Appeared: *Workshop on Compiler Support for System Software* (*WCSSS'96*), Tucson, Arizona, February 1996.

Instruction Scheduling and Executable Editing[1]

Eric Schnarr and James R. Larus

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
(608) 262-9519
{schnarr,larus}@cs.wisc.edu
February 2, 1996

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. The resulting disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost or even no-cost instrumentation for measurement, simulation, or emulation. Instrumentation code that executes in previously unused processor cycles is effectively hidden. These microprocessors also pose another problem, which arises from the machine-specific instruction scheduling necessary for high performance. Different implementations of an architecture, such as the many x86 processors, may benefit from different schedules, which either requires multiple executables or a way to reschedule existing programs for new machines.

We investigated both opportunities by adding an instruction scheduler to the EEL executable editing library. On first-generation, 2 and 3-way superscalar SPARC processors, this simple, local scheduler hid an average of 17% (8–22%) of the overhead cost of profiling instrumentation in the SPECINT benchmarks and an average of 28% (5–53%) of the profiling cost in the SPECFP benchmarks. On a second-generation, 4-way superscalar UltraSPARC, the scheduler hid an average of 16% (8–21%) of the profiling cost in the SPECINT benchmarks and 65% (7–136%) in the SPECFP benchmarks. We also used the scheduler to reschedule uninstrumented code previously compiled for the SuperSPARC. Scheduling that takes into account the UltraSPARC's out-of-order execution improved the SPECFP benchmarks by an average of 9% (1–33%).

1 Introduction

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. For example, the most recent generation of RISC chips—the Digital Alpha 21164, IBM/Motorola Power PC 620, SGI R10000, and Sun UltraSparc—are 4-way superscalar processors that execute up to four independent instructions in a single cycle. Even recent high-end x86 processors—the Intel Pentium Pro and AMD K5—are 3-way superscalars. Unfortunately, exploited instruction-level parallelism lags far behind the hardware. Cvetanovic and Bhandarkar found that programs running on a 2-way superscalar Digital Alpha 21064 could dual issue only 20–50% of their instructions, which means that in 67–90% of cycles, only one instruction executed [1]. Similarly, Diep et al. found that on a 4-way superscalar Power PC 620, four integer SPEC benchmarks completed an average of 1.05–1.25 instructions per cycle and three floating-point SPEC benchmarks completed an average of 1.0–1.9 instructions per cycle [3].

This large disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost or even no-cost instrumentation for measurement, simulation, or emulation. Scheduling reduces instrumentation's perceived cost by

1. Copyright © 1996 by Eric Schnarr and James Larus.
Measurements in this paper are preliminary and may change slightly in later versions.

putting instrumentation instructions in previously unused processor cycles. Instrumentation code that executes in unused cycles is effectively hidden.

Program instrumentation has been used for many purposes, including performance measurement, computer architecture simulations, and software fault isolation and error detection. Although direct instrumentation typically incurs lower cost than alternative approaches, the increased program running time caused by instrumentation is occasionally a limiting factor and is always an annoyance. At one extreme, in parallel or real-time systems, instrumentation overhead can perturb system behavior by introducing time dilation. Even for less demanding applications, the cost of program profiling or error detection is currently too high for production codes. Previous instrumentation systems expended considerable effort to formulate efficient instrumentation code sequences [10], place them parsimoniously [1], and insert instrumentation without affecting a program's behavior [8]. However, few systems attempted to exploit instruction-level parallelism by scheduling the instrumentation code into a program.

This paper describes a simple instruction scheduler that we added to the EEL program executable editing library [9]. We applied the scheduler to a common program instrumentation, which records a basic block's execution frequency with a four-instruction sequence. On the dual-issue hyperSPARC, scheduling hides 13–21% of the overhead of profiling the SPECINT benchmarks (compiled with the Sun Compiler) and 5–53% of the overhead from the SPECFP benchmarks (with one exception). The results on the 3-way SuperSPARC are similar, as scheduling hides 8–22% of the SPECINT overhead and 6–48% of the SPECFP overhead (one exception). The results on the 4-way UltraSPARC are similar to the other two microarchitectures for the SPECINT benchmarks, as scheduling hides 8–21% of profiling overhead. On the SPECFP benchmarks, scheduling is far more effective and hides an average of 65% (7–136%) of the profiling overhead (one exception). In the future, these results may improve, and scheduling become even more attractive, with a more accurate and aggressive instrumentation scheduler and more aggressive microarchitectures that offer further opportunities to hide instrumentation.

Beyond hiding instrumentation, superscalar processors offer another opportunity to combine instruction scheduling with executable editing—which we previously mentioned in passing [9]. Modern, aggressive microprocessors require careful, machine-specific instruction scheduling to achieve their highest performance. A computer architecture may have several implementations whose microarchitectures and scheduling constraints differ widely. Some of these implementations may be older machines, but others may be contemporary, competing products. Consider, for example, the state of the high-end x86 world shown in Table 1. These five high-performance processors offer a wide range of available parallelism and out-of-order execution possibilities, and each processor presents its own set of scheduling rules and constraints.

Processor	Dispatch Rate	Function Units	Out of Order Buffer	Rename Registers
Intel Pentium	2	3	–	–
Intel Pentium Pro	3	5	40	40
AMD K5	2–3	7	16	16
Cyrix M1	2	2	–	32
NexGen Nx586	1	4	14	22

Table 1: High-end x86 processors [4]. Dispatch rate is the number of instructions that can be issued per cycle. Function units is the number of independent units on the chip. Out of order buffer is the size of the instruction reorder buffer (in instructions). Rename registers is the number of registers available for register renaming.

We also used EEL's instruction scheduler to measure the benefits of *instruction rescheduling*, where fully compiled (and likely scheduled) legacy programs are rescheduled to optimize for a different microarchitecture. In experiments on SPARC processors, rescheduling proved beneficial when the new microarchitecture offered features that were unavailable on the older machine. On the UltraSPARC, rescheduling

improved the performance of programs previously compiled and optimized for the SuperSPARC by up to 33% (average speedup of 9% for the SPECFP benchmarks) by scheduling that accounted for out-of-order execution on level-1 cache misses. Rescheduling for the hyperSPARC produced no speedup on average (-3 to 5%), because there are few differences between this machine's microarchitecture and the SuperSPARC.

To implement instruction scheduling, we extended EEL with the specific details of a processor's microarchitecture. EEL uses a concise, high-level specification describing a machine's instruction set architecture. A tool called Spawn [9] translates these specifications into executable C++ code, which becomes the part of EEL that decodes and interprets machine instructions. As part of this work, we extended the architecture specification language to capture salient features of a machine's microarchitecture and used this information to drive an instruction scheduler in EEL. This paper describes this language extension and how EEL uses the information to schedule instructions.

The paper is organized as follows. Section 2 describes related work. Section 3 presents the extensions to Spawn and shows how they represent the information necessary for instruction scheduling. Section 4 describes our experiments on scheduling program instrumentation and rescheduling uninstrumented codes for different target microarchitectures. Finally, Section 5 concludes.

2 Related Work

This paper extends several areas of previous work. Patil and Fischer used a different form of parallelism to hide instrumentation overhead. On a multiprocessor, they ran on a second processor the code to check pointer and array accesses [11]. The additional processor reduced the perceived overhead of error checking by 2–55%. Our approach is more widely applicable since instruction-level parallel processors are, or soon will be, ubiquitous and because instruction-level parallelism permits a tighter coupling between program and instrumentation.

Proebsting and Fraser described an efficient algorithm for detecting structural hazards in single pipeline processors and a language for concisely describing these machines [12]. Our description language is more verbose, but it also captures the syntax and semantics of machine instructions and scheduling constraints for superscalar machines. Our algorithm is also less efficient, but more general since it works for superscalar processors as well as single pipeline machines.

Gyllenhaal described the machine description language (HMDES) used in the Illinois IMPACT compiler [5]. This language, like Spawn's description language, describes instruction encoding and scheduling constraints. HMDES describes instructions from several perspectives, which together provide the basic information for instruction schedulers: instruction latencies and resource usage. Spawn represents this information more concisely as a single semantic expression for a group of instruction, which describes when these instruction acquire and release resources. Spawn descriptions also capture instruction semantics and binary encodings.

Schuette detected processor errors in unused instruction slots on a VLIW processor [14]. His control-flow monitoring inserted check operations into unfilled VLIW instructions. This is similar to EEL's rescheduling of instrumentation, except that EEL also reschedules the original instructions and can handle many different forms of instrumentation. Because a VLIW has fixed size instructions, Schuette's instrumentation did not increase code size. On a superscalar, instrumentation code has a secondary effect of reducing performance by increasing instruction cache misses.

3 Spawn Extensions

Any executable editing tool depends on an accurate description of a machine's instruction set architecture. While most parts of a tool such as EEL are machine independent, it still must disassemble, analyze, and modify binary machine instructions. Past experiences with executable editing tools demonstrated that the code that manipulates binary instructions is simple, but tedious and difficult to debug. For example, in the

profiling and tracing tool qpt [8], over 2,000 lines of hand-written C code exist to manipulate binary instructions. Subtle errors in this code were difficult to detect by inspection and often lay undiscovered for months before a new input executable exercised them.

To remedy this problem and make EEL portable across different processors, it uses a concise description of a machine's instruction set architecture written in a high-level architecture description language. These descriptions are short—the SPARC is 333 lines—and similar in form to the descriptions often found in architecture manuals, which makes them easier to validate. Errors overlooked in a code review are also more likely to arise during testing, since a single description of an instruction's encoding and semantics underlies many different EEL instruction manipulation functions and different instructions often share common code in the description file.

Spawn [9] is the tool that converts an architecture description into the C++ code used by EEL. Spawn analyzes descriptions written in SADL (Spawn Architecture Description Language) to detect errors and extract syntactic and semantic information needed by EEL. Spawn then reads an annotated C/C++ file and replaces its annotations by appropriate code produced using information extracted from the description. The resulting C/C++ file is compiled and linked into EEL to provide efficient functions for manipulating binary instructions.

Instruction scheduling requires more detailed information than the architectural description that sufficed previously for EEL. In particular, scheduling requires a model of a machine's microarchitecture, especially its execution pipeline. Since a microarchitecture is specific to a particular processor implementation, this level of detail entails writing many more descriptions, so each description should be concise and easy to modify. To support instruction scheduling, we extended SADL to include pipeline timing and resource utilization information. This new information can be combined with the instruction semantic description to provide a complete map of an instruction's actions as it moves through a processor's execution pipeline.

Ideally, one would like to separate a microarchitecture-specific description from a general architecture description. SADL only partially accomplishes this goal by supporting functions that encapsulate some of the timing and resource usage characteristics. The coupling of architectural and resource allocation information is necessary to permit Spawn to determine when register values are read and when result values become visible. On the other hand, our experience modeling the hyperSPARC, SuperSPARC, and UltraSPARC processors showed that the architecture semantics are easily carried over into a description of a different microarchitecture, despite large changes in the pipeline description.

A (micro)architecture description, written in SADL, describes several aspects of a machine's architecture: instruction encodings, instruction semantics, architectural registers, and pipeline resources. The first three aspects (encodings, semantics, and registers) are described elsewhere [9]. Section 3.1 describes how pipeline resources and timing operators are combined with instruction semantics to encode details of an execution pipeline. Section 3.2 describes how an instruction scheduler uses this information to predict pipeline behavior.

3.1 Describing an Execution Pipeline

The first part of a pipeline description names and enumerates the hardware resources—register ports, ALUs, etc. Each resource is described by a unit name and a value representing the number of copies of this resource that reside in the processor. Figure 1 shows some of the units defined for the ROSS hyperSPARC microarchitecture [13]. This processor is a dual-issue superscalar with an ALU for arithmetic operations

```
unit Group 2                // number of simultaneous instructions

unit ALU 1, ALUr 2, ALUw 1  // ALU capacities
unit LSU 1, LSUr 2, LSUw 1  // LSU capacities
```

Figure 1: Functional unit definitions used in describing the ROSS hyperSPARC execution pipeline.

```
val multi is AR Group, ()  // potential dual issue instructions
val single is AR Group 2, ()// cannot be dual issued

register untyped{32} R[36] // general purpose registers (GPR)

// Aliases used to read and write from the GPR
alias signed{32} R4r[i] is AR ALUr, R[i] // ALU read from GPR
alias signed{32} R4w[i] is AR ALUw, R[i] // ALU write to GPR

// Defining arithmetic and logical operators
val [ +       -       &       |       ^     ]
 is (\op.\a.\b. D 1, A ALU, x:=op a b, D 1, R ALU, x)
  @ [ add32   sub32   and32   or32    xor32 ]

// Defining shift operators
val [ <<      >>      >>    ]
 is (\op.\a.\b. D 1, A ALU, isShift, x:=op a b, D 1, R ALU, x)
  @ [ sll32   srl32   sra32 ]

// Get the second source operand or immediate value
val src2 is iflag=1 ? #simm13 : R4r[rs2]

// Semantic description of the instructions add, sub, and sra
sem [ add     sub     sra   ]
 is (\op. multi, s1:=R4r[rs1], s2:=src2, R4w[rd]:=op s1 s2)
  @ [ +       -       >>    ]
```

Figure 2: Semantic description of the SPARC instructions add, sub, and sra. The resource usage and timing information are for the ROSS hyperSPARC microarchitecture.

and an ALU for memory address calculations (LSU). The ALUr and ALUw units in the figure represent read and write ports to the register file for the arithmetic ALU. Similarly, the LSUr and LSUw units are ports used by the LSU.

The second part of a description specifies the pipeline behavior of an instruction and is combined with the instruction's semantic description. The SADL commands A, R, AR, and D describe when units are acquired and released and when the pipeline advances. The command A <*unit*> [<*num*>] acquires <*num*> copies of the unit, or stalls the pipeline if not enough copies of the unit are available. If <*num*> is omitted, it is assumed to be 1. R <*unit*> [<*num*>] releases <*num*> copies of a unit. The command AR <*unit*> [<*num*> [<*delay*>]] acts like the A command, but it also releases <*num*> copies of the same unit after the instruction executes for <*delay*> cycles. AR is handy for acquiring a resource for a fixed amount of time, without having to do R's later in a semantic expression. The command D [<*delay*>] advances the pipeline by <*delay*> cycles.

An instruction scheduling algorithm must also know when registers are read and written. When an instruction reads a register, the description must record the pipeline cycle in which the read occurred. Writes are more difficult, since most pipelined implementations forward values between instructions [6]. When a value is computed, Spawn records the cycle in which the computation finished. When this value is written back into a register, Spawn tracks the cycle in which the value was computed, not when the register assignment took place. Therefore, a SADL instruction description must place the computation of an instruction's result value in the cycle in which the value becomes available to the next instruction.

Figure 2 illustrates the semantics and pipeline behavior of three hyperSPARC ALU instructions. This description contains several types of statements: val statements, which act like macro definitions; register declarations, which define the architectural registers; alias statements, which define alternate views of registers (or memory); and sem statements, which bind semantic expressions to instruction mnemonics. The

effect of the sem statement above is to bind the instructions add, sub, and sra, to their semantic expressions which: (1) restrict the pipeline to at most 2 simultaneous instructions; (2) acquire one or two ALU read ports and read the register values; (3) advance the pipeline by 1 cycle and release the ALU read ports; (4) acquire the ALU functional unit and compute the instruction's result value; (5) advance the pipeline by one cycle and release the ALU functional unit; (6) acquire an ALU write port and update the destination register; and (7) advance the pipeline by one cycle and release the ALU write port. From this description, Spawn infers that these instructions can be dual issued, execute in 3 cycles, read their operands in cycle 0, produce a value that subsequent instructions can use in cycle 1, and update the register file in cycle 2.

Spawn organizes resource acquire and release information into groups. Instructions with identical timing and resources allocation patterns go into the same group. Each group records the number of cycles it takes for a member instruction to pass completely through the pipeline, the resources acquired in each cycle, and the resources released in each cycle. Spawn also associates a cycle number with every access to the integer or floating point register file, for both reads and writes. For reads, this number indicates the pipeline execution cycle when the read occurs. For writes, it indicates when the written value actually becomes available to subsequent instructions, not when the value is written to the register file.

3.2 Predicting Pipeline Behavior

The key metric used by EEL's instruction scheduler is the number of cycles that the next instruction must wait before entering the execution pipeline. SADL describes the resource usage and timing for individual instruction. This information can describe the execution behavior of many superscalar processors executing a straight-line sequence of instructions.

Spawn passes this information to an instruction scheduler by filling in annotations in the C++ function, pipeline_stalls, which given a sequence of instruction, computes when the next instruction can start execution (see Appendix A for an overview of this function). This function starts with a representation of the pipeline state, which captures the state of the microarchitecture after the previous instructions in a sequence, and the instruction whose delay is to be computed. The pipeline state includes history information, such as the last cycle in which each register was read and written and which units are currently acquired by previous instructions. pipeline_stalls uses this state information to simulate the pipeline execution of the new instruction and computes how many stall cycles are needed to satisfy the RAW, WAR, WAW dependencies and structural hazards.

The Spawn microarchitecture models can describe a limited subset of the possible integer and floating point pipelines. Our goal has been to describe actual machines rather than hypothetical systems. The descriptions contain no information about a processor's memory interface to instruction prefetching, write buffering, or instruction and data cache behavior. pipeline_stalls does not compute stalls due to these mechanisms. On the other hand, few scheduling algorithms take these features into account since their behavior is data-dependent (c.f. [7]). In addition, SADL does not yet describe out-of-order execution, since it was not needed for the descriptions produced so far. We will soon add this feature.

4 Scheduling Instrumentation

EEL schedules instructions in a basic block (local scheduling). The instructions in the block come either from the original program (for rescheduling) or a combination of the program and instrumentation code. If instrumentation contains branches, the scheduler only processes the regions of straight-line code. The scheduler uses the common two pass list scheduling algorithm [6]. The first pass starts at the end of the block and works backwards to compute the length (in cycles) of the dependence chain between every instruction and the end of the block. This computation only considers the stalls required between data dependent instructions.

The second pass starts at the beginning of the block and works forward, to order instructions with list scheduling. The instruction with the highest priority of any instruction that can be legally scheduled at this

Benchmark	BB Size	Sun Compiler				gcc			
		Uninst. Time	Inst. Time	Sched. Time	% Hidden	Uninst. Time	Inst. Time	Sched. Time	% Hidden
espresso	2.6	8.1	20.4 (2.52)	17.9 (2.20)	21.2%	9.2	20.5 (2.22)	17.9 (1.94)	23.2%
xlisp	2.2	118.1	273.8 (2.32)	249.0 (2.11)	15.9%	127.2	277.4 (2.18)	252.3 (1.98)	16.7%
eqntott	2.0	10.5	27.0 (2.57)	24.3 (2.32)	16.3%	21.2	58.7 (2.77)	48.3 (2.28)	27.6%
compress	3.3	8.7	14.1 (1.62)	13.1 (1.51)	17.5%	8.9	14.6 (1.64)	13.8 (1.55)	13.4%
sc	1.9	47.9	84.1 (1.76)	78.2 (1.63)	16.3%	31.8	65.4 (2.06)	58.8 (1.85)	19.8%
cc1	2.5	10.6	23.4 (2.21)	21.8 (2.06)	12.6%	6.4	16.2 (2.53)	15.2 (2.38)	9.9%
spice2g6	4.1	510.3	794.5 (1.56)	711.7 (1.39)	29.1%				
doduc	5.6	28.3	43.2 (1.52)	41.1 (1.45)	13.7%				
mdljdp2	4.9	6.6	10.1 (1.53)	8.8 (1.33)	37.7%				
wave5	8.0	79.0	102.4 (1.30)	95.6 (1.21)	29.3%				
tomcatv	19.9	42.5	46.5 (1.10)	45.8 (1.08)	19.1%				
ora	4.5	45.9	56.5 (1.23)	51.8 (1.13)	44.6%				
alvinn	6.8	132.6	204.4 (1.54)	166.7 (1.26)	52.5%				
ear	5.8	433.4	591.3 (1.36)	551.1 (1.27)	25.5%				
mdljsp2	4.6	5.7	9.1 (1.61)	7.7 (1.36)	41.0%				
swm256	67.3	223.8	234.5 (1.05)	235.6 (1.05)	-10.2%				
su2cor	12.1	117.5	135.3 (1.15)	128.9 (1.10)	35.8%				
hydro2d	4.8	147.5	231.4 (1.51)	209.9 (1.42)	25.5%				
nasa7	15.0	318.0	343.8 (1.08)	335.8 (1.06)	31.3%				
fpppp	12.9	116.1	161.9 (1.39)	159.6 (1.37)	5.0%				

Table 2: Slow profiling instrumentation on the hyperSPARC. *Avg. BB Size* is the (dynamic) average basic block size (instructions). *Uninst. Time* is a program's uninstrumented execution time (seconds). Timings were the minimum user and system time from 3 runs on an unloaded system. *Inst. Time* is a program's instrumented, but unscheduled execution time. The number in parentheses is the ratio to the uninstrumented time. *Sched. Time* is the instrumented time after scheduling. Finally, *% Hidden* is the fraction of instrumentation overhead hidden by scheduling.

point is put next in the schedule. An instruction's priority is determined primarily by how few stalls it requires before it can start execution (as computed by pipeline_stalls). If two instructions require the same number of stalls, the instruction farthest from the end of the block, using the metric computed in the first pass, has higher priority. If two instructions still have the same priority, the instruction listed earlier in the unscheduled sequence is chosen under the assumption that the instructions were previously scheduled.

When computing data dependencies in both passes, the scheduler conservatively assumes that loads and stores from the original code access the same address. We also assume that loads and stores in instrumentation code access the same address, which differs from the address accessed by original instructions. This permits instrumentation loads and stores, which typically do not conflict with the original loads and stores, more freedom of movement. Since some instrumentation's memory references are more constrained, we are adding options to limit the movement of instrumentation code.

4.1 Limitations on Scheduling Instrumentation

On aggressive superscalar machines, one could hope that all instrumentation code could be hidden in unused pipeline stall cycles. Unfortunately, processor limitations, such as memory latency (a load on the hyperSPARC has a one cycle latency) and resource usage (stores on the hyperSPARC use the LSU for 2 cycles and loads use it for 1 cycle), limit the cycles in which to hide instrumentation. A further problem is that in many programs, most basic blocks are short and so present few opportunity to hide instrumentation. On the SPARC, the SPECINT benchmarks have average dynamic block sizes of 1.9–3.3 instructions (the floating point benchmarks average 4.1–67.3 instructions, but most are under 7 instructions per block).

Benchmark	Avg. BB Size	SUN Compiler				gcc			
		Uninst. Time	Inst. Time	Sched. Time	% Hidden	Uninst. Time	Inst. Time	Sched. Time	% Hidden
espresso	2.6	8.2	19.8 (2.43)	17.6 (2.16)	19.1%	8.7	19.4 (2.24)	16.8 (1.95)	23.7%
xlisp	2.2	109.9	223.4 (2.03)	201.4 (1.83)	19.4%	120.3	234.3 (1.95)	211.6 (1.76)	20.0%
eqntott	2.0	10.8	26.6 (2.47)	24.5 (2.27)	13.0%	21.5	60.7 (2.82)	53.7 (2.49)	18.0%
compress	3.3	7.8	14.0 (1.80)	12.7 (1.63)	21.8%	7.8	14.0 (1.79)	13.5 (1.72)	8.9%
sc	1.9	47.1	78.2 (1.66)	71.6 (1.52)	21.2%	21.9	42.3 (1.93)	38.9 (1.78)	16.8%
cc1	2.5	9.2	20.4 (2.22)	19.5 (2.13)	7.9%	6.3	15.3 (2.44)	14.6 (2.33)	8.2%
spice2g6	4.1	494.5	779.6 (1.58)	699.6 (1.41)	28.1%				
doduc	5.6	26.6	37.1 (1.40)	36.5 (1.37)	5.7%				
mdljdp2	4.9	8.7	11.8 (1.36)	10.7 (1.24)	35.2%				
wave5	8.0	73.3	98.2 (1.34)	94.4 (1.29)	15.1%				
tomcatv	19.9	34.0	38.5 (1.13)	37.5 (1.10)	22.5%				
ora	4.5	48.5	58.5 (1.20)	54.6 (1.12)	38.9%				
alvinn	6.8	160.2	233.9 (1.46)	198.7 (1.24)	47.7%				
ear	5.8	479.5	647.7 (1.35)	583.1 (1.22)	38.4%				
mdljsp2	4.6	7.8	10.9 (1.39)	9.8 (1.25)	36.1%				
swm256	67.3	284.3	289.8 (1.02)	295.7 (1.04)	-108.7%				
su2cor	12.1	144.6	165.3 (1.14)	162.9 (1.13)	11.7%				
hydro2d	4.8	162.7	238.4 (1.47)	213.9 (1.32)	32.3%				
nasa7	15.0	270.9	297.0 (1.10)	293.7 (1.08)	12.7%				
fpppp	12.9	103.9	121.4 (1.17)	117.9 (1.13)	19.8%				

Table 3: Slow profiling instrumentation on the SuperSPARC.

Finally, scheduling instrumentation does not reduce instruction (or data) cache misses caused by instrumentation, since the additional instructions increase the code size regardless of how few stalls the program incurs. Lebeck and Wood proposed a model for the instruction cache effects of program instrumentation, which reasonably accurately predicted that instrumentation that increases a program's size by a factor of E, will increase cache misses by $E \times \sqrt{E}$ [10]. Profiling increases a program's text size by a factor of 2–3. Fortunately, many programs have low instruction cache miss rates, so the increase is not significant.

4.2 Scheduling Profiling Instrumentation

We scheduled QPT2's slow profiling instrumentation [1], which adds 4 instructions—set immediate, load, add, and store—into most basic blocks in a program. This code executes in 3 cycles on the hyperSPARC and 4 cycles on the SuperSPARC and UltraSPARC. Blocks with a single instrumented single-exit predecessor or a single instrumented single-entry successor are not instrumented. The SuperSPARC and hyperSPARC experiments ran on dual processor Sun SPARCstation 20s equipped with 55Mhz SUN SuperSPARC [16] processors and 66Mhz ROSS hyperSPARC processors [13]. Both systems ran Solaris 2.4. The UltraSPARC experiments ran on a SPARCstation 140 with a 143Mhz SUN UltraSPARC processor [15] running Solaris 2.5. The test programs were compiled -O (not at SPEC optimization levels) by the Sun C and Fortran compilers (version 3.0.1 for the older machines and version 4.0 for the UltraSPARC). We did not use the compiler options that generate UltraSPARC-specific code. In addition, as a comparison, we compiled the SPECINT benchmarks with gcc (version 2.6.3) with the -O flag. In all cases, we ran the programs with the single SPEC input file that produced the longest execution.

Table 2 contains measurements for the hyperSPARC, Table 3 contains results for the SuperSPARC, and Table 4 contains results for the UltraSPARC. On all three machines, several trends are clear. Scheduling is less effective for integer programs than for floating-point programs (average improvement of 17%, 17%, and 16%, respectively for integer benchmarks verses 30%, 26%, and 60%, respectively for floating point benchmarks). We see three reasons for this difference. First, integer benchmarks have significantly shorter basic blocks (integer blocks average 2.4 instructions and floating point blocks average 12.6 instructions).

Benchmark	BB Size	Uninst. Time	Inst. Time	Sched. Time	% Hidden
espresso		3.4	7.0 (2.06)	6.3 (1.85)	20.1%
xlisp		46.1	97.1 (2.01)	93.2 (2.02)	7.7%
eqntott		5.0	9.8 (1.96)	9.0 (1.81)	15.7%
compress		2.3	4.2 (1.83)	3.9 (1.69)	16.8%
sc		11.1	20.9 (1.88)	18.8 (1.70)	21.2%
cc1		3.0	6.6 (2.20)	6.2 (2.06)	11.7%
spice2g6		191.8	284.2 (1.48)	252.8 (1.32)	34.0%
doduc		12.3	16.4 (1.33)	16.2 (1.31)	6.6%
mdljdp2		2.8	3.7 (1.31)	3.3 (1.18)	42.5%
wave5		27.5	34.1 (1.24)	32.2 (1.17)	29.4%
tomcatv		13.2	14.5 (1.10)	13.5 (1.03)	74.1%
ora		21.2	25.9 (1.22)	24.1 (1.13)	38.3%
alvinn		30.3	55.9 (1.84)	50.2 (1.65)	22.4%
ear		127.2	175.8 (1.38)	155.3 (1.22)	42.2%
mdljsp2		2.1	66.5 (31.09)	66.5 (31.05)	0.1%
swm256		77.8	80.8 (1.04)	76.7 (0.99)	135.6%
su2cor		51.9	58.0 (1.12)	50.5 (0.97)	122.4%
hydro2d		64.0	89.3 (1.40)	76.7 (1.20)	49.8%
nasa7		125.9	134.4 (1.07)	125.0 (0.99)	110.9%
fpppp		54.8	59.5 (1.08)	53.4 (0.97)	131.7%

Table 4: Slow profiling instrumentation on the UltraSPARC.

Since our scheduler was local and did not look between block boundaries, shorter blocks offer fewer opportunities to hide instrumentation overhead. However, performance improvement did not appear correlated to average block size (correlation coefficient of 0.034 for hyperSPARC and -0.13 for SuperSPARC).

In addition, the SuperSPARC and UltraSPARC both issue at most 2 integer instructions per cycle, so for integer codes they do not offer more parallelism than the 2-way superscalar hyperSPARC. The profiling instrumentation executed entirely in the integer unit. In theory, a 2-way superscalar could hide this instrumentation's overhead. However, small blocks, resource constraints, and instruction parallelism in the original program make this outcome unlikely. In particular, conflicts for scarce resources, such as the single load/store unit in all processors, hinder us from overlapping instrumentation with integer instructions in both sets of benchmarks.

By contrast, floating point instructions, on all three machines, execute in an autonomous FPU. Not only could we easily schedule these instructions with our integer instrumentation without resource conflicts, but the long latency floating point instructions easily overlap the entire profiling code sequence.

Nevertheless, the results are very encouraging. On a first-generation superscalar, such as the dual-issue hyperSPARC, scheduling hides 13–21% of the overhead in profiling the SPECINT benchmarks and 5–53% of the overhead in the SPECFP benchmarks.[1] The results on the SuperSPARC are similar, since scheduling hides 8–22% of the SPECINT overhead and 6–48% of the SPECFP overhead (again, excluding swm256). More aggressive superscalar implementation, such as the UltraSPARC, can hide even more instrumentation overhead, ranging from 8–21% of the SPECINT overhead and 7–136% of the SPECFP overhead[2]. An interesting sidelight is that four SPECFP benchmarks ran faster with instrumentation, thanks to EEL's instruction scheduling.

1. With the exception of swm256, whose behavior on both processors seems to indicate problems in our scheduler's handling of the very large basic block in which this program spends most of its time. We are studying this problem.

2. Excluding mdljsp2. Instrumented versions of mdljsp2 on the UltraSPARC seem to spend their time in the operating system. We are studying this problem.

Benchmark	hyperSPARC			UltraSPARC			
	Uninst. Time	Re-Sched. Time	% Speedup	Uninst. Time	Re-Sched. Time	% Speedup	4.0 Time
espresso	8.1	8.0	1.2%	3.2	3.1	2.5%	3.4
xlisp	118.1	117.4	0.6%	48.5	49.4	-1.8%	46.1
eqntott	10.5	10.6	-0.9%	5.1	5.0	1.0%	5.0
compress	8.7	8.6	1.4%	2.1	2.1	3.3%	2.3
sc	47.9	47.8	0.3%	26.3	26.3	-0.3%	11.1
cc1	10.6	10.5	0.7%	3.0	2.9	2.7%	3.0
spice2g6	510.3	507.8	0.5%	209.8	207.0	1.3%	191.8
doduc	28.3	28.6	-1.0%	12.9	11.0	14.8%	12.3
mdljdp2	6.6	6.6	0.0%	2.4	2.3	4.6%	2.8
wave5	79.0	78.8	0.3%	29.5	27.7	6.1%	27.5
tomcatv	42.5	43.7	-2.9%	11.6	10.7	7.7%	13.2
ora	45.9	45.6	0.6%	20.6	19.9	2.9%	21.2
alvinn	132.6	125.7	5.2%	50.5	50.0	0.9%	30.3
ear	433.4	434.2	-0.2%	190.6	171.8	9.9%	127.2
mdljsp2	5.7	5.7	-1.2%	2.1	2.1	2.8%	2.1
swm256	223.8	228.8	-2.2%	87.9	83.1	5.5%	77.8
su2cor	117.5	116.6	0.8%	49.2	42.7	13.3%	51.9
hydro2d	147.5	151.4	-2.7%	58.1	51.5	11.3%	64.0
nasa7	318.0	317.2	0.2%	115.9	106.2	8.4%	125.9
fpppp	116.1	118.0	-1.6%	52.8	35.4	32.8%	54.8

Table 5: Rescheduling SuperSPARC program on the hyperSPARC and UltraSPARC. These programs were compiled by the Sun compilers (v. 3.0.1) for a SuperSPARC. *Uninst. Time* is a program's uninstrumented execution time (seconds). *Re-Sched. Time* is a program's time after rescheduling for the respective machine. *% Speedup* is the improvement due to rescheduling. *4.0 Time* is the execution time (seconds) on the UltraSPARC for programs compiled -0 by the v. 4.0 UltraSPARC compiler (without UltraSPARC-specific optimizations).

4.3 Instruction Rescheduling

In another experiment, we measured the effect of rescheduling uninstrumented SPEC92 benchmarks on the same SPARC platforms as above. The original code was compiled at optimization level -O by the Sun compilers for the SuperSPARC (v. 3.0.1) and then rescheduled by EEL.

Table 5 contains measurements of the SPEC92 benchmarks rescheduled by EEL for the hyperSPARC and UltraSPARC, running on hyperSPARC and UltraSPARC, respectively. Not surprisingly, rescheduling SuperSPARC programs for a similar processor, such as the hyperSPARC, produced little performance improvement. Rescheduling SPECINT benchmarks for the UltraSPARC also yielded little improvement. As noted above, the hyperSPARC, SuperSPARC, and UltraSPARC have similar microarchitectures for purely integer codes and present few opportunities for rescheduling to improve performance.

Only the SPECFP benchmarks on the UltraSPARC show consistent improvement. On this platform, every floating point program ran faster, with an average speedup of 9% (1–33%). Most of this improvement is due to modeling the UltraSPARC's out-of-order execution on level-1 cache misses. Rescheduled code on the UltraSPARC only showed significant improvement when the scheduler assumed a 7 cycle load latency (cost of a level-2 cache hit). The small block size in the integer benchmarks precluded most rescheduling for this long latency.

As expected, a scheduling algorithm cannot improve a good compiler's instruction scheduling for two machines with similar microarchitectures. Instruction rescheduling, however, can produce significant performance improvements by exploiting previously unavailable features, such as out-of-order execution. In

the future, rescheduling will become even more attractive, as superscalar processors become more aggressive and offer new features, such as out-of-order execution, lockup-free caches, and speculative execution, that can be exploited by rescheduling legacy code.

5 Conclusion

This paper investigated the benefits of combining instruction scheduling with executable editing. Measurements on the SPEC92 benchmarks show that on a dual-issue superscalar processor, a simple, local instruction scheduler can hide an average of 17–30% of the overhead introduced by program profiling instrumentation. On a more modern, 4-way superscalar, the scheduler can hide an average of 16–65% of the profiling overhead. As future machines offer more abundant, less restrictive instruction-level parallelism, it should be possible to hide even more instrumentation overhead. Already, the benefits of scheduling program instrumentation are clear enough that existing and future instrumentation systems should adopt this simple technique to reduce instrumentation overhead. In addition, this approach promises to help reduce the cost of error checking, such as array bounds or null pointer tests, to a level at which it is routinely included in production code.

The results for our second application, instruction rescheduling, are also encouraging. Local rescheduling of optimized code offers little benefit when a new machine's microarchitecture is similar to the machine for which a program was originally compiled. However, when the two microarchitectures differ, rescheduling can significantly improve the performance of legacy code. Moreover, we have not yet investigated the possibility of optimizing the code by introducing new instructions, such as conditional moves or integer multiplies.

Acknowledgments

Many thanks to Bob Roessler for arranging the loan of the UltraSPARC used in these experiments. Mark Hill suggested that our scheduler model the L2 cache in the UltraSPARC. Vinod Grover and Kurt Goebel provided helpful comments on a draft of this paper.

This work 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, an NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation and Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, July 1994, pages 1319–1360.

[2] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of the Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[3] Trung A. Diep, Christopher Nelson, and John Paul Shen. Performance Evaluation of the PowerPC 620 Microarchitecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 163–174, June 1995.

[4] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2):9–15, February 16 1995.

[5] John C. Gyllenhaal. A Machine Description Language for Compilation. Master's thesis, Department of Electrical Engineering, University of Illinois, Urbana IL, September 1994.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[7] Daniel R. Kerns and Susan J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, June 1993.

[8] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[9] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[10] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 220–230, May 1995.

[11] Harish Patil and Charles Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*, St. Malo, France, May 1995.

[12] Todd A. Proebsting and Christopher W. Fraser. Detecting Pipeline Structural Hazards Quickly. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 280–286, Portland, Oregon, January 1994.

[13] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[14] Michael A. Schuette. Exploitation of Instruction-Level Parallelism for Detection of Processor Execution Errors. Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA, January 1991.

[15] SUN Microsystems, Inc. *UltraSPARC-I User's Manual*, August 1995.

[16] Texas Instruments. *SuperSPARC User's Guide*, October 1993.

Appendix A: Function pipeline_stalls

```
unsigned long
pipeline_stalls(unsigned long cycle,  // cycle when mi starts executing
                UnitValues &state,    // current pipeline state
                const mach_inst* mi)  // next instruction
{
    unsigned long stalls = 0;

  {{INST mi CATEGORY any::
    // All Spawn annotations now refer to instruction mi.

    unsigned long gid = {{GROUP}};    // mi's timing group
    long ii;

    // Trace[] records the resources used by
    // this instruction in the current cycle.
    unsigned long trace[{{UNITS COUNT}}];
    for(ii=0; ii<{{UNITS COUNT}}; ++ii) trace[ii] = 0;

    // Search for stalls
    unsigned long mi_cycle = 0;   // current cycle in mi's pipeline
    while(mi_cycle <= {{GRP gid CYCLES}}) {

        // Units[] records the number of unused resources in this cycle
        // after allocating resources for all previous instructions.
        unsigned long* units = state[cycle];
        bool advance = true;

        // Test for structural hazzards.
        if(advance)
            for(ii=0; ii<{{GRP gid ACQUIRE mi_cycle COUNT}}; ++ii) {
                unsigned long unit_val =
                    units[{{GRP gid ACQUIRE mi_cycle UNIT ii}}] -
                    trace[{{GRP gid ACQUIRE mi_cycle UNIT ii}}];
                if(unit_val < {{GRP gid ACQUIRE mi_cycle NUM ii}}) {
                    advance = false;
                    break;
                }
           }

        // Test for RAW hazzards.
        if(advance)
            for(ii=0; ii<{{R READ COUNT}}; ++ii)
                if({{R READ ii TIME}} == mi_cycle &&
                   cycle < state.write_cy[0][{{R READ ii}}]) {
                    advance = false;
                    break;
                }

        // Similar tests for WAR and WAW hazzards.
        ...

        // Advance the execution pipeline for
        // previously scheduled instructions.
        ++cycle;

        // Advance instruction pipeline or record the stall
        if(advance) {
            for(ii=0; ii<{{GRP gid ACQUIRE mi_cycle COUNT}}; ++ii)
                trace[{{GRP gid ACQUIRE mi_cycle UNIT ii}}]
                    += {{GRP gid ACQUIRE mi_cycle NUM ii}};
            ++mi_cycle;
            for(ii=0; ii<{{GRP gid RELEASE mi_cycle COUNT}}; ++ii)
                trace[{{GRP gid RELEASE mi_cycle UNIT ii}}]
                    -= {{GRP gid RELEASE mi_cycle NUM ii}};
        } else
            ++stalls;
    };;
  }}

    return stalls;
}
```