulong) - 1)?cvt(cwp_t);
    CANRESTORE = (CANRESTORE?cvt(ulong) + 1)?cvt(cwp_t);
    Rx(rd,xx);
};

sem restore {
    val xx = Rx(rs1) + SRC2;
    if(CANRESTORE?cvt(ulong) > 0) {
        CANSAVE = (CANSAVE?cvt(ulong) + 1)?cvt(cwp_t);
        CANRESTORE = (CANRESTORE?cvt(ulong) - 1)?cvt(cwp_t);
    } else {
        restore_regs(CWP);
        if(CANSAVE?cvt(ulong) < NWINDOWS-2)
            CANSAVE = (CANSAVE?cvt(ulong) + 1)?cvt(cwp_t);
    }
    CWP = ((CWP?cvt(ulong) + NWINDOWS - 1) % NWINDOWS)?bits(5);
    Rx(rd,xx);
};

sem sethi { Rx(rd,imm22?ext(64)<<10); };

//
// Floating-point arithmetic:

sem [ fadds fsubs fmuls fdivs ]
{ F4(rd, op(F4(rs1),F4(rs2))); }
where op in [ + - * / ];

sem [ faddd fsubd fmuld fdivd ]
{ F8(rd, op(F8(rs1),F8(rs2))); }
where op in [ + - * / ];

sem [ fcmps fcmpd ] { (f(rs1)-f(rs2))?cc(fcc[cond]); } where f in [F4 F8];
sem [ fcmpes fcmped ] {
    (f(rs1)-f(rs2))?cc(fcc[cond]);
    if(f_u(cond)) trap_sparc(INVALID,CWP,CANRESTORE);
} where f in [F4 F8];

sem fstox { F8(rd,F4(rs2)?cvt(llong)?cast(ullong)); };
sem fdtox { F8(rd,F8(rs2)?cvt(llong)?cast(ullong)); };
sem fstoi { F4(rd,F4(rs2)?cvt(long)?cast(ulong)); };
```

```
sem fdtoi { F4(rd,F8(rs2)?cvt(long)?cast(ulong)); };
sem fstod { F8(rd,F4(rs2)?cvt(double)); };
sem fxtos { F4(rd,F8(rs2)?cast(llong)?cvt(float)); };
sem fxtod { F8(rd,F8(rs2)?cast(llong)?cvt(double)); };
sem fitos { F4(rd,F4(rs2)?cast(long)?cvt(float)); };
sem fitod { F8(rd,F4(rs2)?cast(long)?cvt(double)); };
sem fdtos { F4(rd,F8(rs2)?cvt(float)); };

sem [ fmovs fmovd ] { fd(rd,fs(rs2)); } where fs in [F4 F8], fd in [F4 F8];
sem [ fnegs fnegd ] { fd(rd,-fs(rs2)); } where fs in [F4 F8], fd in [F4 F8];
sem [ fabss fabsd ] { if(fs(rs2)<z) fd(rd,-fs(rs2)); else fd(rd,fs(rs2)); }
where fs in [F4 F8], fd in [F4 F8], z in [ (0?cvt(float)) (0?cvt(double)) ];

sem fsmuld { F8(rd, F4(rs1)?cvt(double) * F4(rs2)?cvt(double)); };

sem [fsqrts fsqrtd] {fd(rd,sqrt(fs(rs2)));} where fs in [F4 F8], fd in [F4 F8];

//
// Load, store, etc.:

sem [                   ldf     lddf                    ldfa    lddfa
        ldsb    ldsh    ldsw    ldub    lduh    lduw    ldx     ldd
        ldsba   ldsha   ldswa   lduba   lduha   lduwa   ldxa    ldda    ] {
    r(rd, m(Rx(rs1) + SRC2));
} where r in [          F4      F8                      F4      F8
        Rx      Rx      Rx      Rx      Rx      Rx      Rx      R8
        Rx      Rx      Rx      Rx      Rx      Rx      Rx      R8      ],
        m in [          M4      M8                      M4      M8
        M1s     M2s     M4s     M1      M2      M4      M8      M8
        M1s     M2s     M4s     M1      M2      M4      M8      M8      ];

sem [           stf     stdf    stb     sth     stw     stx     std
                stfa    stdfa   stba    stha    stwa    stxa    stda    ] {
    m(Rx(rs1) + SRC2, r(rd));
} where r in [  F4      F8      Rx      Rx      Rx      Rx      R8
                F4      F8      Rx      Rx      Rx      Rx      R8      ],
        m in [  M4      M8      M1      M2      M4      M8      M8
                M4      M8      M1      M2      M4      M8      M8      ];

sem [ casa casxa ] {
    val aa = Rx(rs1); val xx = ms(aa);
    if(rsrc(rs2)?ext(64) == xx) md(aa,rsrc(rd)); rdest(rd,xx);
} where rsrc in [R4 Rx], rdest in [R4 Rx],
        ms in [M4 M8], md in [M4 M8];

sem [ ldstub ldstuba ] { val aa=Rx(rs1)+SRC2; Rx(rd,M1(aa)); M1(aa,0xff); };

sem [ prefetch prefetcha ] {};

sem [ swap swapa ] {
    val aa = Rx(rs1) + SRC2; val xx = M4(aa);
```

```
    M4(aa,R4(rd)); R4(rd,xx);
};

sem ldfsr { set_FSR4(M4(Rx(rs1) + SRC2)?bits(32)); };
sem ldxfsr { set_FSR8(M8(Rx(rs1) + SRC2)); };
sem [ stfsr stxfsr ] { m(Rx(rs1) + SRC2, r()); }
where r in [ get_FSR4 get_FSR8 ], m in [ M4 M8 ];
```

BIBLIOGRAPHY

[1] A. Agarwal, J Hennessy, and M. Horowitz, “Cache Performance of Operating System and Multiprogramming Workloads,” in *ACM Transactions on Computer Systems*, vol. 6, no. 4, 393-431, November 1988.

[2] Anant Agarwal, Richard L. Sites, and Mark Horowitz, “ATUM: A New Technique for Capturing Address Traces Using Microcode,” in the *Proceedings of the 13th International Symposium on Computer Architecture*, 119-127, June 1986.

[3] Lars Ole Andersen, Program Analysis and Specialization for the C Programming Language, Ph.D. dissertation, DIKU, University of Compenhagen, Denmark, May 1994.

[4] Peter Holst Anderson, “Partial Evaluation Applied to Ray Tracing,” University of Copenhagen, DIKU, January 1993.

[5] Kristy Andrews and Duane Sand, “Migrating a CISC Computer Family onto RISC via Object Code Translation,” in the *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 213-222, October 1992.

[6] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad, “Fast, Effective Dynamic Compilation,” in the *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[7] Romana Baier, Robert Glük, Robert Zöchling, “Partial Evaluation of Numerical Programs in Fortran,” in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (, ed.), pp. 119-132, Volume 94/9 of Technical Report, University of Melbourne, Australia, 1994.

[8] Robert A. Baumann, “Z80MU,” Byte, 203-216, Oct. 1986.

[9] Robert Bedichek, “Some Efficient Architecture Simulation Techniques,” *USENIX’88*.

[10] Robert C. Bedichek, “Talisman: Fast and Accurate Multicomputer Simulation,” MIT, 1995.

[11] Anita Borg, R. E. Kessler, and David W. Wall, “Generation and Analysis of Very Long Address Traces,” in the *Proceedings of the 17th Annual Symposium on Computer Architecture*, 270-279, May 1990.

[12] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, “PROTEUS: A High-Performance Parallel-Architecture Simulator,” MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[13] Doug Burger, and Todd M. Austin, “The SimpleScalar Tool Set, Version 2.0,” *Tech Report #1342*, University of Wisconsin-Madison, Department of Computer Sciences, June, 1997.

[14] Craig Chambers, David Ungar, and Elgin Lee, “An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes,” *OOPSLA '89 Proceedings*, 49-70, October 1989.

[15] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, John Yates, “FX!32 a profile-directed binary translator,” in *IEEE Micro*, vol. 18, no. 2, 56-64, March-April 1998.

[16] F. Chow, M. Himelstein, E. Killian, and L. Weber, “Engineering a RISC Compiler System,” IEEE COMPCON, March 1986.

[17] Bob Cmelik, and David Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling,” in the *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[18] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi, “A Uniform Approach for Compile-time and Run-time Specialization,” *Rapport de recherche N° 2775, Institut National de Recherche en Informatique et en Automatique (INRIA)*, France, 1996.

[19] Charles Consel and François Noël, “A General Approach for Run-Time Specialization and its Application to C,” in the *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Conference on Principles Of Programing Languages (POPL)*, St. Petersburg Beach, FL, 145-156, January 1996.

[20] T. M. Conte, Systematic Computer Architecture Prototyping, Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

[21] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, “The Rice Parallel Processing Testbed,” *ACM SIGMETRICS*, 4-11, 1988.

[22] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel, “Effective Control for Pipelined Computers,” in the *Spring COMPCON’75 Digest of Papers*, 181-184, February 1975.

[23] Peter Davies, Philippe LaCroute, John Heinlein, and Mark Horowitz, “Mable: A Technique for Efficient Machine Simulation,” (to appear), Quantum Effect Design, Inc., and Stanford University.

[24] Peter Deutsch and Alan M. Schiffman, “Efficient Implementation of the Smalltalk-80 System,” in the *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, 297-302, Jan. 1984.

[25] S. Devadas, S Malik, K. Keutzer, and A. Wang, “Event Suppression: Improving the Efficiency of Timing simulation for Synchronous Digital Circuits,” in *IEEE Transactions on Computer-Aided Design*, vol. 13, 814-822, June 1994.

[26] Susan J. Eggers, David Keppel, Eric J. Koldinger, and Henry M. Levy, “Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor,” *ACM SIGMETRICS*, 37-47, May 1990.

[27] Alexander E. Eichenberger and Edward S. Davidson, “A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints,” in the Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, PA, 12-22, May 1996.

[28] Richard M. Fujimoto and William B. Campbell, “Efficient Instruction Level Simulation of Computers,” in *Transactions of The Society for Computer Simulation*, 5(2): 109-124, 1988.

[29] Stephen R. Goldschmidt and John L. Hennessy, “The Accuracy of Trace-Driven Simulations of Multiprocessors,” CSL-TR-92-546, Stanford University Computer Systems Laboratory, Sepember 1992.

[30] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers, “Annotation-Directed Run-Time Specialization in C,” in the *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 163-178, 1997.

[31] John C. Gyllenhaal, Wen-mei W. Hwu, and B. Ramabriohna Rau, “Optimization of Machine Descriptions for Efficient Use,” in the *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, 349-358, 1996.

[32] Reed Hastings and Bob Joyce, “Purify: Fast Detection of Memory Leaks and Access Errors,” in the *Proceedings of the Winter Usenix Conference*, 125-136, Jan. 1992.

[33] John Hennessy and David Patterson, Computer Organization and Design: The Hardware-Software Interface (Appendix A, by James R. Larus), Morgan Kaufman, 1993.

[34] Steve Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Kinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben Verghese, “The SimOS Simulation Environment,” Computer Systems Laboratory, Stanford University, 1996.

[35] N. D. Jones, C. Gomard, and P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[36] R. E. Kessler, Mark D. Hill, and David A. Wood, “A Comparison of Trace Sampling Techniques for Multi-Megabyte Caches,” in *IEEE Transactions on Computers*, vol. 43, no. 6, 664-675, June 1994.

[37] Subhasis Laha, Janak H. Patel, and Ravishankar K. Iyer, “Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems,” in IEEE Transactions on Computers, vol. 37, no. 11, 1325-1336, November 1988.

[38] James R. Larus and Thomas Ball, “Rewriting Executable Files to Measure Program Behavior,” *Software - Practice and Experience*, 24(2): 197-218, February 1994.

[39] James R. Larus and Eric Schnarr, “EEL: Machine-Independent Executable Editing,” in the *Proceedings of the ACM SIGPLAN ‘95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[40] G. Lauterbach, “Accelerating Architecture Simulation by Parallel Execution,” in the *Proceedings of the 27th Hawaii International Conference on System Science*, Maui, HI, January 1994.

[41] Peter Lee and Mark Leone, “Optimizing ML with Run-Time Code Generation,” in the *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 137-148, May 1996.

[42] Mark Leone and Peter Lee, “Lightweight Run-Time Code Generation,” in the *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 97-106, June 1994.

[43] Peter S. Magnusson, “A Design For Efficient Simulation of a Multiprocessor,” in the *Proceedings of the First International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, La Jolla, California, January 1993.

[44] Peter S. Magnusson, “Partial Translation,” Swedish Institute of Computer Science, March 1994.

[45] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner, “SimICS/sun4m: A Virtual Workstation,” in *Usenix Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[46] Cathy May, “Mimic: A Fast S/370 Simulator,” in the *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*; SIGPLAN Notices, 22(6): 1-13, June 1987.

[47] Robin Milner, “A theory of Type Polymorphism in Programming,” in the Journal of Computer and System Sciences, 17(3): 348-375, December 1978.

[48] MIPS, Languages and Programmer's Manual, MIPS Computer Systems, Inc., 1986.

[49] Torben Mogensen, The Application of Partial Evaluation to Ray Tracing, Masters Thesis, DIKU, University of Copenhagen, Denmark, 1986.

[50] Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet, “Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol,” in the *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Amsterdam, The Netherlands, 116-126, June 1997.

[51] Thomas Müller, “Employing Finite Automata for Resource Scheduling,” in the *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 12-20, 1993.

[52] Robert D. Nielsen, “DOS on the Dock,” NeXTWorld, 50-51, Mar./Apr. 1991.

[53] Soner Önder and Rajiv Gupta, “Automatic Generation of Microarchitecture Simulators,” in the *IEEE International Conference on Computer Languages (ICCL98)*, Chicago, May 1998.

[54] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, “RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors,” in the *Proceedings of the 3rd Workshop on computer Architecture Education (held in conjunction with the 3rd International Symposium on High Performance Computer Architecture)*, February 1997.

[55] Kwang Il Park and Kyu Ho Park, “Event Suppression by Optimizing VHDL Programs,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 8, 682-691, August 1998.

[56] A. Poursepanj, “The PowerPC Performance Modeling Methodology,” in communications of the ACM, vol. 37, no. 6, 47-55, June 1994.

[57] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. "Microlanguages for Operating System Specialization". in the *Proceedings of the SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[58] Calton Pu, Tito Autrey, Andrew Black, Charles Consul, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang, “Optimistic Incremental Specialization: Streamlining a Commercial Operating System,” in the *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, ACM Operating Systems Reviews, 29(5), 314-324, December 1995.

[59] Calton Pu, Henry Massalin, and John Ioannidis, "The Synthesis Kernel", in *Computing Systems*, University of California Press, 1(1):11-32, Winter 1988.

[60] Norman Ramsey and Mary Fernandez, “The New Jersey Machine-Code Toolkit,” in the *Proceedings of the USENIX Technical Conference*, New Orleans, LA, 289-302, January 1995.

[61] R. Razdan, G. P. Bischoff, and E. G. Ulrich, “Clock Suppression Techniques for Synchronous Circuits,” in *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 1457-1556, October 1993.

[62] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers on Measurement and Modeling of Computer Systems,” *ACM SIGMETRICS*, 48-60, June 1993.

[63] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood, “Decoupled Hardware Support for Distributed Shared Memory,” in the *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1996.

[64] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod, “Using the SimOS Machine Simulator to Study Complex Computer Systems,” in *ACM Transactions on Modeling and computer Simulation*, vol. 7, no. 1, 78-103, January 1997.

[65] Eric Schnarr and James R. Larus, “Fast Out-Of-Order Processor Simulation Using Memoization,” in the *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 4-7, 1998.

[66] Gabriel M. Silberman and Kemal Ebciog–lu, “An Architectural Framework for Supporting Heterogeneous Instruction Set Architectures,” *IEEE Computer*, 39-56, June 1993.

[67] Richard L. Sites, Anton Chernoff, Matthew B. Kerk, Maurice P. Marks, and Scott G. Robinson, “Binary Translation,” *CACM*, 36(2): 69-81, February 1993.

[68] Rok Sosic, “Dynascope: A Tool for Program Directing,” in the *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation (PLDI)*, 12-21, June 1992.

[69] Amitabh Srivastava and Alan Eustace, “ATOM: A System for Building Customized Program Analysis Tools,” *WRL Research Report 94/2*, Western Research Laboratory, Digital Equipment Corporation, 1994.

[70] H. S. Stone, High-Performance Computer Architecture, second ed., Reading, MA, Addison-Wesley, 1990.

[71] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs, “Address Tracing of Parallel Systems via TRAPEDS,” *Microprocessors and Microsystems*, 16(5): 249-261, 1992.

[72] Sun Microsystems, The SPARC Architecture Manual (Version 8), December 1990.

[73] Jack E. Veenstra and Robert J. Fowler, “MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors,” in the *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 201-207, January 1994.

[74] Darren Erik Vengroff, Kenneth Simpson, and Guang R. Gao, “Non-Clustered Statistical Trace Sampling for Large Cache Design Space Exploration,” in the *Proceedings of the Workshop on Performance Analysis and its Impact on Design (PAID)*, Denver, Colorado, June 1997.

[75] Eugen N. Volanschi, Charles Counsel, Gilles Muller, and Crispin Cowan, “Declarative Specialization of Object-Oriented Programs,” in the *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Atlanta, GA, 286-300, October 1997.

[76] Edward Wang and Paul N. Hilfinger, “Analysis of Recursive Types in Lisp-like Languages,” in the Proceeding of the Conference on Lisp and Functional Programming, San Francisco, CA, 216-225, June 1992.

[77] Qiang Wang and David M. Lewis, “Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation,” in the *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 145-154, 1997.

[78] SPARC International, Inc., The SPARC Architecture Manual Version 9, edited by David L. Weaver and Tom Germond, PTR Prentice Hall, 1994.

[79] John C. Willis and Daniel P. Siewiorek, “Optimizing VHDL Compilation for Parallel Simulation,” in *IEEE Design & Test of Computers*, vol. 9, issue 3, 42-53, September 1992.

[80] D. A. Wood, M. D. Hill, and R. E. Kessler, “A Model for Estimating, Trace-Sample Miss Ratios,” in the *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 79-89, 1991.

[81] Yeager, “The Mips R10000 Superscalar Microprocessor,” in *IEEE Micro*, April 1996.