MASTER/SLAVE SPECULATIVE PARALLELIZATION

AND APPROXIMATE CODE

by

Craig B. Zilles

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2002

Abstract

This dissertation describes *Master/Slave Speculative Parallelization* (MSSP), a novel execution paradigm to improve the execution rate of sequential programs by parallelizing them speculatively for execution on a multiprocessor. In MSSP, one processor—the *master*—executes an *approximate* copy of the program to compute values the program's execution is expected to compute. The master's results are then checked by the *slave* processors by comparing them to the results computed by the original program. This validation is parallelized by cutting the program's execution into tasks. Each slave uses its predicted inputs (as computed by the master) to validate the input predictions of the next task, inductively validating the whole execution.

Approximate code, because it has no correctness requirements—in essence it is a software value predictor—can be optimized more effectively than traditionally generated code. It is free to sacrifice correctness in the uncommon case in order to maximize performance in the common case. In addition to introducing the notion of approximate code, this dissertation describes a prototype implementation of a program distiller that uses profile information to automatically generate approximate code. The distiller first applies unsafe transformations to remove uncommon case behaviors that are preventing optimization; second, it applies traditional safe optimizations to exploit the newly created opportunities.

The mechanisms necessary for a MSSP execution and an example implementation based on a chip multiprocessor are also described. These mechanisms buffer the master's predictions and make them available to the slaves, capture the input and output values for each slave task, and verify and commit the slave tasks to give the appearance of a sequential execution. A hardware implementation of these mechanisms require only a modest amount of resources (likely less than 5% area) beyond a next-generation chip multiprocessor.

This dissertation includes a simulation-based evaluation of the example MSSP implementation. Performance results show that MSSP achieves speedups up to 1.75 (harmonic mean 1.25) for the SPEC2000 integer benchmarks. Performance is currently limited by the effectiveness of code approximation, which can likely be significantly improved.

Acknowledgments

Before I selected Guri to be my adviser, I asked him what was involved in doing a Ph.D. His response was “you have to go through hell.” Although perhaps a slight exaggeration, there is no doubt that the worst times are made bearable and the best times genuinely memorable by the people in our lives.

First and foremost in this regard is family. My deepest thanks go to my wife Julie and my son Brian for being loving and supportive throughout the whole process. You two are my life. I thank also my parents Stephen and Connie, my brother Karl, and Julie’s parents George and Jan for their constant encouragement and understanding.

I thank Guri for helping me find important and challenging problems to solve and the perspective to develop the solutions. There is no measure to the extent he has shaped my technical thinking; he has a unique ability to seek out what is critical. I also appreciate the freedom he has given me to develop my own vision and the self confidence to follow it.

I’d also like to thank the other members of my committees--Ras Bodik, Charlie Fischer, Jim Goodman, Mark Hill, and Jim Smith--for their contributions to my personal development and the development of this dissertation. In particular, I’d like to thank Ras for his constructive criticism and encouragement and his effort to push me to formalize my thinking.

Much of my development as a researcher is the result of the academic community to which I belonged. The architecture group at Wisconsin is at once technically ruthless and personally supportive. The faculty should be congratulated on the environment they have created. I have benefited from many of the students of this community, but a few of them deserve special acknowledgment.

I want to thank Eric Rotenberg for jump starting my involvement in architecture by engaging me in full-contact technical discussions in my first term at Wisconsin. I also benefited in these early years from discussions with Andy Glew, Quinn Jacobsen, Timothy Heil, S. Subramanya Sastry, and the members of the Kestrel project. In addition, I want to thank Andreas Moshovos for advising me on the workings of graduate school.

In my later years, it was Adam Butts, Brian Fields, Milo Martin, Ravi Rajwar, Amir Roth, and Dan Sorin who were my “go-to guys.” They helped me refine my thinking, writing, and presentation skills. In addition, Milo Martin, my office-mate of 5 years, both tolerated my odious personal habits and taught me most everything I know about multiprocessors. I like you, Milo, even if nobody else does. Amir Roth deserves my special thanks for helping me to organize my writing and not pushing me off the chair lift.

Lastly, I would like to acknowlege the organizations that made this research possible. I thank the National Science Foundation, Intel Corporation, and Cisco Systems for the fellowships that funded much of my graduate studies and addition support from NSF grants CCR-9900584 and EIA-0071924. In addition, thanks go to the Condor project and the Computer Systems Lab at Wisconsin for providing the infrastructure for doing my research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exponential growth of processor performance has enabled single-processor computers to achieve execution rates in the billions of operations per second and multiprocessor computers capable of hundreds of billions or even trillions of operations per second. Despite these achievements, there remains a desire to have even faster computers that will enable additional software functionality and achieve these performance levels in smaller, cheaper form factors that consume less power.

Advances in semiconductor technology continues to provide the raw materials for continued performance growth, providing larger chips capable of holding more and faster transistors. There remains a challenge though, in how to utilize these resources in a manner that improves performance. Extrapolation of previously used techniques (*e.g.*, naively increasing superscalar width) does not "scale up" efficiently due to clock cycle constraints [57] and true dependences—both control and data—in programs, especially non-numeric programs. Benefitting fully from larger transistor budgets necessitates the development of new execution paradigms to enhance and exploit parallelism. In this dissertation, I propose one such paradigm, Master/Slave Speculative Parallelization (MSSP).

1.1 Constraints Guiding the Development of the Execution Paradigm

Before describing the MSSP paradigm in detail, it is illuminating to discuss the constraints, both physical and social, that I perceived would affect the adoption of a new execution paradigm. These constraints shaped the development of the MSSP paradigm. Five constraints—wire delay, programmer productivity, infrastructure transparency, minimal verification complexity, and flexibility—are described in the following paragraphs.

Wire delay: As clock frequencies increase, the distance a signal can travel in a single cycle decreases. With slow global communication, de-centralized microarchitectures consisting of many loosely-coupled regions are desirable. Designing such architectures can be difficult because partitioning programs so they can be efficiently mapped onto such hardware is a challenging, open problem. Control and data dependences between operations necessitates enforcing some ordering constraints on the operations and facilitating communication between them.

Thesis: A new execution paradigm should provide an efficient means of partitioning programs for mapping onto decentralized microarchitectures.

Programmer Productivity: In the early days of computing, computers were so few in number and so expensive that it was appropriate for programmers to hand optimize code to achieve the most benefit from a limited resource. Declining component costs have made computers ubiquitous and cheap, making programmers the limiting resource. Modern computer designers cannot rely on programmers to hand optimize their code—including techniques like manual parallelization—except in performance critical domains (*e.g.*, database server applications). Most programmers are likely to program in a way that maximizes their productivity by using high-level languages and methodologies that favor clarity, verifiability, reusability, and maintainability over performance.

Thesis: A new execution paradigm should handle compiled code well and be tolerant of the manner a computation is expressed. In the limit, the intrinsic complexity of computation should determine execution time, not the manner in which it is expressed.

Infrastructure Transparency: Widespread adoption of any technology is slowed if its deployment requires outside entities to change. For processor design teams, such outside entities include compiler vendors, operating system vendors, programmers, and library writers. Requiring changes in the compiler or that programmers change their methodology can take a long time and can be risky. It is much preferable if technology can be introduced in such a manner that it is transparent to the existing infrastructure.

Thesis: A new execution paradigm should be language agnostic, require few, if any, changes to the compiler or operating system, and, therefore, handle legacy code.

Minimal Verification Complexity: Much of the cost/effort/time in the development of a modern microprocessor is dedicated to verification. As such, it is desirable to develop micro-architectural techniques that reduce verification complexity or at worst increase it minimally.

Thesis: A new execution paradigm should minimize any additional verification complexity.

Flexibility: The above constraints have expounded on the importance of not *requiring* changes to compiler infrastructure or programmer behavior, but it is desirable that if beneficial changes are made they can be exploited. For example, if code is manually parallelized or hand assembled to expose a lot of explicit parallelism, the machine can provide the necessary resources. Thus it is desirable for the micro-architecture to be largely composed of general purpose execution resources (processors, functional units, caches, etc.) rather than special-purpose widgets that will be idle when an alternative execution paradigm is used.

Thesis: A new execution paradigm should require a minimum of special purpose widgets.

In the next subsection, I briefly overview the proposed execution paradigm and how it addresses these perceived constraints.

1.2 Overview of Master/Slave Speculative Parallelism

This dissertation describes the Master/Slave Speculative Parallelization (MSSP) paradigm. This paradigm enables the automatic parallelization of sequential programs, including legacy binaries. This parallelization is effected by dedicating one processor to be the "master[1]," who orchestrates the parallel execution by (1) instructing each "slave" processor where (*i.e.*, at what program counter value) to begin executing and (2) providing each slave processor with the set of live-in values needed to execute the assigned segment (or "task") of the program.

1. The use of the word "master" comes from the definition "a device or mechanism that controls the operation of another" and not from the connotation of a master copy from which other copies are made (*e.g.*, phonograph records).

The master processor—to which I will refer simply as the master—performs its duties by executing what I call a distilled program. The distilled program is an "approximate" version of the original sequential program that has been augmented with directives for managing the slave processors. The distilled program's approximate nature allows it to execute more quickly than the original program, but it may occasionally diverge from a correct execution. By coupling the master's execution of the distilled program with the slaves' execution of the original program, MSSP can potentially achieve the superior performance of the distilled program with the correctness of the original program.

I briefly overview the MSSP paradigm in Figure 1.1. Four processors are shown; one (**P0**) is allocated to be the master processor, and the rest (**P1**, **P2**, and **P3**) are slaves that begin the example idle. By executing the distilled program, the master processor performs many of the register and memory writes that the original program would, but these writes are not committed. Instead, these writes are used to create a predicted checkpoint of program state to be used by the slave processors and are held in a special "checkpoint" buffer. At task boundaries in the distilled program there are fork instructions that signal the master to spawn a new task in the original program on a free processor and to provide it access to the buffered checkpoint values. At the annotation **1** in the figure, the master processor spawns **Task B** onto processor **P2**. **P2** begins executing the task after some latency due to the inter-processor communication (**2**). **P0** continues executing (**3**) the distilled program segment that corresponds to **Task B**, which I refer to as **Task B´**.

As the slave **Task B** executes, it will read values that it did not write (live-in values) and perform writes of its own (live-out values). If a corresponding checkpoint value is available, it is used for the live-in value; otherwise the value is read from current visible (architected) state. As the task is speculative—it is using

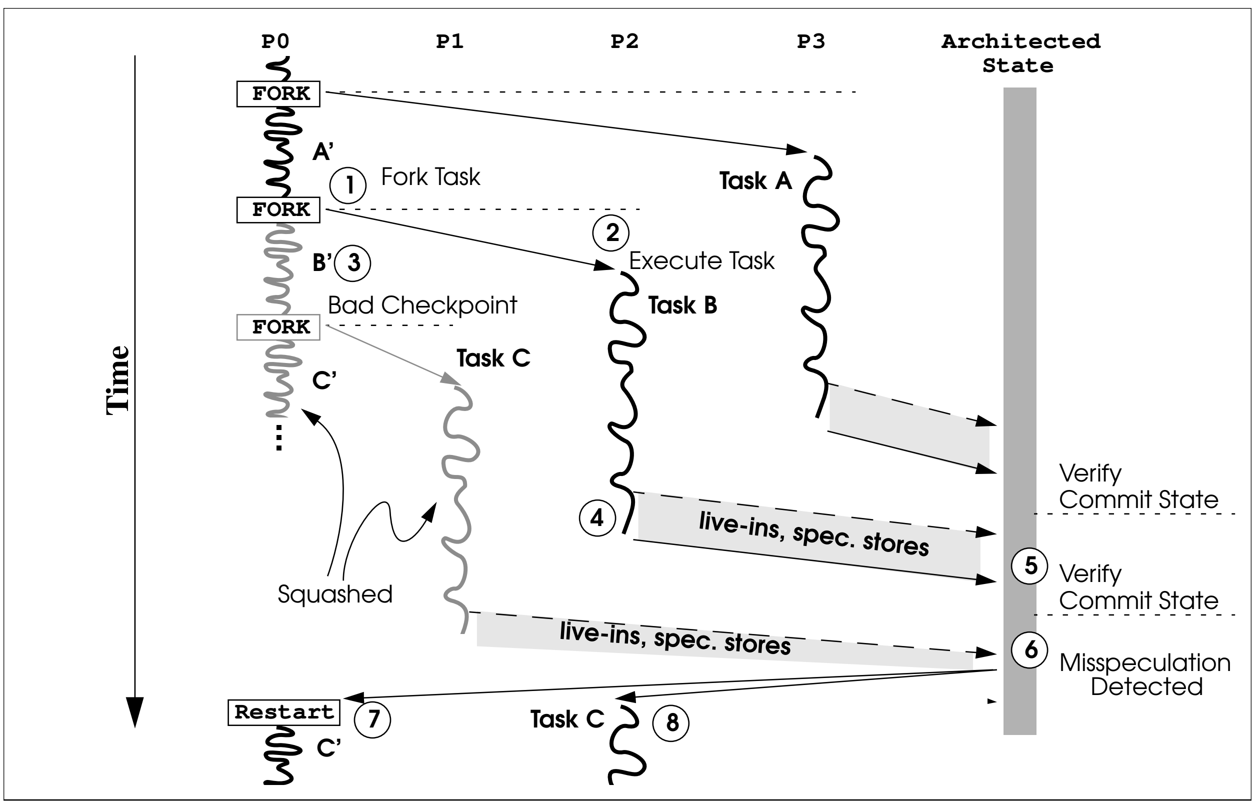


Figure 1.1: Master processor distributes checkpoints to slaves. *The master—executing the distilled program on processor P0—assigns tasks to slave processors, providing them with predicted live-in values in the form of checkpoints. The live-in values are verified when the previous task retires. Misspeculations, due to incorrect checkpoints, cause the master to be restarted with the correct architected state.*

predicted live-in values—its live-out values cannot be immediately committed. Instead its live-in and live-out values are recorded in live-in and live-out buffers and associated with the task. When the task is complete (**4**), **P2** sends its live-in and live-out values to the authority on architected state. If the recorded live-in values exactly correspond to architected state, then the task has been verified and can be committed, and architected state can be updated (**5**) with the task's live-out values.

If one of the recorded live-in values differs from the corresponding value in the architect state (*e.g.*, because the master wrote an incorrect value (**3**)), this mismatch will be detected during verification. On detection of the misspeculation (**6**), the master is squashed, as are all other in-flight tasks. At this time, the master is restarted at **C´** (**7**), using the current architected state. In parallel, non-speculative execution of the corresponding task in the original program (**Task C**) begins (**8**).

1.3 Contributions of this Dissertation

In this dissertation, I focus on three topics: the Master/Slave Speculative Parallelization paradigm, the abstract concept of approximate code, and distilled programs, the specific form of approximate code used in MSSP. In addition, with regards to the constraints discussed in Section 1.1, I demonstrate that the MSSP paradigm has the following characteristics:

1. By parallelizing the program into tasks and predicting their live-in values, MSSP effectively creates independent units of work—generally consisting of hundreds of dynamic instructions—that can be mapped onto a distributed micro-architecture (*e.g.*, a chip multi-processor). If live-in predictions are accurate, the MSSP paradigm can be very tolerant of communication latency.
2. In approximating a program, the distilled program is free to re-express computation in a manner that facilitates fast execution. Because the distilled program need not be correct in all circumstances, it is sufficient to find an expression that is correct for the "common case", potentially enabling more aggressive optimizations.
3. No compiler modifications are required and legacy binaries can be supported. All information needed to construct the distilled program can be derived from the original program's static image and profile information. Thus, transparent implementations that perform the distillation at run time are feasible.
4. The program distiller, the mechanism that constructs the distilled program, need not be verified. The unmodified original program is executed (by the slaves) and remains the authority for correct execution. All results computed by the distilled program are predictions, untrusted by the machine until verified against architected state. Thus, any bugs in the distiller translate into mispredictions, which could reduce performance but will not affect correctness.
5. The MSSP paradigm can be implemented as an extension to a chip multiprocessor (CMP). By implementing the distiller in software, the required MSSP-specific hardware mechanisms are modest. For example, about 4kB of storage is required at each processor (3% of the size of the caches on a processor with 64kB instruction and data caches) and around 24kB of storage at the L2 (or a little more than 1% of a 2MB L2 cache). These modest hardware requirements can also limit the incremental verification complexity beyond the CMP designs already present in industry. Furthermore, it allows processor vendors to produce a single chip that efficiently handles sequential programs (using MSSP) and parallel programs (using CMP).

In order to justify and explore the implications of the above statements, this dissertation describes the Master/Slave Speculative Parallelization paradigm. Specifically, in this dissertation, I make the following contributions:

1. I define and describe the Master/Slave Speculative Parallelization paradigm, comparing it to previously proposed execution paradigms.

2. I demonstrate the opportunity for approximating programs, discussing a prototype implementation of a program distiller.
3. I describe the necessary mechanisms for the MSSP paradigm and outline a possible implementation.
4. I characterize the distilled programs generated by my prototype program distiller and empirically evaluate the performance of an implementation MSSP.

1.4 Organization of the Dissertation

The organization of this dissertation corresponds directly to the above contributions, where each topic is the focus of a separate chapter (Chapters 2 through 5). I conclude in Chapter 6 by describing some lessons learned during this work, the correctness requirements of the implementations, and outline some unanswered questions to guide future work.

Chapter 2

The Master/Slave Speculative Parallelization Paradigm

In this chapter, I define and present the Master/Slave Speculative Parallelization (MSSP) paradigm. As this is a paradigm for automatic parallelization of sequential programs, I begin the chapter (in Section 2.1) with a description of programs and program parallelization in general. In Section 2.2, I present the MSSP paradigm and introduce the concept of approximate code, including definitions and examples of each. Lastly, in Section 2.3, I compare and contrast the MSSP paradigm to previously proposed execution paradigms.

2.1 Programs and Parallelization

This section begins with a brief description of the components that make up programs—operations and inter-operation dependences—and introduces the *execution dependence graph* (EDG), a tool for visualizing the important dependences. In Section 2.1.2, I demonstrate how parallelizing programs involves partitioning EDG's into tasks. In general, such a partitioning results in dependences that cross task boundaries; in Section 2.1.3, I describe the three ways that such dependences can be dealt with. Section 2.1.4 closes this section with a discussion of two mechanisms for verification if these dependences are handled speculatively.

2.1.1 Programs as Operations and Dependences

This dissertation is concerned with the automatic parallelization of programs written with a sequential execution paradigm in mind. These programs consist of operations that name (either explicitly or implicitly) the storage location of their source and destination operands. Inter-operation communication is performed by matching up the producer of the value at a given storage location to the consumer of that location and results in a dependence between the two operations.

In this work, I concern myself with two classes of inter-operation dependences: true data dependences and control dependences. Anti- and output dependences can be easily alleviated by renaming storage locations and in-order retirement, respectively, so they need not be a concern.

True data dependences ensure that storage locations named by source operand contain the correct values at the time an operation is executed. For the purposes of describing parallelization paradigms, dependences through registers and memory are effectively equivalent, so I will not make this distinction.

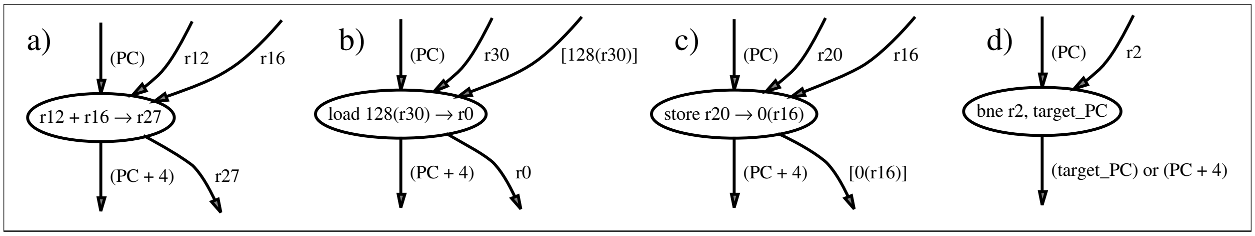


Figure 2.1: Examples of register, memory, and control dependences on four classes of instructions. *(a) arithmetic, (b) load, (c) store, and (d) control.*

Control dependences determine which operations are executed and, as a result, which data dependences are realized. For simplicity, I will model control dependences as data dependences by adding implicit reads and writes to the program counter (PC) for each instruction. This introduces a serial data dependence from each instruction to the next. Note that, in general, this over-constrains the inter-operation dependences, but results in little loss of generality for the discussion of program parallelization to follow. Figure 2.1 shows example operations and their dependences.

Different executions (dynamic instances) of the same (static) instruction can potentially have different dependences due to different paths through the computation or values computed upon. For a given execution of a program, I can create a graph of the dependences that were realized. This Execution Dependence Graph (EDG) is an execution trace augmented with arcs indicating the inter-operation dependences; an example EDG snippet is shown in Figure 2.2. **The EDG contains the set of dependences that need to have been observed to have correctly performed the program's execution.** The EDG differs from a control-flow graph (CFG) and a data-flow graph (DFG) in that it is concerned with what *did* happen on a particular execution, not what *could* happen on across all possible executions.

2.1.2 Parallelizing Programs: Partitioning into Tasks

Although there are many levels at which parallelism can be exploited (*e.g.*, word, instruction, thread), in this dissertation I focus on thread-level parallelism (TLP). Techniques to exploit word-level parallelism (WLP) and instruction-level parallelism (ILP) have been actively researched and these forms of parallelism are being effectively exploited in existing processors (by SIMD instruction set extensions and superscalar/ VLIW architectures, respectively). TLP, on the other hand, is not being exploited in most programs due to the high cost of manual parallelization, so there is significant untapped performance potential. Furthermore, because TLP exploits parallelism at a larger granularity than WLP or ILP techniques, performance benefits of exploiting TLP will likely complement performance achieved through those techniques.

To achieve a thread-level parallel execution for a computation, I will partition it into *tasks* that can be performed in parallel by independent processors. In this work, the program is partitioned into tasks by specifying *task boundaries* with static annotations to the program, much like what was done in Multiscalar [72, 81][1]. The tasks that result from these task boundaries depend on the control flow path the program follows. Figure 2.3 shows a control flow graph (CFG) of a computation with a set of task boundaries and the set of possible tasks that could arise from the execution of the computation.

> ***Task Boundary:*** *a point in the static program, generally associated with the static instructions that it precedes, that terminates a task when it is encountered during execution.* **(Definition 1)**

> ***Task:*** *a continuous sequence of operations in the program's execution bounded at each end by a task boundary annotation in the static program, with no task boundary annotations in the task's interior.* **(Definition 2)**

---

1. The difference between this definition and the one used for Multiscalar task selection [72, 81] is that there is no constraint that the tasks have a single entry.

a)

```
0x120012530:
lda     sp, -16(sp)
ldq     t12, -32528(gp)
stq     ra, 0(sp)
stq     s0, 8(sp)
move    a0, s0
ldah    a0, -8193(gp)
move    s0, a1
ldl     t1, -26816(gp)
lda     a0, -3624(a0)
cmple   t1, 0x4, t1
bne     t1, 0x12001256c

0x12001256c:
cmple   s0, 0x3, v0
move    s0, a2
```

register dependence
memory dependence

b)

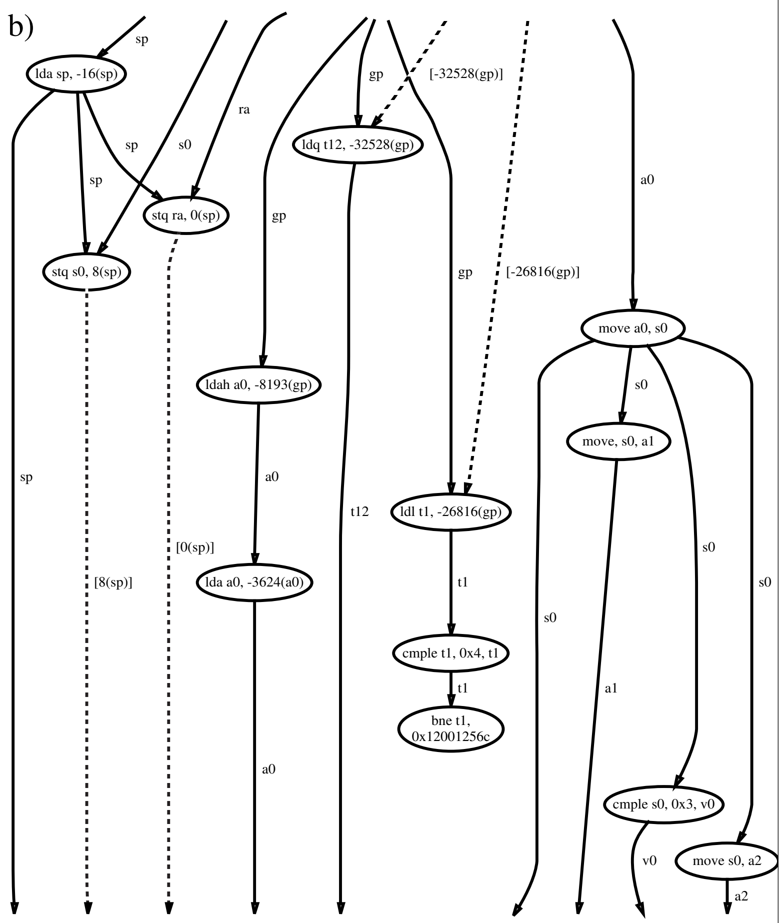


Figure 2.2: Example Execution Dependence Graph (EDG). *(a) code snippet from* `bzip2`*, and (b) the associated EDG (control dependences are not shown for clarity).*

Generally, the program's execution requires the execution of a large number of dynamic tasks, but only a small number of them are *in flight* (*i.e.*, actively being executed) at any one time. As shown in Figure 2.4, a program's execution can be divided into 3 segments: 1) "the past": a segment consisting entirely of completed tasks starting at the beginning of the execution, 2) "the present": a segment of in-flight tasks[2], and 3) "the future" a segment consisting of un-initiated tasks and ending at the program's completion. I'll refer to the second (middle) segment as the *task window* because it effectively moves like an instruction window down the execution.

In partitioning a program into tasks, I am making horizontal cuts through the EDG, effectively selecting a set of dependency edges to become *inter-task dependences*. I will refer to the set of inter-task dependences whose consumers belong to a given task as being *live-in* to the task; a task's live-in values are those that it reads before writing. An example that classifies dependences is shown in Figure 2.5. I will refer to the state of all variables at a task boundary as a *checkpoint*; a checkpoint from the immediately proceeding task boundary is sufficient to satisfy all live-in dependences of a task.

2. Strictly speaking it is possible to have completed and un-initiated tasks in the "present" segment. More exactly this segment spans from the first in-flight or un-initiated task to the last in-flight or completed task.

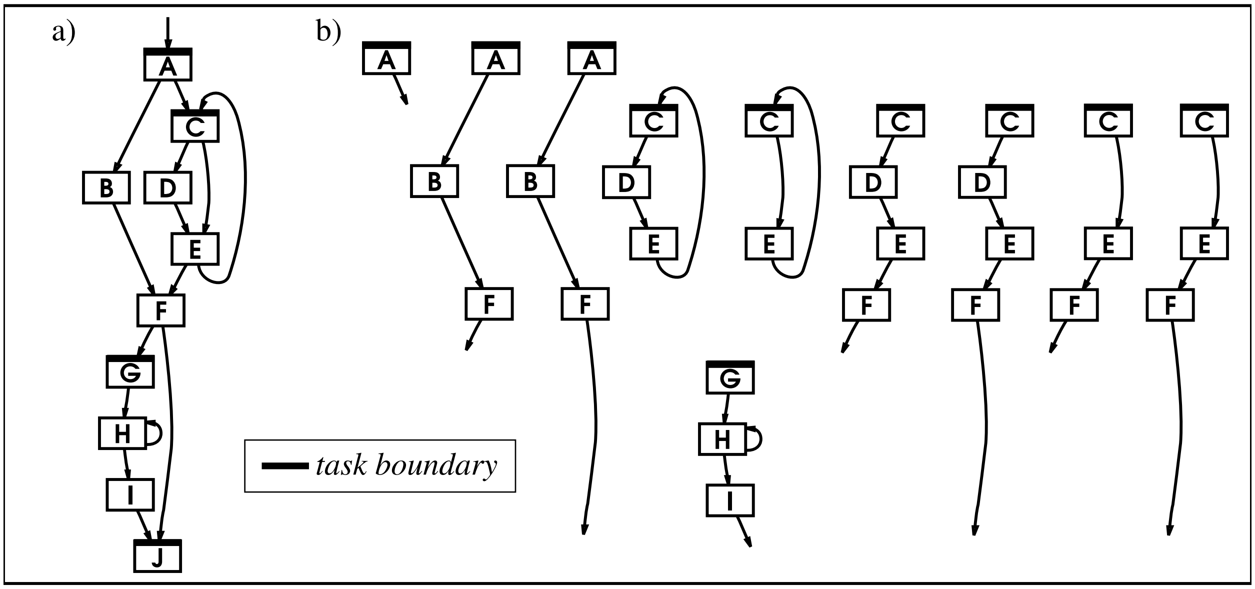


Figure 2.3: Control flow graph with task boundaries (a), and resulting potential tasks (b).

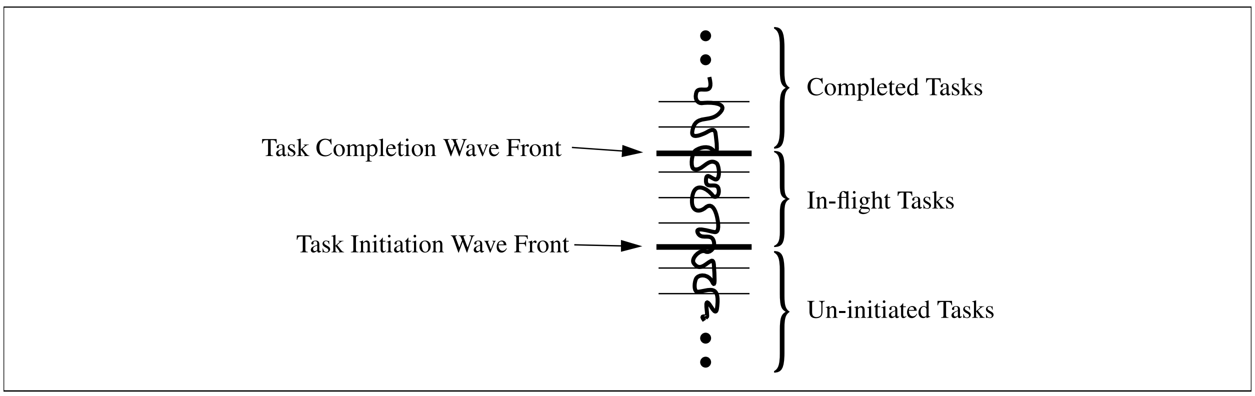


Figure 2.4: The program's execution can be divided into three regions. *(1) a continuous span of completed tasks starting at the program's beginning, (2) a short span of in-flight tasks, and (3) a continuous span of un-initiated tasks.*

While intra-task dependences can be handled by any number of existing micro-architectural techniques, the set of live-in dependences is the responsibility of the parallelization paradigm. The set of writes (to registers or memory) that are not overwritten within the task form the set of live-out values.

2.1.3 Handling Inter-Task Dependences

Before an operation with an inter-task dependence can be executed, the dependence must be satisfied. In the case of a true data dependence, this means the operand value must be supplied. In the case of a control dependence, the starting PC for a task must be supplied to identify which operation to perform. In both cases, a value must be provided to the consuming operation.

Before discussing the means of providing these values, it is important to describe one attribute of dependences: length. The *length* of a dependence is the number of dynamic instructions executed between the producing operation and the consuming operation in an in-order execution of the program. Like tasks, dependence length is a purely dynamic phenomenon, a characteristic of a pair of dynamic instructions that communicate. The length of a dependence—specifically its length relative to the size of the task window—may affect the cost of satisfying the dependence. Therefore, I make the following definitions:

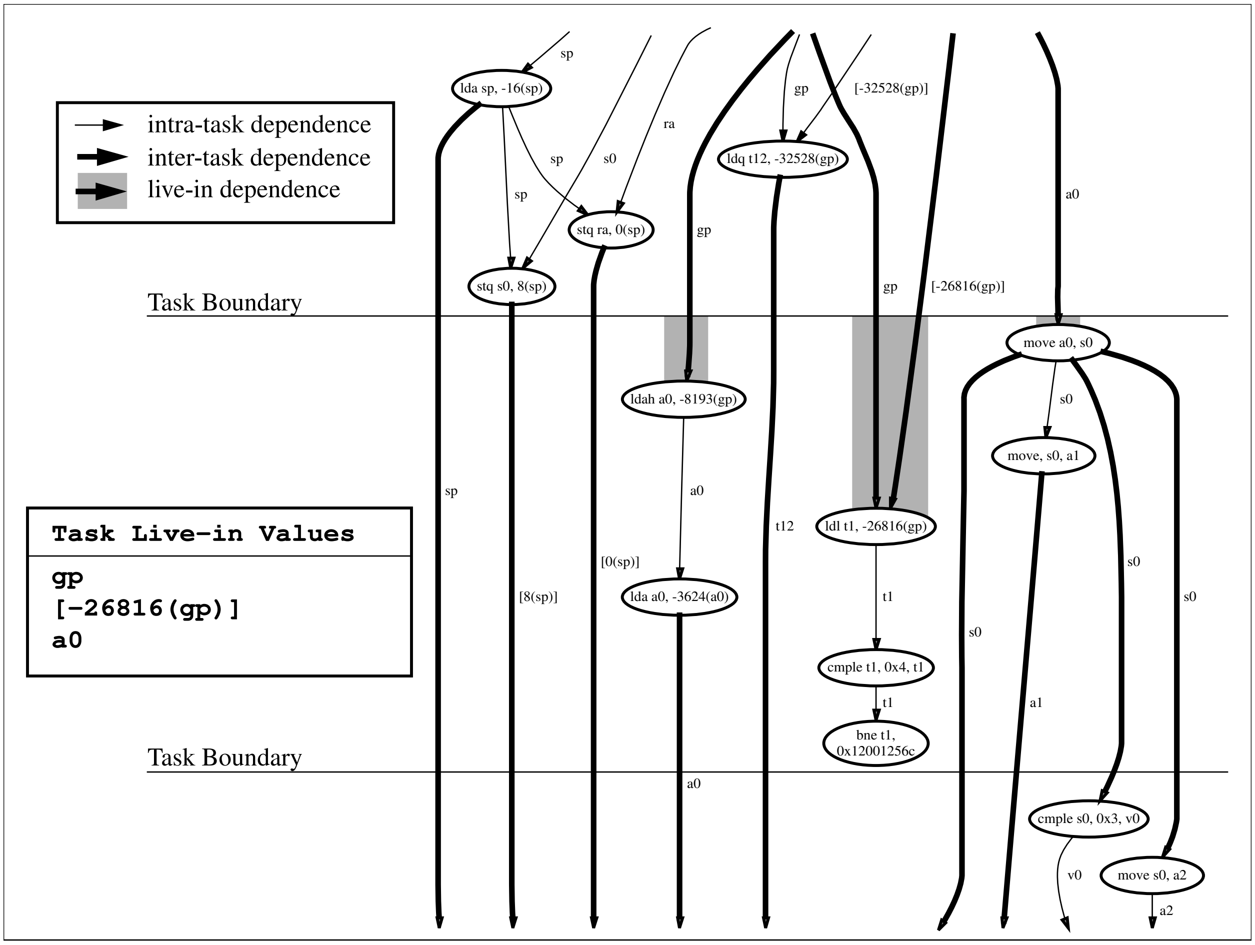


Figure 2.5: Example partition of EDG with annotated dependences. *Generally, the live-in set is a small fraction of the inter-task dependences crossing a task.*

long dependence: *a dependence with a length exceeding the task window size.* **(Definition 3)**

short dependence: *a dependence with a length less than or equal to the task window size.* **(Definition 4)**

Although this characteristic is defined with respect to a dynamic dependence, it is quite stable (across instances of the same static instruction) and predictable; the stability of very short dependences has previously been demonstrated [51]. Accurate predictions of dependence length can be exploited to select an appropriate mechanism for satisfying the dependence.

Below, I describe three ways to satisfy dependences: (1) reading the value from architected storage, (2) inter-task communication and (3) value prediction. The first mechanism is appropriate for long dependences; the other two are for short dependences. Figure 2.6 diagrams these three mechanisms.

As tasks commit instructions, they update user-visible state, also known as *architected state*. For long dependences we can be assured that the producing operation has been committed—by definition it belongs to a completed task—and that no in-flight operation is updating the storage location associated with the dependence. As a result, there is a single, unambiguous value for the name and it should be stored in the architected state. Thus, these dependences can be satisfied trivially by reading the value from the architected state.

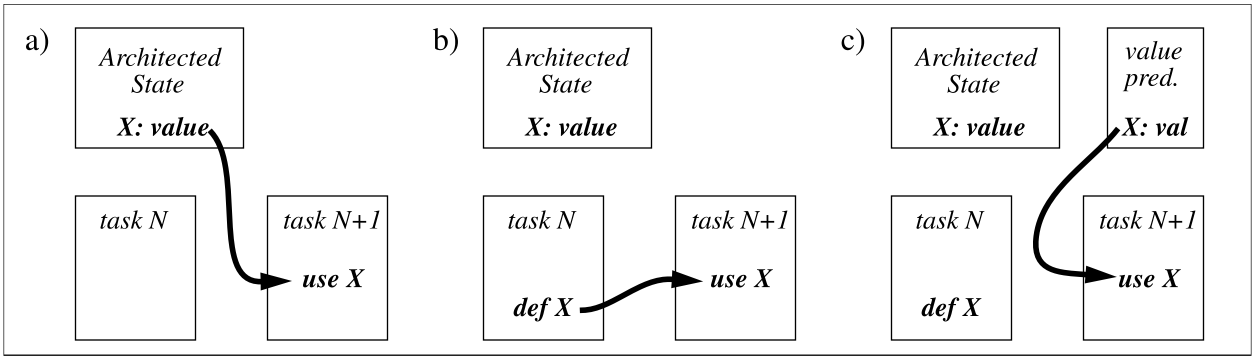


Figure 2.6: Three techniques for satisfying dependences. *(a) reading values from architected state, (b) communicating values from producing task, (c) predicting value with value predictor.*

For short dependences, the task that produces the value will be in flight simultaneously with the task that consumes the value. A natural way to handle such a dependence is to have the producing task communicate the value to the consuming tasks. Generally this approach requires a synchronization mechanism, as we cannot assume that an instruction that produces a value will have completed execution before the consuming operation is considered for execution. The synchronization mechanism must identify the producing instruction and stall the consumer until that instruction has completed.

Alternatively, we can satisfy the dependence by predicting the communicated value [22, 45]. In this case, the value will not be provided from the producing task, but from a separate entity, the value predictor.

If a value can be predicted effectively, using value prediction can result in better inter-processor communication patterns than inter-task communication. Figure 2.7 illustrates this point with a simple example involving a sequence of tasks with a single communicated value. In a traditional speculative parallelization paradigm this dependence is communicated from task to task, exposing the latency of inter-processor communication between each task. In MSSP, these live-in values are produced by a centralized value predictor. Because it is centralized, no inter-processor communication is required between executions of the increment operation that generate the communicated value. As a result, the required inter-processor communication for multiple tasks can be overlapped.

2.1.4 Verification of Speculatively Satisfied Dependences

Up to this point I have made no distinction between traditional and speculative parallelization techniques. Traditional parallelization techniques ensure correctness by proving that all operations that potentially have inter-task dependences that are shorter than the task window get the correct value, generally by using synchronization. Because correctness is ensured statically by construction, no run-time verification is required in these systems.

The requirements necessary to automatically construct parallel programs that could be proven correct have effectively prevented traditional parallelization of all but a few specific classes of programs. Speculative parallelization, where the process of getting the required values can be speculative, enables a broader class of programs to be parallelized. This speculation relaxes the correctness guarantees required from the compiler and substitutes run-time checks to ensure correctness. State updates need to be buffered speculatively so they can be discarded if run-time checks detect a dependency violation. There are two approaches that can be used for verification: (1) verifying dependences and (2) verifying values.

Verifying dependences requires tracking the source (*i.e.*, the producing operation or architected storage location) from which a value was retrieved and verifying that this source was in fact the correct source. This verification is generally composed of two parts: 1) selecting what appears to be the right value when the consuming operation executes, and 2) watching all writes to ensure that a more appropriate value does

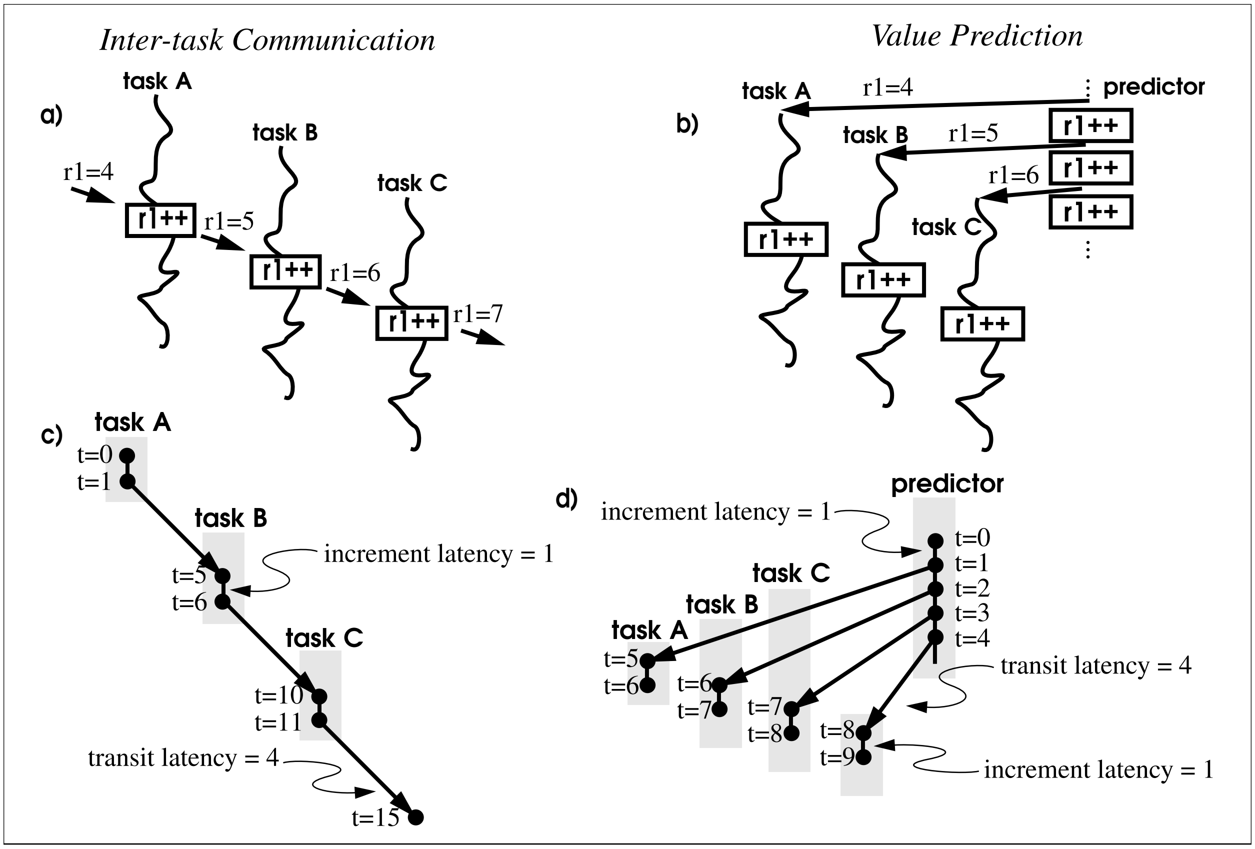


Figure 2.7: Alternatives for short dependences in speculative parallel architectures. *This figure shows an illustrative example, where the only inter-task dependence is due to a register-allocated counter that is incremented by each task. Live-ins can be supplied from previous in-flight tasks (a), or task live-in values can be value predicted (b). With inter-task communication, the inter-processor communication latencies are serialized (c), but with value prediction from a centralized value predictor, the latencies can be overlapped (d).*

not become available. More specifically, these requirements are: 1) ensuring this source provided the most up to date (*i.e.*, latest in program order) visible (*i.e.*, earlier in program order than the consuming operation) value bound to the desired name at the time the consumer was executed, and 2) ensuring that between the consumer's execution and retirement no operation modifies the value at the desired name and is both visible to the consumer and more up to date than the source. Typically, the first requirement is verified by construction (*i.e.*, only the most up-to-date visible value can be provided) and the second by snooping the stream of state updates. Clearly this approach does not work if value prediction is used, as the original dependence is ignored.

The alternative approach is to verify the value directly, ensuring that the value that was used is the same as the one held in the named storage location at retirement time. Such a verification can be accomplished in two ways: (1) effectively re-execute all instructions at retirement, and (2) verify task live-in values. Re-execution, used in DMT[2], is certainly sufficient for verification, but may be over-kill. If intra-task communication and computation can be assumed to be reliable, then it is sufficient to verify task live-in values. This approach involves recording the names and values of the data items a task has used before defining. When the previous task is complete, the current (architected) values associated with those names can be compared to the stored value.

By comparing values directly, any means for generating values (including prediction) can be supported. In addition, comparing values does not signal a misspeculation if the right value was used for the wrong reason. Research [43, 55, 78] has shown that many writes are "silent," they write a value into a storage location already containing that value. Dependence-based verification mechanisms would signal a misprediction if the storage location was read before the execution of a silent write that was earlier in program order, while value-based ones recognize that the correct value was read.

In effect, our verification and commitment is "reuse" of a block of instructions in the instruction reuse [71] sense. Said another way, the computation is being memoized [8]. The inputs and outputs of a task are packaged up, and if the input values match, then the output values can be committed to architected state.

2.2 Master/Slave Speculative Parallelization and Distilled Programs

With the background and vocabulary in place, I am prepared to discuss the MSSP paradigm. I describe first (in Section 2.2.1) a broader class of execution paradigms called checkpoint speculative parallelization paradigms. Second (in Section 2.2.2), I describe the concept of approximate code and a specific form of approximate code that I call distilled programs that the MSSP paradigm uses to predict checkpoint values. Then, in Section 2.2.3, I present the MSSP paradigm itself and describe the mechanisms that it requires. Finally, I present examples of how a program fragment is approximated and how the MSSP paradigm executes the program fragment, in Sections 2.2.5 and 2.2.6.

2.2.1 Checkpoint Speculative Parallelization Paradigms

I am coining the term *checkpoint speculative parallelization* for a class of execution paradigms that predominantly use value prediction to handle short inter-task dependences. When a new task is initiated these techniques logically construct a checkpoint of state as it is predicted to appear at the beginning of the task. The task can then satisfy all its live-in dependences by reading values from the checkpoint. If the checkpoint is constructed before the beginning of the task, no synchronization is required.

Though logically each task has an independent checkpoint, in practice checkpoints of neighboring tasks are largely similar, suggesting that such replication is unnecessary. In fact, since only a small number of tasks is in flight at a given time and each task generally writes only a small fraction of all architected state, most of a checkpoint is the same as architected state. Hence, only differences (*diffs*) from architected state need to be stored. Similarly, since the architected state holds the correct value for most storage locations, it is only the diffs—the values that could potentially have short dependences—that need to be predicted.

By storing the differences between each checkpoint and the previous one, the logical checkpoint for a task can be created by overlaying the checkpoint diffs of earlier in-flight tasks on the architected state in program order (see Figure 2.8). Thus, the logical checkpoint can be created on demand by using multi-version memory hardware structures like the ARB [10].

As discussed in Section 2.1.3, checkpoint-based paradigms, because they use value prediction, have the potential to be tolerant of inter-processor communication latency, but this latency tolerance can only be

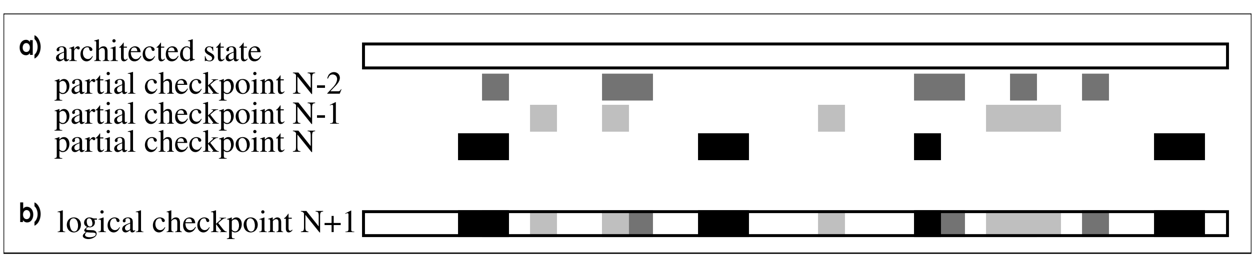


Figure 2.8: A live-in checkpoint is assembled from partial checkpoints. *(a) a partial checkpoint is predicted for values updated by each task, and (b) a checkpoint image for task N+1 is assembled by selecting the most recent copy of each value from the partial checkpoints and architected state.*

achieved if most of the live-in predictions are correct. Whenever a prediction is incorrect, the execution is serialized by an inter-processor communication to restart the value predictor with the correct architected state. In fact, an analytical model described in Section 4.2 predicts speedups increasing super-linearly with fraction of correct checkpoints. Thus, it is desirable to have near perfect prediction accuracy. Current hardware value predictors do not provide satisfactory prediction accuracy or coverage (or both) and often require large dedicated hardware structures [11, 30, 42, 45, 44, 67, 78, 82]. As a result, I instead use software, executed on a general purpose processor, to generate the necessary value predictions. Previous research has demonstrated that software can be used for accurate value prediction [18, 52, 64, 65, 66, 85].

2.2.2 Approximating Code and Distilled Programs

The MSSP paradigm predicts checkpoint values by executing a *distilled program*, generated by *approximating* the original static program. The definition of approximate code is as follows:

> ***Approximate code:*** *a distinct executable computation that has a high probability, but no guarantee, of computing a desired subset of the state computed by a given piece of reference code.* **(Definition 5)**

There are two important features of this definition. First, that the approximate code is not guaranteed to be functionally equivalent to the reference code. Second, the approximate code is distinct from the original code and therefore can be composed arbitrarily with respect to the reference. These two features, together, provide significant flexibility in construction of approximate code.

In return for a loss of precision, the approximate code is generally expected to have better execution characteristics. These better execution characteristics result from the large discrepancy found in many programs between what the program could do and what the program commonly does. Some examples of this discrepancy have been demonstrated:

Program paths. Even small programs like SpecInt95 benchmarks have more than 2^{32} potential acyclic paths, but less than a thousand paths account for more than 90% of the execution [6].

Size of points-to sets. Points-to sets measured during an execution were 5 times smaller than conservative sets constructed statically [50].

The discrepancy between the number of potential and common case behaviors results in a similar discrepancy between the set of transformations that can be proven safe and those that are safe in practice. Since approximate code need not be correct, it can be optimized beyond what traditional code generation allows by using these additional transformations. Selecting optimizations that both improve performance and largely compute the desired values correctly is facilitated by knowledge of what behaviors the program exhibits in practice. A simple way to collect this information is to monitor the program's execution to gather profile information.

A somewhat subtle part of the definition is that the approximate code only tries to compute a subset of the values computed by the reference code. In general, we are not concerned with arriving at the exact machine state achieved by the reference code but, rather, are focused on particular values of interest. Many of the values computed by the reference code are intermediate values, and forcing the approximate code to compute these values over-constrains it.

The MSSP paradigm uses a specific form of approximate code, called a distilled program. The distilled program is used to predict task live-in values. Thus, the "desired subset of state" computed by the distilled programs are those variables involved in short inter-task dependences. Long inter-task dependence values need not be computed by the distilled program because they can be obtained by other means (*e.g.*, read from the architected state). In addition, the distilled program specifies the task boundaries to that the MSSP implementation. These two characteristics result in the following definition:

Distilled Program: *a form of approximate code which conveys a division of the original program intotasksandcomputesthevaluesassociatedwiththeresultingshortinter-taskdependences.* **(Definition 6)**

After defining the MSSP paradigm in the next section, I present an example of how distilled programs are constructed in Section 2.2.5.

2.2.3 Master/Slave Speculative Parallelization

The Master/Slave Speculative Parallelization paradigm is a checkpoint speculative parallelization paradigm, but rather than using a hardware value predictor it uses a distilled program to generate value predictions. In fact, this feature is the essence of MSSP, resulting in Definition 7. In this section, I describe the MSSP paradigm at a high level. An illustrative example of its execution is presented in Section 2.2.6.

Master/Slave Speculative Parallelization: *A checkpoint speculative parallelization architecture that uses approximate code (in the form of a distilled program) to generate checkpoint value predictions.* **(Definition 7)**

MSSP (unlike other speculative parallelization (SP) paradigms) consists of two executions of the program: the *master*, or leading, execution and the *slave* execution. The two executions use different copies of the program as they serve different purposes. The master, which is responsible for performance, executes the highly optimized, but possibly incorrect distilled program. The slave execution, which is responsible for correctness, executes code much like the code generated by existing compilers. I'll refer to this code as the original program, because in a transparent implementation of the paradigm this program would be the original binary generated by an existing compiler.

Both executions use the same architected memory state, but, because the master execution is executing a potentially flawed program, it is not allowed to update architected state. Only the slave execution is permitted to update architected state. Values created by the master are buffered speculatively and used to facilitate the execution of the slave execution.

The two executions run in close succession—the master's lead is limited by the availability of speculative buffering—so performance will be determined by the slower of the two. To achieve execution throughput equivalent to the master, additional processors can be allocated to the slave execution. If the distilled program is capable of outperforming the original program by a factor of N, then perhaps N slave processors will need to be allocated to "match the impedances" of the two executions.

The parallelization of the slave execution is orchestrated by the master execution. To break the slave execution into tasks, task boundaries in the original program are selected. At the corresponding locations, `FORK` instructions are inserted into the distilled program. Upon encountering one of these `FORK` instructions, the master execution assigns the next task to an idle slave processor.

The master provides each slave processor with a starting program counter (PC) and predictions for the live-in values the slave requires. Because the control flow in the two programs roughly corresponds, the master predicts the slave task's starting PC by mapping its own program counter (PC) from the distilled program to a task start PC in the original program. Furthermore, speculative state (*e.g.*, register and memory values) generated by the execution of the distilled program serves as predicted live-in values for the tasks. This speculative state, along with the architected state, form a checkpoint of sorts of the program's state.

As these checkpoint values provided by the master are only predictions, the slave processor must execute speculatively until its inputs are verified. To permit this verification, the values read for live-in variables are recorded and buffered. Likewise, all side-effects of slave task execution (*e.g.*, register writes and stores) are buffered in a live-out value buffer. At the end of a task, if its live-in values match the architected state, the task's live-outs can be committed. Mechanisms are required to buffer all of the non-architectural data (checkpoint, live-in, and live-out values) and to correctly route it between processors. A full description of the required mechanisms, with an example implementation, can be found in Chapter 4.

Because all communication between the master and the slaves is in the form of predictions, which are verified before the slave updates architected state, there are absolutely no correctness requirements on the distilled program. This lack of correctness requirements means that the mechanism used to generate the distilled program need not be verified perfectly, as bugs can only result in performance degradation. This facilitates constructing the distilled program at run-time, when the most accurate profile information is likely to be available but verification may be costly in terms of computation or memory resources.

2.2.4 MSSP Exception Model

One important feature of MSSP, is that the paradigm's functionality is a super-set of the functionality of the execution paradigm used by the underlying processor cores. For example, if an MSSP execution is composed of out-of-order superscalar processors, all of the features of traditional out-of-order superscalar processors are present. This characteristic is exploited for handling exceptions. If an exception occurs, the execution can revert back to the committed architected state (by squashing all in-flight speculative execution) and then re-execute the excepting code using the underlying execution paradigm. Thus, MSSP has no exception model of its own, just a means to revert to a traditional execution to exploit that paradigm's execution model. More detail on how exceptions are handled is covered in Section 4.5.6. Next, I demonstrate distilled program construction.

2.2.5 Distilled Program Construction Example

An intuitive understanding of the potential of approximate code is most easily achieved through an example. In Figure 2.9, a hot loop nest from `gcc` is shown. The corresponding control-flow graph (CFG) is shown in Figure 2.10a. From the execution frequency annotations on the CFG edges it can be seen that the inner loop (blocks **B**, **C**, **D**, and **E**) is only executed on 0.1% of the interations of the outer loop (blocks **A** and **F**). In approximating this code fragment, the inner loop will be removed to optimize the execution of the outer loop.

```
   for (i = 0; i < regset_size; i++) {
      register REGSET_ELT_TYPE diff = live[i] & ~maxlive[i];
A
      if (diff) {
B       register int regno;
        maxlive[i] |= diff;
C       for (regno = 0; diff && regno < REGSET_ELT_BITS; regno++) {
          if (diff & ((REGSET_ELT_TYPE) 1 << regno)) {
            regs_sometimes_live[sometimes_max].offset = i;
            regs_sometimes_live[sometimes_max].bit = regno;
D           diff &= ~ ((REGSET_ELT_TYPE) 1 << regno);
            sometimes_max++;
          }
      E }
      }
 F }
```

Figure 2.9: Example code fragment from SPEC 2000 benchmark **`gcc`*****.*** *Loop nest extracted from function* `propagate_block`*, lines 1660-1678. Statements annotated with the control flow block to which they belong; blocks **E** and **F** perform the loop counter increment and test for the inner and outer loops, respectively.*

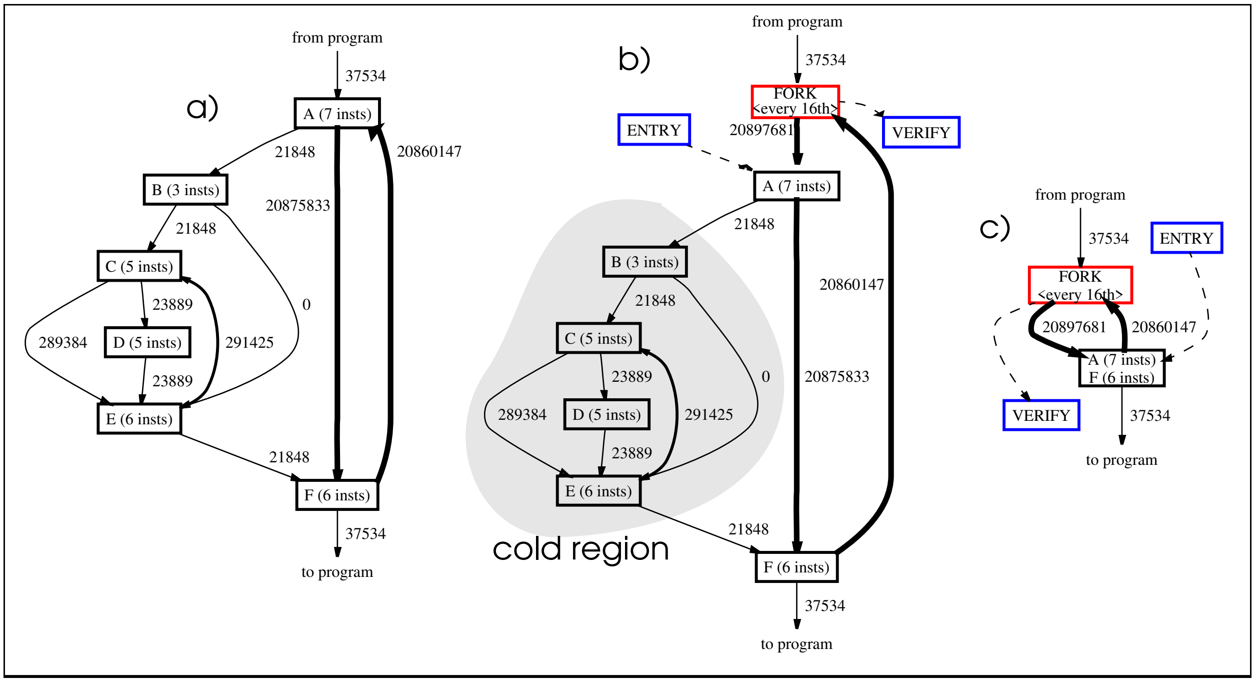


Figure 2.10: Example task selection and application of root optimizations. *The control flow graph for this code example is shown for: (a) the original code, (b) the code after it has been transformed to include a task boundary, and (c) after the cold code region has been removed.*

Constructing the distilled program consists of a series of steps, shown in Figure 2.11. For this example, the only profile information needed are the previously mentioned edge frequencies. The internal representation is constructed by reading the instructions from the program's memory image. Task selection is described in detail in Section 3.2. For this example it is sufficient to say that inner loop iterations are too small (11 dynamic instructions on average), so outer loop iterations are used for tasks, with a task boundary inserted at the loop header. In fact, even outer loop iterations are too small (13 dynamic instructions on average), so the task boundary is annotated to pack 16 iterations into a task.

When tasks are selected the control flow structure of the internal representation is transformed to include additional blocks. For each task boundary, three blocks are added: a fork block, a verify block, and an entry block. The fork block will contain a `FORK` instruction that signals to the master processor to allocate a slave processor to begin executing at the corresponding verify block. The entry block is used—after a task misspeculation has occurred—to jump start the master execution using the original program's state.

Next the program distiller performs a liveness analysis on the original code to determine the set of values that are live across the fork block. This set of values will need to be preserved even if all of their uses in

```
1. collect profile information
2. build internal representation (IR)
3. select task boundaries
4. perform liveness analysis
5. apply (speculative) root optimizations
6. apply (non-speculative) supporting optimizations
7. layout and generate code
```

Figure 2.11: Logical steps in constructing a distilled program.

```
                     live before A: t0 (&live[i]), t3 (&maxlive[i]), t12 (i)

                     A:
DEAD                   ldl      t6, 0(t3)          #1 load maxlive[i]
DEAD                   ldl      t2, 0(t0)          #2 load live[i]
DEAD                   zapnot   t12, 0x3, t11      #3 truncate i to 16b
DEAD                   bis      zero, zero, ra     #4 regno = 0
DEAD                   bis      zero, zero, t9     #5 regno = 0
DEAD                   bic      t2, t6, t2         #6 diff = live[i] & ~maxlive[i]
SINGLE TARGET          beq      t2, F              #7 if (diff)

                     F:
                       addl     t12, 0x1, t12      #1 i ++
                       lda      t0, 4(t0)          #2 &live[i]
                       lda      t3, 4(t3)          #3 &maxlive[i]
LOOP INVARIANT         ldl      s3, -10104(gp)     #4 load regset_size
                       cmplt    t12, s3, s3        #5 i < regset_size
                       bne      s3, A              #6 loop back-edge
```

Figure 2.12: Distilled program fragment after application of root optimizations. *Branch A7 is unnecessary as block A has a single successor block. A3-A5 are instructions hoisted out of the inner loop, which are now dead. Removal of A7 results in A1, A2, and A6 becoming dead. Instruction F4 is loop invariant.*

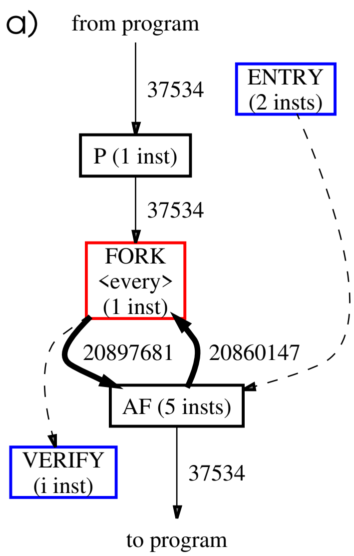
the distilled program are removed, as they may contribute to the live-in set of tasks in the original program spawned at this task boundary. The relevant live-in values for this example are shown at the top of Figure 2.12.

After liveness analysis, the next step is to apply root optimizations. These are optimizations that would be unsafe to apply to traditionally generated code, but are used in approximate code to remove uncommon-case behaviors that prevent optimization of common-case paths. In this example, the inner loop (labelled the cold region in Figure 2.10b) is removed from the distilled program. Eliminating this path may result in task misspeculations when this path is executed by the original program but greatly simplifies the CFG of the remaining code (Figure 2.10c). Once the inner loop is removed, outer-loop blocks **A** and **F** can be unified with no further loss of precision.

Application of the root optimizations creates new opportunities for traditional safe optimization (what I call supporting optimizations in this dissertation). The code that remains after the inner loop has been removed is shown in Figure 2.12. Traditional analysis can detect that both paths from branch **A7** lead to block **F** and that instructions **A1**-**A6** are dead; all of these instructions are removed from the distilled program. In addition, instruction **F4** is detected to be loop invariant, so it can be hoisted out of the loop.

Moving the instruction **F4** to the loop pre-header—block **P** is inserted in the CFG as shown in Figure 2.13—is not without complication. This code motion crosses the incoming edge from the entry block to block **AF**. To compensate for this code motion, a copy of **F4** is added to the end of the entry block to ensure that the value will be computed when transitioning from the original program to the distilled program. There are situations—discussed in Section 3.3.2—that require adding compensation code to the verify block as well.

Block **AF** is then unrolled by a factor of 16 (a degree equal to number of loop iterations per task). Since the induction variables are dead on exit from the loop, only the final branch instruction in the unrolled loop is required, as it subsumes the others. Each of the three induction variable calculations (now each consisting of 16 copies of an add immediate instruction) can be algebraically simplified to a single a)



b)

```
ENTRY:
  ldl      s3, -10104(gp)     # was F4
  br       A

P: (inserted loop pre-header block)
  ldl      s3, -10104(gp)     # was F4

FORK:
  fork     VERIFY

AF:
  addl     t12, 0x16, t12     # was F1
  lda      t0, 64(t0)         # was F2
  lda      t3, 64(t3)         # was F3
  cmplt    t12, s3, s3        # was F5
  bne      s3, AF             # was F6

VERIFY:
  verify   A (in orig program)
```

Figure 2.13: Distilled version of example code fragment: *a) the CFG is further transformed to support a loop header block (H), (b) the instruction contents of each block is shown including the compensation code in the entry block resulting from the motion of instruction* ***F4*** *to the loop header. Note the constants of instructions* ***F1****,* ***F2****, and* ***F3*** *have been scaled by 16.*

instruction with the original constant scaled by 16. The fork instruction is further annotated to spawn on every iteration, although each task should still contain 16 iterations of the original program.

The resulting distilled program (shown in Figure 2.13) executes only 6 dynamic instructions per task, where the original program executes over 200 dynamic instructions per task, on average. In the next section, I demonstrate an MSSP execution using this code fragment.

2.2.6 MSSP Execution Example

While the MSSP paradigm consists of a small number of straight-forward activities, an execution can be rather complex because many of these activities are occurring concurrently. In Figure 2.14, I list the activities and show the causal relationships between them in a flow chart. The first action performed in an MSSP execution is the restart, which initiates both the master and a non-speculative slave task. The slave

```
1. restart
2. task execution
   2a. non-speculative execution
   2b. master execution
   2c. speculative execution
3. task verification
4. task commit
5. task misspeculation recovery
```

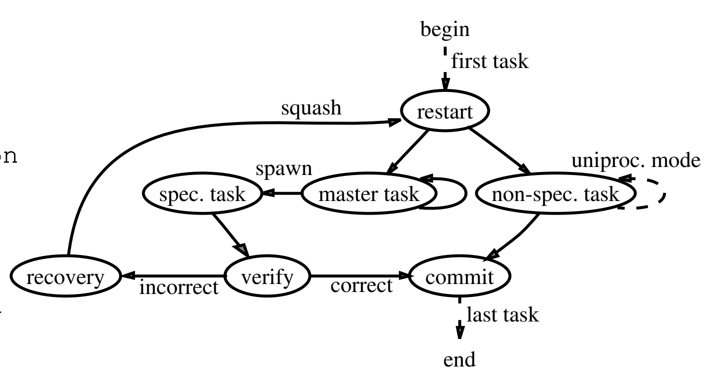


Figure 2.14: Flow chart of MSSP execution.

task can execute non-speculatively (*i.e.*, like a normal uniprocessor) because all of its live-in values are available from architected state and therefore need not be verified. This non-speculative task commits its writes immediately and terminates when it reaches the end of the task.

The master processor, on the other hand, executes task after task in the distilled program (as indicated by the arc from the master to itself in Figure 2.14). In addition, at every task boundary it creates a speculative slave task. There can be many speculative slave tasks concurrently executing, but there will be at most a single non-speculative slave task at any time. If no non-speculative task is running, then the oldest speculative slave task can be verified. If incorrect, a task misspeculation has occurred; recovery is performed by squashing all in-flight tasks and restarting from the most recent architected state. If correct, the task will be committed. The execution ends when the last task is committed.

MSSP can revert to a traditional uniprocessor execution by allowing a non-speculative task to "spawn" the next non-speculative task when it has completed (as shown by the dashed arc from non-spec. task to itself). The slave need not wait until the first task has committed before beginning the next; by simply disabling the task end annotations the slave will execute the program sequentially until they are turned back on.

To more concretely demonstrate the execution, Figure 2.15 shows an illustrative execution of the code fragment distilled in the previous sub-section. The execution begins partway through the loop's execution with the variable `i` equal to 48 and `sometimes_max` equal to 11. For simplicity, I will ignore the other variables. The distilled program is constructed such that the variable `i` is always correctly computed, but the variable `sometimes_max` may be incorrect when the original code executes the inner loop, which has be removed from the distilled program.

The execution commences with the machine quiesced: all processors are idle and all buffers for non-architected data are empty. The restart process initiates two executions of task #0: a non-speculative slave one and a master one. While the non-speculative slave is provided with the architected state, the PC for the master is first mapped (using a lookup table) to find the PC of the corresponding entry into the distilled program. The master executes the entry code before entering the distilled program proper. Both tasks execute simultaneously. The slave task executes like a normal uniprocessor, immediately committing its state. In this example, I assume that the 16 iterations executed by the non-speculative slave all avoid the inner loop, so its only writes are 16 updates to the variable i, of which only the first and last are shown.

Writes performed by the master, on the other hand, are buffered as checkpoint data and associated with the master's current task. In task #0 the master performs a single write (i=64) before encountering a fork instruction. At a fork instruction, the master increments its task number and notifies the system that the speculative slave task #1 is ready for execution. This task is allocated to processor P2, which was previously idle. Rather than the current architected value, P2 is sent the checkpoint value (64) for variable `i`. Since no checkpoint value is available for `sometimes_max`, the architected value (11) is sent. The slave processor begins execution in the verify block of the distilled program—using *verifyPC*, provided by the master—which ends with a jump into the original program. While executing the original program, the speculative slave records its live-in values; I will assume that inner loop iterations are executed in both tasks #1 and #2, so the predictions for both variables `i` and `sometimes_max` are recorded and sent to the live-in buffer.

During the slave execution of task #1, the master forks task #2. Because no processor is available, slave task #2 cannot be immediately assigned to a processor. When the non-speculative task on P0 completes—after its final write to the variable `i`—task #2 is assigned to P0. Speculative slave task #2 is provided with the most recent checkpoint value (80) for `i` and the architected value (11) for `sometimes_max`, because no checkpoint value exists. Like slave task #1, both of these values are read and recorded as live-in values.

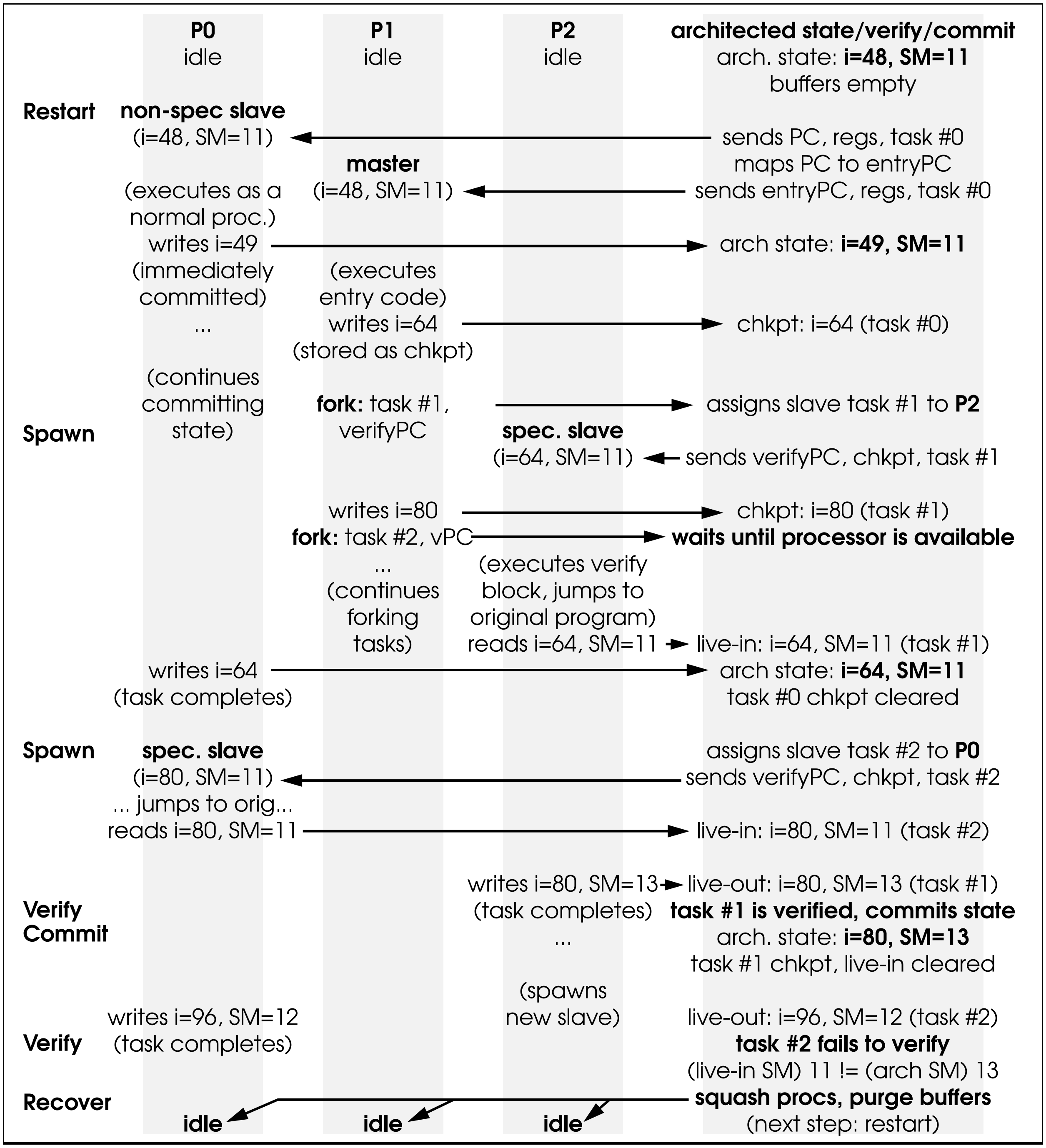


Figure 2.15: Detailed example MSSP execution. *The code example from Figure 2.9 is executed on 3 processors. The values of only two variables are displayed* `i` *(the induction variable of the outer loop, which is correctly updated by the master) and* `sometimes_max` *(which is abbreviated SM and is incorrectly predicted by the master when the inner loop is executed by the original program). In this example both speculative slave tasks execute the inner loop, resulting in a task misspeculation when the second task undergoes verification. In the real execution, this situation is uncommon as only one iteration out of every thousand execute the inner loop.*

At the end of slave task #1, the values for `i` and `sometimes_max` are 80 and 13, respectively. These values are recorded in the live-out buffer. Because task #0 has completed, task #1 is the oldest task in the system and can be verified against architected state. In this case, the live-in values match the architected state, and the live-out values for task #1 can be committed. At this point the checkpoint and live-in values for task #1 can be discarded.

When task #2 completes and requests verification, a task misspeculation is discovered. When the recorded live-in value for `sometimes_max` (11) is compared to the architected value (13), the values do not match. The live-in predictions the master processor provided to slave task #2 implicitly assumed that task #1 would not modify the variable `sometimes_max`. Because its live-in values were incorrect, the live-out values for task #2 are assumed to be incorrect and discarded. To recover from the task misspeculation, all in-flight tasks are squashed and all non-architected state buffers are purged, returning the system back to the quiesced state.

2.3 Related Work

My development of the MSSP paradigm was heavily influenced by four main bodies of work: speculative parallelization (SP), leader/follower architectures, speculative compiler optimizations, and dynamic optimization systems. In the next four sub-sections, I summarize the work in each of these areas, describing the relationship to the MSSP paradigm.

2.3.1 Speculative Parallelization (SP)

The insight [39] that speculation could overcome the difficulties of traditional automatic parallelization has unleashed a large and active body of research. The Multiscalar paradigm [25, 72] has been at the vanguard of this movement and a major influence on me, due to my involvement in the Kestrel project, a VLSI feasibility study of a Multiscalar architecture.

In the Multiscalar paradigm, a Multiscalar compiler generates a Multiscalar binary to be executed on a collection of tightly coupled processing elements. Tasks, annotated in the binary, are spawned in order by a next task predictor. Two mechanisms are used for inter-task communication: 1) the register ring forwards register values as directed by annotations in the binary, and 2) the address resolution buffer (ARB) matches load and store requests using the logical ordering of the tasks.

The ideas in the Multiscalar work have been evolved in many directions. One common attribute of follow-on research was toward compatibility with explicit parallelism architectures like chip-multiprocessors (CMP) [16, 33, 73], simultaneous multithreading (SMT) [2, 47], and distributed shared memory (DSM) [15, 74] architectures. Such architectures enable resources to be used flexibly to support whatever form of parallelism is available, an attribute shared by MSSP.

All of the proposed speculative parallelism architectures have to handle a small set of requirements: predict the sequence of tasks, handle inter-task communication, buffer speculative state, and detect misspeculation. In the paragraphs that follow, I describe the ways in which MSSP differs from other proposals.

The MSSP paradigm predicts task PCs in a similar manner as it does any live-in value, by using the approximate program. The approximate code explicitly specifies the task start PC by a combination of a `FORK` instruction and a `VERIFY` instruction (described in detail in Section 3.3). In some respects, this technique is similar to SP paradigms that include explicit fork instructions that specify the next task PC (*e.g.*, [20, 33, 76]). The difference lies in the fact that in MSSP, it is not the tasks that spawn tasks but the centralized master. In this respect, MSSP is similar to the centralized “next task predictor” [34, 72] found in the Multiscalar paradigm and others [2, 16]; the clear distinction being that in MSSP the prediction is made by software rather than a hardware mechanism.

MSSP also differs from most SP paradigms because it is a checkpoint SP architecture and therefore has no mechanism for inter-task value communication. Instead, it relies on value prediction for short inter-task dependences. Prior research includes some partial checkpoint SP architectures that provide a checkpoint of register state. DMT [2] uses the register state of the forking task (at the time of the fork) as a prediction of the task's starting register state, relying on cheap incremental recovery to handle the inevitable misspeculations. Speculative Multithreaded processors [47] go one step further by using stride value prediction for predictable sequences; synchronization is used for unpredictable register values. The stride value prediction is performed by starting with the previous task's register state and injecting additional arithmetic operations into the window to add the stride. Speculative Multithreaded processors (and others paradigms that use value prediction [16, 56]) include a hardware value predictor; the MSSP paradigm differs in that it uses software executed on a general purpose processor to compute checkpoint value predictions.

The MSSP paradigm makes little contribution in how speculative state is buffered. Like most SP paradigms it extends the cache memory system to support speculative state storage. This enhancement generally requires the memory to support multiple versions of a given storage element simultaneously and provide a mechanism to select/construct the storage as it is seen by each task. Centralized [26], snooping [32, 33, 73] and directory [15, 74] implementations have been proposed. The mechanism required in MSSP is somewhat simpler than a general SP paradigm since the checkpoints are constructed in-order and before the (slave) task begins.

MSSP, like any SP paradigm that uses value prediction, uses value-based verification. The mechanism described in Section 2.2.3 tracks the live-in values consumed by a task and explicitly checks these values against architected state. This approach has similarities to the ones used in TLDS [75], which buffers the name/value pair of values that are explicitly predicted, and SM processors [47], which explicitly predicts task outputs and verifies them against architected state after the task commits. It differs somewhat from the system proposed for DMT, which re-executes the instructions at retire time to detect misspeculations.

There are many similarities between MSSP and the more limited form of speculative parallelization found in the trace processor microarchitecture [61, 63, 80]. Although the trace processor has a centralized fetch mechanism, its instruction window and physical register file are partitioned and distributed. Traces of instructions are distributed to each window segment in control-flow order. The trace processor microarchitecture proposed the use of live-in value prediction to further de-couple the execution of the traces. Aside from the differences between trace processors and other speculative parallelization techniques, the two central difference between MSSP and Trace Processors are that (1) all live-in values are predicted and (2) approximate code is used rather than a hardware value predictor. Architectures similar to the trace processor that support different partitioning policies [36] and dynamic vectorization [79] have been explored.

2.3.2 Leader/Follower Architectures

The master/slave architecture of the MSSP paradigm corresponds to the leader/follower architectures proposed for sequential processors. The first of which I am aware was the decoupled access/execute architecture [69], which, to tolerate memory latency, broke the program into a stream that loaded and stored data values (leader) and a stream that performed non-address computation (follower).

More recently leader/follower architectures have gathered a lot of attention, but with a micro-architectural rather than architectural implementation. Pre-execution proposals [18, 52, 64, 65, 66, 85] execute a speculative subset of the program (leader) to prefetch and generate predictions for the complete execution of the original program (follower). It was my work in this context that lead to the concept of approximate code [84].

Leader/follower architectures have also been proposed for fault tolerance, where both processes execute the original program, detecting inconsistencies in the executions. AR-SMT [60] performs the two executions as threads on an SMT. Diva [4] performs the second execution on a simpler processor that can be verified to detect design faults in the core. In some respects, the use of approximate code in MSSP is the software analog of Diva; the distilled program and the parallelized correct code work symbiotically to create a fast, correct execution.

Of the leader/follower architectures, the closest to MSSP is SlipStream [77]. In the SlipStream paradigm, the leader is also a speculatively reduced version of the code executed by the follower. The MSSP paradigm differs from SlipStream in three major ways: (1) the follower execution is parallelized in MSSP, (2) the code executed by the leader is a separate static image rather than a strict subset of the original program—a separate image provides additional optimization opportunities, but it requires explicit maps to correlate the two executions—and (3) SlipStream dynamically selects the program subset based on the predicted path, where MSSP uses a static distilled program.

2.3.3 Speculative Program Optimizations/Transformations

Our leader, the distilled program, derives its advantage over the whole execution through applying profile-driven speculative transformations. Previous work on speculative transformations involves generating code for the common case that includes checks to detect uncommon case executions. If a check fails, control flow is re-directed to fix-up code that can handle the general case. The use of speculative transformations in MSSP differs from this previous work, because verification is not performed by the transformed code but by running the original program in parallel. Said another way, the traditional approach to speculative transformations is to put the checks inline, while in MSSP they can be placed in a separate thread.

This distinction—shown visually in Figure 2.16—is an important one, because the amount of check code can be substantial. The check code typically can be scheduled in parallel with the rest of the code, but by being inserted inline, it will consume fetch, decode, execute, commit, and cache bandwidth and occupy space in the processor's instruction cache. These requirements often necessitate wider processors, whose width makes achieving high clock frequencies difficult [57]. In contrast, MSSP relegates the check code to a separate thread. In this way, the program that determines performance is not encumbered with the check code and can efficiently execute on a narrow processor with a high clock frequency.

The transformations can be classified into three main types: control-flow-speculative, data-dependence-speculative, and value-speculative. Control-flow based optimizations typically involve hoisting operations across basic-blocks, introducing them onto paths on which they were not originally present [13, 17, 19, 24, 46, 70]. Data-dependence based optimizations are employed to hoist load operations above

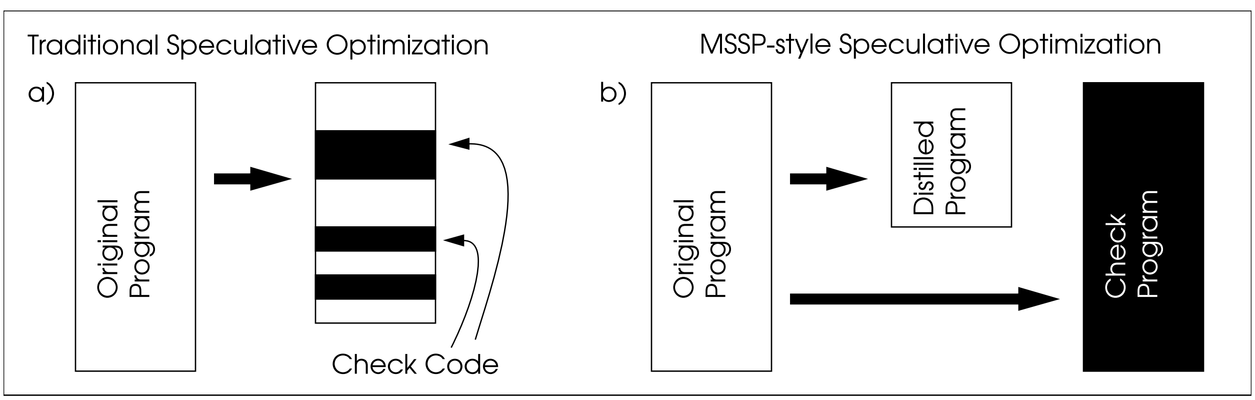


Figure 2.16: Traditional vs. MSSP-style speculative optimization. *Previous approaches to speculative optimization (a) optimize the expected path of the program, inserting checks inline. Although these checks can usually be scheduled off the critical path, they still will require fetch and execution bandwidth, perhaps necessitating a wide machine. In contrast, in MSSP (b) no checks are included in the distilled program: all checking is performed by separate threads by a "check program" (in this case the original program). If the full execution progresses at the rate of the distilled program's execution, then the benefit of the speculative optimizations can be achieved without needing a wide machine.*

store operations that could potentially alias [28, 35, 50]. Value-based optimizations exploit dynamically invariant values, by specializing the code for the dominant value [12, 27].

These speculative optimizations accelerate hot paths at the expense of the cold paths. Typically, the optimizations transform the hot path such that it is not safe to directly transfer control to the cold path. First some compensation code must be executed that reverse the optimization applied to the hot path. MSSP incorporates a form of compensation code that I call transition code to support transitioning from the distilled program back to the original program, but a key feature of the transition code is that—like the distilled program—it need not be 100% correct. Like MSSP, the rePLay architecture [23, 58] relies on the original code when the speculatively optimized code is inappropriate (rePLay avoids the need for compensation code altogether), but it still relies on checks inlined in the optimized code to verify its appropriateness.

2.3.4 Dynamic Optimization Systems

Because the most representative profile information will be available at run-time, I envision that an MSSP implementation will construct the distilled program at run-time, like a dynamic optimization system. There has been a lot of recent research in dynamic optimizers and dynamic translators [5, 14, 17, 21, 38, 83]. The structure of the distilled program shares some characteristics with these systems. Specifically, both sets of systems create a second copy of parts the program (which need not be in the same instruction set architecture as the original program) that resides at a different location in memory. This necessitates functionality to map from one program to the other. In both systems, code is translated at a larger granularity than an instruction—traces in dynamic optimizers, tasks in MSSP—and support is required to transition back to the original executable at boundaries of these translation units.

The main difference between MSSP and these systems is that they intend to use the translated copy of code *instead of* the original code, while MSSP uses it *in addition to* the original code. MSSP always executes the original code so there are no correctness constraints on the optimized copy. In contrast, dynamic optimization systems are highly constrained.

One of the most severe constraints is that they must produce the precise architected state of the original executable at any instruction boundary in the original executable where a trap could occur. This prevents removal of some instructions and limits the degree instructions can be reordered. In MSSP, traps are handled much in the same way that they are in existing out-of-order processors; if a slave encounters a trap, it flushes all post trap instructions and retires all pre-trap instructions to recover the precise architected state. Before handling the trap, the slave may need to wait to be verified and become non-speculative, if some of the trap handling actions cannot be performed speculatively. All traps encountered by the master can be ignored.

Dynamic optimizers are also constrained by the need to avoid memory ordering violations. In a multiprocessor context (or even in a uniprocessor system that supports coherent I/O), unless the dynamic optimizer has somehow been ensured that a variable is thread private, it must be careful when removing memory operations. Even if reads or writes appear redundant in a uniprocessor context, they may be necessary due to external reads or writes. This problem could be solved by committing groups of instructions atomically (as is done in the Crusoe processor [38] for precise exceptions), allowing redundant loads and stores within a group to be eliminated. In MSSP, the master's writes are not visible to external parties, so such precautions are unnecessary, but similar techniques are required for committing live-out values, because no intra-task ordering information is maintained in the live-out buffer.

2.4 Chapter Summary

In this chapter, I described the parallelization of sequential program as the process of breaking them into continuous sequences of instructions (tasks) and supplying each task with the live-in values it requires.

I demonstrated this concept with the Execution Dependence Graphs to emphasize that the execution paradigm should focus on the set of live-ins actually consumed, not those that could be consumed.

I also introduced the Master/Slave Speculative Parallelization (MSSP) paradigm. MSSP, as a checkpoint speculative parallelization paradigm, supplies these live-in values by predicting them before the task is executed. The MSSP paradigm differs from previous research, primarily in that it uses a specially constructed piece of software, the distilled program, to accurately predict these values.

Chapter 3

Approximate Code and Constructing Distilled Programs

In this section, I discuss the constraints on and opportunities for distilled programs, the form of approximate code used by the MSSP paradigm. First (in Section 3.1), I discuss the requirements of distilled programs and some design decisions to maximize distilled program performance. The two major issues—dividing programs into tasks and the structure of distilled programs—are described in the next two sections (Section 3.2 and Section 3.3, respectively). In the final two sections, I present the framework (Section 3.4) and implementation (Section 3.5) of the prototype distiller I constructed, with emphasis on how the structure of the distilled program affects the traditional compiler algorithms used.

3.1 "Requirements" of the Distilled Program

Since the distilled program cannot affect correctness there are no requirements on it *per se*, but it should perform two functions to be useful: 1) it should break the program's execution into tasks and 2) at task boundaries it should provide the program state expected by the original program. What makes these "requirements" challenging is that, ideally, they are accomplished in such a way to maximize the performance of the distilled program and, as a result, the whole execution. In this section, I describe my approach to maximizing performance as it relates to these two requirements of the distilled program.

Task Selection. Although, in theory, task boundaries could be selected arbitrarily (*e.g.*, after a fixed number of instructions executed by the master), performance is improved by making an informed selection of task boundaries and tailoring the distilled program for them. As was shown in Section 2.1.2, the selection of task boundaries determines the set of live-in values to a task. Two implications result:

- By selecting the task boundaries, the set of live-in values can be selected. Boundaries with many or hard-to-compute live-in values can be avoided.
- Since we can determine which values are live at the task boundary, we are free to remove any dead value without a risk of introducing a misspeculation.

Secondarily, it is desirable to have tasks about the same size. This desire is secondary because task size imbalance can be tolerated, at the expense of lower resource efficiency.

In addition, a mechanism is required to indicate the end of a task to the slave processor. When executing properly, the slave processor should stop executing just before the instruction that the master specified as the first instruction of the following task. Coordinating task ends with task beginnings allows the concatenated task executions to be equivalent to a sequential execution of the program.

Providing the Expected Program State. Correctly stitching individual task executions together requires that the state used by a task must match the state at the end of the previous task. This requirement can be accomplished trivially—without correctness guarantees—by taking a strict subset of the original program as is done in the SlipStream paradigm [77]. To improve performance of the distilled program beyond that of SlipStream's A-stream, I relax two constraints:

- The distilled program need not be a strict subset of the original program. Code can be transformed and re-scheduled, and new code can be introduced. As a result, there is no longer a one-to-one mapping from the distilled program to the original program, so an explicit mapping function for program counters will be required.
- The distilled program can store data in different locations than the original program. Removing this constraint allows limited register resources to be used more effectively by changing which architectural register is bound to a particular value and binding memory values to registers. Before the original code can be executed by the slave, the distilled program's state will need to be mapped to the locations in which the original program (*i.e.*, the task executed by the slave processor) expects it. This functionality will be performed by *transition code*.

In the two sections that follow, I explore these two issues in greater detail.

3.2 Selecting Task Boundaries

The location of the task boundaries can affect the performance of the MSSP paradigm. In selecting task boundaries, there are three main issues to consider: 1) how big should tasks be, 2) what program boundaries result in good tasks, and 3) what mechanism should be used to specify task end conditions to slave processors. These three issues have tightly-coupled interactions, but, in the sub-sections that follow, I try to consider each in turn. At the end of this section (in Section 3.2.4), I describe my task selector implementation.

Before getting into the details, it is useful to briefly contrast task selection for Multiscalar from that for MSSP, as there are important differences. The Multiscalar task selector tries to encapsulate reconvergent control flow within a task to facilitate next task prediction by minimizing the number of task successors; for MSSP control independence is not a concern. Both task selectors try to minimize the number of inter-task data dependences, but for different reasons. Multiscalar is trying to minimize the likelihood that tasks stall or are squashed due to value communication, while MSSP is trying to facilitate distillation. For both paradigms, tasks must be a minimum size to tolerate the startup overhead, but for the tightly-coupled Multiscalar paradigm tasks of 10-20 instructions are sufficient, while for MSSP tasks of hundreds of instructions are more desirable. Both paradigms desire tasks to be roughly equal in size, but MSSP may be more tolerant in this regard because, without the constraint of the register ring, it can dispatch tasks to any idle processor.

3.2.1 Task Size

The size of tasks can have a first-order impact on performance, as many of the characteristics of an MSSP execution are indirectly affected by task size. The two main factors that improve with larger tasks are distillation effectiveness and bandwidth utilization. The two main factors that are worse with larger tasks are misspeculation detection latency and the storage requirements for buffering speculative state.

Optimization Effectiveness. Larger tasks means fewer task boundaries, which generally results in less constrained approximate code, improving the effectiveness with which it can be optimized. Sensitivity analysis in Section 5.2.7 shows that my distiller prototype exhibits a 9% reduction in the number of dynamic instructions executed in the distilled program when average task size increases from 87 to 342 instructions (a factor of 4).

Bandwidth Utilization. Interconnect bandwidth is used to move around task, live-in and live-out values between processors. The amount of bandwidth required per task scales sub-linearly with task size, so the bandwidth required per original program instruction decreases with larger tasks. This reduction is found in both the estimates in Section 4.4.3 and the data shown in Section 5.2.7

Misspeculation Detection Latency. Less desirably, larger tasks will increase the time between task spawning and task retirement. Larger tasks generally results in a longer latency to detect a misspeculation as is shown in Section 5.2.7.

Storage Requirements. Hardware structures are required to buffer checkpoint, live-in, and live-out data for in-flight tasks. Estimates in Section 4.4.3 indicate that larger tasks will have larger checkpoints, more live-in values, and more live-out values, requiring additional hardware buffering at the L2 cache banks as is shown empirically in Section 5.2.7.

To achieve tasks with a given size, the task boundary locations must be selected at suitable intervals. Since the number of instructions along different control-flow paths varies, not all tasks will be uniform in length. In addition, as described in the next sub-section, some task boundaries are more desirable than others, leading to additional variability in task size.

3.2.2 Task Boundary Locations

Because task boundary locations determine live-in sets, their selection can impact the effectiveness of distillation. Optimal task selection is still an open problem (and possibly undecidable), but I understand some of the factors that make good task boundaries. Task boundaries should be placed where they minimally constrain optimization. For example, if all uses of a value are removed from the distilled program, the value becomes dead and its creating instruction can be removed, *unless* a task boundary is placed in its original program live range, forcing the value to be computed for the checkpoint. Thus, one goal of task selection is to minimize the number of otherwise dead values whose live ranges are cut[1].

This task selection may seem to be a simple optimization problem, but a circular dependence makes it particularly challenging to solve. To choose task boundaries, the algorithms wants to know which values are dead, but the dead values are the result of optimizations that require knowledge of the task boundaries. This circular dependence is somewhat analogous to the inter-dependence between register allocation and instruction scheduling [9, 31]. My current implementation of the task selector relies on control-flow-based heuristics to select task boundary locations. Integrating the task selector with the optimization passes or performing the process iteratively is an area of future work.

The three heuristics I use have been identified in previous research on speculative multithreading: (1) breaking tasks after function calls (function continuations) [2, 56], (2) in loop headers (loop iterations) [47, 33, 76], and (3) after loop exits (loop continuations) [2]. These points in programs are expected to generally have below average numbers of short dependences crossing them because programmers often encapsulate tightly coupled operations in functions, loop bodies, and loops. These heuristics generally result in

---

1. In addition to constraining approximation by forcing particular values to be computed, task boundaries constrain code transformations that involve code replication (*e.g.*, in-lining). These constraints result from co-location of entries into the distilled program with task boundaries. The proposed hardware implementation maintains a one-to-one mapping from the original code to the distilled program at the entry sites. If an entry site is replicated, only one copy can be mapped, making the other sites unreachable for misspeculation recovery.

reasonable task selection, but this aspect of distillation deserves further attention as some selections are far from optimal.

3.2.3 Specifying Task Ends and Task End Suppression

Tasks executed by slave processors must end at the point the next task will begin. If this invariant is not maintained, the tasks cannot be "stitched" together to be equivalent to a uniprocessor execution. After considering a number of alternative approaches, I settled on something akin to Multiscalar's *stop bits* [72] for my implementation. In this simple approach, each static instructions in the original program is either the first instruction of a task or not, and task beginning instructions are annotated. These single-bit annotations can be stored in a separate region of memory, fetched with the instruction on an instruction cache miss, and buffered along side the associated instruction in the instruction cache. With these annotations it is trivial for a slave processor to stop executing at the beginning of the next task.

Because one goal of my MSSP implementation was to avoid modifying the original program in any way, this approach to marking task boundaries constrains task selection. As the annotations are unconditional, any path from any calling context through an annotated instruction terminates the task. In general, this inflexibility is not a large concern, as control-flow profile information can be used to place task boundaries appropriately along hot paths, and imperfect task selection along cold paths can be tolerated.

There is one common idiom for which this constraint in untenable: tight loops with high iteration counts. If constrained to make either a single iteration a task or the whole loop a task, we are left with the sub-optimal choice of small tasks or very large ones. To handle this problem with minimal additional complexity, I introduced task end suppression.

Task end suppression enables multiple iterations of these small loops to be encapsulated into a single task. To use suppression, a `SUPPRESS` instruction is included in the distilled program. the `SUPPRESS` instruction specifies two things: 1) the PC of an annotated instruction to ignore and 2) the number of times it should be ignored. When executed, the `SUPPRESS` instruction's information is communicated to the appropriate slave processor. For simple loops, the loop in the distilled program is unrolled to a corresponding degree. In Section 5.2.2, I quantitatively demonstrate the importance of task suppression.

3.2.4 Task Selection Implementation

In this sub-section, I describe some of the details of my task selection algorithm (shown in Figure 3.1). All task selection decisions are made using only a call graph and the functions' control-flow graphs annotated with edge profile information. Task selection is performed one function at a time, using a post-order traversal of the call graph to process a function's callees before itself. The task selection algorithm has two main phases: 1) using the heuristics described in Section 3.2.2, the algorithm identifies a set of potential task boundary sites, and 2) a subset of those sites is selected to balance task sizes while trying to construct tasks of a desired size.

```
select_tasks(call_graph, cfg[]):
   foreach function f in a post-order traversal of call_graph:
      regions := []
      regions[0] := copy of cfg[f]
      insert potential task boundaries after loops and calls
      handle_loops(regions) (see Figure 3.3)
      promote_task_boundaries(regions) (see Figure 3.4)
      transform cfg[f] to include the task boundaries in regions
```

***Figure 3.1:* Top-level algorithm pseudocode for selecting tasks.**

Currently, I use the number of dynamic instructions as the metric for load balancing tasks. Clearly, a task's dynamic instruction count is path-sensitive. To evaluate a task's dynamic length in the presence of a potentially unbounded number of cyclic paths, the algorithm considers only acyclic paths; it works on path fragments from most to least frequently executed until each function is covered.

The first step of the algorithm is to make a temporary copy of the function's control flow graph (CFG). This copy of the CFG is traversed and potential task boundaries are inserted at the beginning of blocks following function call invocations and loop exits. Figure 3.2 shows an example CFG before (a) and after (b) this step.

Cyclic control flow is removed by the algorithm shown in Figure 3.3. First, I find all loops and consider each (inside-out) for insertion of a task boundary at the top of the loop. Task boundaries are inserted if the loop body exceeds a certain dynamic size. If a loop header is inserted, the loop body is considered as a separate acyclic region for the purpose of inserting additional task boundaries. If no task boundary is inserted into the loop, the loop is logically collapsed and considered as an acyclic region with a weight equal to the length of the dominant path through the loop body scaled by the average number of iterations executed.

The algorithm for selecting which potential task boundaries to promote as the final task boundaries is shown in Figure 3.4. For each acyclic region, a path with the highest execution count is selected. The dis-

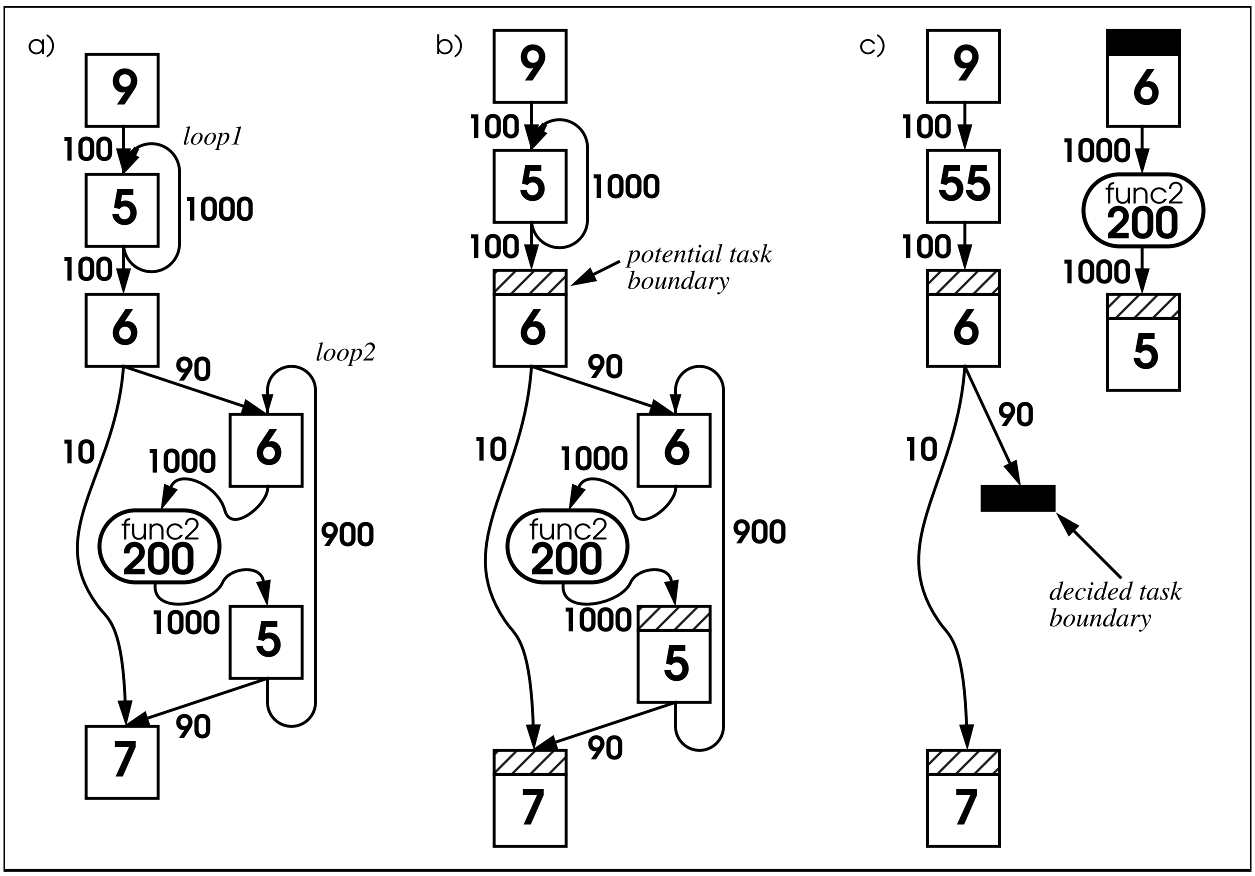


Figure 3.2: Task selection algorithm example. *A function with two loops, the second of which is conditional and calls another function (a). To select tasks the algorithm first inserts potential task boundaries after loops and function calls (b), then acyclic regions are formed (c) by either collapsing loops (e.g., loop1) or by separating the loop body into a new region by inserting a task boundary in the loop header (e.g., loop2). Lastly, a subset of the potential task boundaries are promoted to create the final set of task boundaries (not shown).*

```
handle_loops(regions): /* regions initially has a single region */
   loops := interval analysis of regions[0] to find loop hierarchy
   foreach loop L in post-order traversal of loops:
      whole_path_len := loop_weight(L) / exec_freq(loop_preheader(L))
      iter_path_len := loop_weight(L) / exec_freq(loop_header(L))
      if (whole_path_len < whole_path_threshold):
       substitute L in regions[0] with a block of length "whole_path_len"
      else:
       unroll_factor := loop_path_threshold / iter_path_len
       insert task boundary in L's loop header, with unroll_factor
       remove L from regions[0]
       remove back edges from L
       regions.append(L)

loop_weight(loop):
   weight := 0
   foreach block B in loop:
      weight += block_weight(B)
   return weight

block_weight(block):
   weight = instruction_count(block) * execution_frequency(block)
   foreach function F called by block:
      weight += average path length through F
   return weight
```

***Figure 3.3:** Algorithm pseudocode for creating acyclic regions for task selection.*

```
promote_task_boundaries(regions):
  foreach region R in regions:
    foreach block B in region R (in order of execution frequency):
      select dominant path through B (stopping at promoted task boundaries)
      compute path lengths between potential task boundaries
      promote task boundaries in path, as suggested by Equation 1
      remove un-promoted task boundaries in path
```

***Figure 3.4:** Algorithm pseudocode for selecting potential task boundaries to promote.*

tance between possible task breaks along this path can be trivially calculated. All that is required is to determine the best possible subset. I exhaustively quantify the quality of all sets of tasks by evaluating Equation 1, where Π is the quantity to minimize, T is the target task size, and t_n is the dynamic instruction length of the nth task. This function is agnostic of the number of tasks for a given path, but penalizes tasks that are not close to the desired size; the t_n term in the denominator biases task selection toward oversized tasks. The selected set of task boundaries are promoted to permanent status and the other candidate task boundaries along that path are removed.

$$\Pi = \sum_{n=1}^{N} \frac{(T - t_n)^2}{T \cdot t_n} \qquad \textbf{(EQ 1)}$$

Because the number of combinations grows exponentially with the number of potential task boundaries, I limit the computation required by divide-and-conquer. When there are many potential task boundaries (greater than 10 in the current implementation) and the path is above the target path length, the path is split by promoting the potential task boundary that will come closest to balancing the two resulting subpaths.

Once the dominant path has been handled, I select blocks (in order of execution frequency) that are on paths that have not yet been evaluated. For each block, I trace backward and forward along the most frequently executed path until a promoted task boundary is reached. This path is then evaluated as above for new task boundaries, only considering task boundary sites that have not previously been considered. This process is repeated until all blocks in the function have been covered by paths.

After processing each function, I collect some summary data about the function for use by functions that call this function. If the function has no task boundaries on the dominant path, the length of the dominant path is stored. Otherwise I collect the distance along the dominant path from the function entry to the first task boundary and from the last task boundary to the return. If a function calls functions that have not yet been processed because of recursion or mutual recursion, a task boundary is inserted into the beginning of the function.

Once the final task selection has been decided, the distilled program's control flow graph is transformed in a manner described in the next section.

3.3 Distilled Program Structure

Partitioning the program into tasks is only the first step to constructing the distilled program. In this section, I describe the structure of the distilled program. This structure tries to maximize the flexibility available in constructing the distilled program while maintaining its ability to correctly compute the state expected by the original program. Specifically, the distilled program structure has two main features: 1) it is a distinct code image from the original program necessitating explicit mapping between the two programs, and 2) it employs transition code enabling the distilled program to store data in different locations than the original program.

3.3.1 Mapping Between Programs

Transformations like inlining and constant folding can significantly improve the distilled program's performance but result in a distilled program that is no longer a subset of the original program. Once this subset property is lost, there is no longer a one-to-one mapping between instructions in the distilled and original programs. Instead, by making the distilled program a distinct program image, explicit maps are required to map program counters (PCs) from one image to the other.

This mapping problem can be found in other contexts where an additional copy of the program is generated. Software binary translators and dynamic optimization systems [5, 14, 21, 38, 83], which typically construct optimized versions of traces, construct a hash table that maps an original program PC to an address in a translation cache where the optimized trace resides. At points in the trace where control flow exits the translated code, the original PC is embedded somehow to enable recommencement of interpretation. Hardware-based dynamic optimizers [49, 58] and trace caches [62] use hardware lookup tables to perform the mapping function.

There are four occasions in MSSP that PCs need to be mapped from one program to the other. The first two cases arise from the fact that although the distilled and original programs are distinct program images, they share the same data image. This sharing causes difficulties when program counters (PCs) are stored as data in registers or memory, because the stored value has to point to one image *or* the other. It is simplest if all stored PCs point to the original program. To implement this approach, whenever the master writes its

PC to a register (*e.g.*, the link operation of a call) or jumps to a PC from the data space (*e.g.*, a return, switch statement, or virtual function), the PC will first have to be mapped.

The other two cases when PCs have to be mapped are when the master is restarted after a misspeculation and when the slave task is forked by the master. To support restarting the master, a set of mappings is maintained that encode an *entry* PC in the distilled program for each annotated instruction in the original program. The process for forking a slave is discussed in Section 3.3.3.

3.3.2 Transition Code

Some desirable optimizations, including register re-allocation and scheduling, imply that the data image of the distilled program not exactly match that of the original program. Registers are a valuable resource, and the register allocation used in the original program may be rather inefficient for the distilled program. The ability to re-allocate registers and allocate additional variables to registers enables register moves, saves and restores to be removed from the distilled program and reduces cache bandwidth requirements. Similarly, re-scheduling the code can be beneficial, but it changes the lifetimes of variables in the program.

Although the distilled program may execute faster with a modified data image, the checkpoints must supply the state as expected by the original program. This apparent contradiction can be resolved by the use of *transition code*. Transition code, which is much like the compensation code required for trace scheduling [24], is executed during the transition from the original program to the distilled program (and vice-versa) to map the register and memory state of one program to how it is expected in the other program. I will demonstrate the importance of transition code with two optimizations.

Figure 3.5 shows a loop that contains code that is loop invariant. Task selection inserts a task boundary at the loop header making each loop iteration a task. Associated with the task boundary is an entry that will

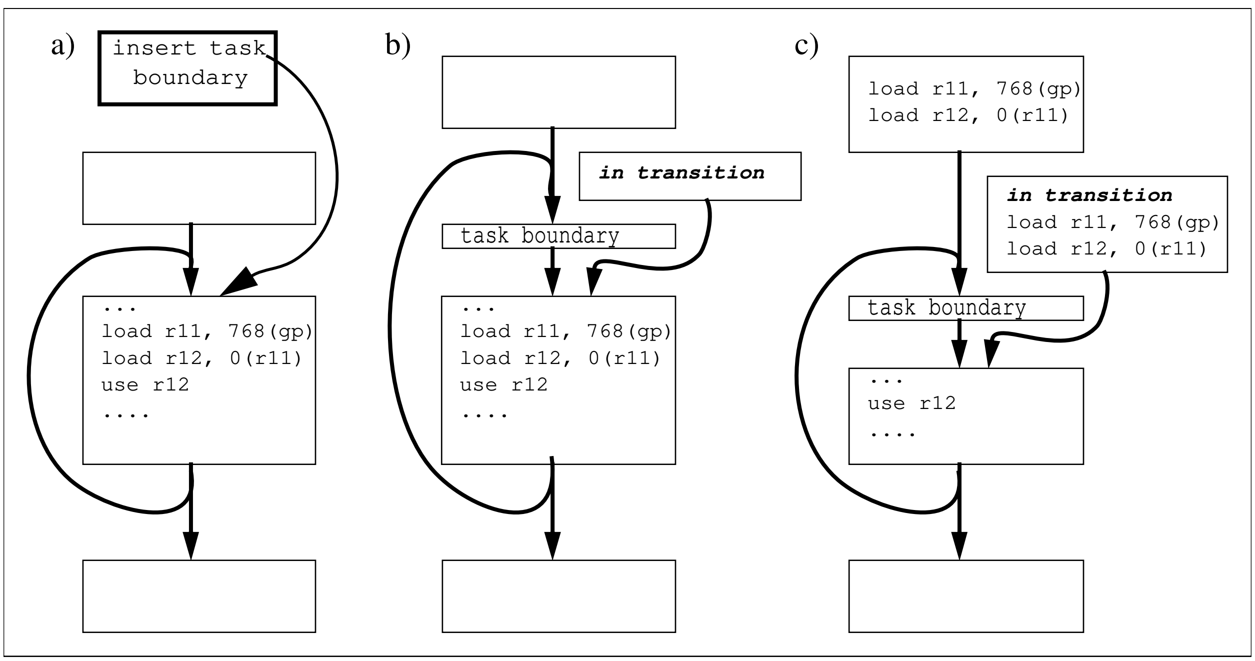


Figure 3.5: Code hoisted across a fork instruction is replicated in the transition code. *Illustrative example shows a loop containing a loop invariant computation (a), which was not hoisted in the original code (perhaps due to an ambiguous memory dependence). Because a task boundary is inserted at the head of the loop (b), hoisting the code out of the loop crosses the task boundary (c). The transition code requires a copy of the hoisted code to fill r12 with the expected value.*

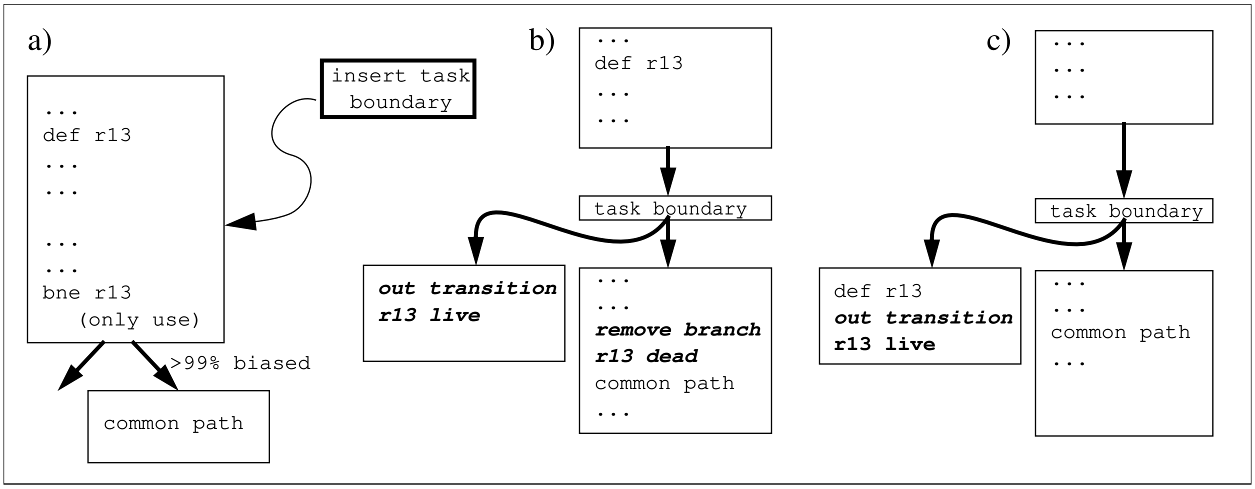


Figure 3.6: Code can be percolated down into transition code if used only in one task of the original program.

The illustrative example shows a widely separated definition (def) of r13 and its only use (a). The use is by a branch that is sufficiently biased that it will be removed from the distilled program, but a task boundary separates the def and the use and therefore r13 is live in the checkpoint (b). Since the def is dead with respect to the distilled program it can be percolated down into the transition code (c).

be used to restart the master if a misspeculation occurs inside the loop. When the loop invariant code is hoisted out of the loop in the distilled program, a discrepancy from the original program's state is created. Without the transition code, `r12` will possibly contain the wrong value. To assure that `r12` has the right value, the loop invariant code is copied into transition code that is executed before the master enters the distilled program proper. The transition code will affect the time it takes to restart the master, but the loop invariant code has been removed from the steady state distilled program execution.

Figure 3.6 shows a case where a variable is live across task boundaries but only into the following task of the original program. In this case, the computation of the variable can be completely delegated to transition code, via partial dead code elimination [40]. The only use of `r13` is a highly biased branch. This branch is removed from the distilled program, but because `r13`'s live range crosses a task boundary it still must be in the checkpoint. Since r13 is not needed in the distilled program, its computation can be percolated downward into the "out transition" code. The result of this transformation is that the master is no longer encumbered by this computation, as this transition code will be executed by the slave before executing the task in the unmodified original program. While the master's execution is frequently the critical path, the slave's execution is exceedingly tolerant of additional overhead.

3.3.3 Resulting Structure

The resulting structure of the distilled program is shown in Figure 3.7. Two special instructions are used to manage the transition from the distilled program to the original program. The first, using a branch-style format, is the `FORK` instruction, which marks the task boundary in the distilled program. Upon encountering a `FORK` instruction, the master processor continues on the fall-through path. In parallel, an idle slave processor is instructed to begin fetching at the target specified by the fork instruction, which is the starting PC of the out transition code.

At the end of the out transition code, the second special instruction (`VERIFY)` is inserted. The `VERIFY` instruction signals to the hardware that a transition to the original code is to be made and that collection of live-in values should commence. Because the distilled program may not be spatially close to the original program, the jump distance will likely be larger than the maximum displacement in a branch format instruction. Typically indirect branches are used in such circumstances but are problematic here because all

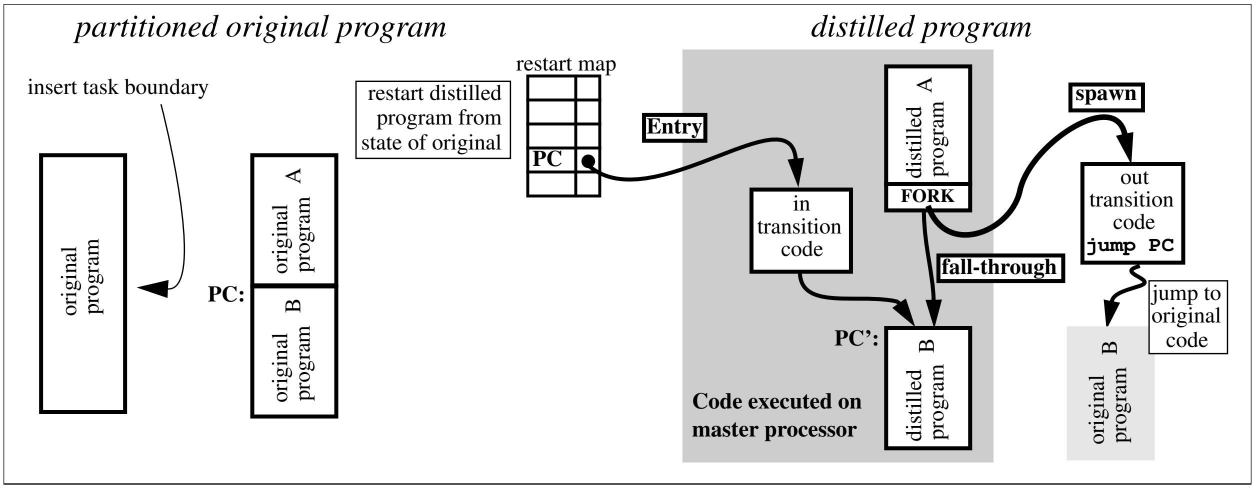


*Figure 3.7: **The distilled program structure supports checkpointing and misspeculation recovery.** Basic blocks ending in* `FORK` *instructions continue executing the fall-through path on the master processor and spawn task execution on an idle slave processor. Entries, with associated transition code, enable restarting the master after a misspeculation.*

registers might need to contain the values expected by the original code. As a result, I specify the target PC using a combination of the target bits in the `VERIFY` instruction and the 32-bits that follow in the instruction stream.

3.4 Framework for Automatic Program Distillation

In this section, I describe the approximation framework (program distiller) used in this dissertation. The structure of this distiller is not intended for a real implementation. Rather, it is intended as a fast path to identifying the important factors in designing a real approximator.

The implementation that I envision for use in real systems is one that transparently constructs distilled programs at run time. By doing the construction at run time, the approximator has access to the most relevant profile information: that from the current run of the program. Accurate profile information allows the program to be aggressively approximated while maintaining a low task misspeculation rate. Of course, run-time construction of the distilled program is not mandatory, but I believe that some run-time control mechanism is. Otherwise, because profile information is so highly leveraged, an unrepresentative profile could lead to the situation where a task misspeculation occurs for every task. Minimally, the run-time system should recognize recurring task misspeculations and shift out of MSSP mode for a period of time. Furthermore, it is easy to imagine distiller implementations with on-line and off-line components, where a candidate distilled program is constructed off-line and refined on-line as more representative profile information is collected.

To approximate the quality of a run-time approximator without its complexity of implementation, I have built a static approximator, which I provide with profile information from the run of the program with which the evaluation is to be performed. While this self training is optimistic in some respects, this methodology can not be considered to achieve a strict upper bound on performance. A real run-time system has two advantages over my static implementation. First, a run-time system can exploit phase behavior, deploying different versions of code as usage patterns in the code change. Second, a run-time system can evaluate how well the constructed distilled program is working and use the feedback to tune the approximation.

My implementation of the approximator is a binary-to-binary translator for the Alpha architecture [68], except rather than working on binaries it works on already loaded static program images. The system is built inside a simulator built using the libraries from the SimpleScalar toolkit [10]. The simulator loads the static image from the program binary into memory, where it is readable by the approximator. The approximator does not modify the original static image but creates a second one that exists in parallel in a different range of memory.

The internal representation (IR) used within the approximator is very close to the Alpha instruction set, as was done in the Alto binary translator [54]. This IR makes the process of translating into and out of the IR trivial. Though many of the transformations are performed on the IR directly, some higher level constructs (*e.g.*, loops, variable live ranges) are reconstructed from this low level IR.

The program optimizations can be broken into two classes—speculative and non-speculative—summarized in Figure 3.8. The speculative optimizations—enabled by the approximate nature of the distilled program—are applied first. Their application is controlled by "correctness thresholds" specified to the distiller and profile information described in Section 3.4.1. Once these speculative optimizations have been applied, the resulting code generally has new opportunities to apply non-speculative optimizations. These transformations and optimizations (described in Section 3.5 below) are largely drawn from the existing literature in optimizing compilers, but some of them need to be altered slightly to support the distilled program structure. An example of such an interaction for liveness analysis is described in Section 3.4.2.

3.4.1 Guiding Root Optimizations with Profile Information

The critical advantage of approximate code is that it need only be correct in the common case. As a result, the approximator can consider a broader range of transformations and optimizations than a traditional optimizing compiler. In a traditional compiler, any optimization that is correct and deemed profitable can be applied. In approximation, correctness is not mandatory, it only contributes to which optimizations are profitable. For the root optimizations, which do not preserve correctness, I use profile information to estimate an optimization's impact on correctness.

I assign a required correctness threshold for each optimization. To apply a given optimization, the profile information must indicate that the optimization will preserve correctness a fraction of times greater than the threshold. For example, a branch removal threshold of 98% may be set that will remove a branch with a 98.2% static bias, but not one with a 97.6% static bias. Thresholds for each optimization can be tuned individually; the thresholds available on the current distilled prototype are shown in Table 3.1.

By varying these thresholds a whole range of approximate programs can be generated that trade off, to varying degrees, accuracy for common-case performance. As will be demonstrated in Section 5.2.1, many

root (speculative) optimizations:	**supporting (non-speculative) optimizations:**
branch elimination	dead code elimination
indirect-to-direct call conversion	inlining
long dependency store removal	register re-allocation
idempotent operation removal	register move elimination
silent store removal	save/restore removal
	loop unrolling
	global pointer computation removal
	control-flow simplification
	stack pointer optimization
	constant folding

Figure 3.8: Summary of root and supporting optimizations. *Root optimizations are applied first and expose new opportunities for the supporting optimizations.*

TABLE 3.1 **Correctness thresholds for program distiller optimizations.**

Parameter	Typical Value	Explanation
Branch Elimination Threshold	99% biased	The fraction of dynamic instances that must go to the dominant target for the branch to be removed.
Idempotent Optimization Threshold	99% correct	The fraction of times that an instructions output must match an input for it to be removed.
Silent Store Threshold	99% correct	The fraction of times that a store must have overwritten the value in memory with the same value for it to be removed.
Long Store Definition	1000 instructions	The store to load distance at which a dependence is considered a long dependence.
Long Store Elimination Threshold	99% long	The fraction of store instances that have to be classified as long for the static store to be eliminated.

of the optimizations are largely insensitive to the supplied threshold value because where the optimizations are applicable they will be correct almost all of the time. It is mainly the branch elimination threshold that has leverage on distillation. Sensitivity analysis in Section 5.2.9 shows that for most benchmarks, setting this threshold in the 98 to 99 percent range generally results in performance rivaling that of the best configuration.

Currently I do not consider the benefit of an optimization, only its cost (in terms of induced misspeculations). If benefit can be predicted, the approximator can potentially avoid applying optimizations that incur misspeculation but do not significantly improve common-case performance. This approach is not done currently as it is difficult to predict the impact that a transformation will make on later optimizations. Obvious future work includes developing techniques to predict benefit or support for generating and evaluating multiple versions of a piece of code.

3.4.2 Implications of Distilled Program Structure on Liveness Analysis

As discussed in Section 3.3, the distilled program is structured to support transitions to and from the original program; this structure impacts how liveness analysis is performed. Some operations (*e.g.*, a branch that is never taken) in the original program will be removed from the distilled program. This generally creates new opportunities for dead code elimination in the distilled program. It is important, though, to not consider dead any value that will be needed as a live-in to a task in the original program.

When doing dead code elimination on the distilled code we must consider paths through the transition code from the inserted `FORK` instructions. The `VERIFY` instructions at the end of the transition code are considered as uses of all variables live at that point in the original code. To minimize the number of live-in variables, liveness analysis is performed on the original code only considering paths that have been observed to be exercised. By ignoring unobserved paths, the analysis is more precise for the active paths, but misspeculations may be introduced if the unobserved paths are executed.

3.5 Implementation

In this section, I present the flow of the approximator implementation. This implementation should not be construed as *the* way an approximator should be built, but rather as documentation of the system that I have built for the purpose of interpreting the results in Chapter 3. As most of the transformations are well understood in the existing literature—generally covered in [53]—I intend this section to be an overview, only delving into details when peculiarities of approximation are important.

The approximation implementation can be broken down into a sequence of 5 stages: initialization, instruction-level optimizations, dead code elimination, function-level optimizations, and output. I describe each of these in the following sub-sections and then briefly discuss the implementation's performance. A flow chart that overviews the distiller's operation is shown in Figure 3.9.

3.5.1 Initialization

At the end of the initialization phase an un-optimized version of the internal representation has been constructed and partitioned into tasks. This phase consists of the following sub-steps.

Build Internal Representation (IR). Internal representations for each instruction can be created from the static program. Using a control-flow edge profile, IR instructions are assembled into blocks, which in turn are assembled into control-flow graphs (CFGs) on a per-function basis, and a call graph is constructed. Only instructions on paths that were observed to execute are considered. Like the FX!32 binary translator [14], my distiller identifies candidate targets of indirect branches through profiling.

Liveness Analysis. At this point the IR exactly matches the original program, so I can identify the set of live variables at all points of the original program that are expected to be reached. This set is found by performing an inter-procedural liveness analysis and is used to provide the live-in sets for each `VERIFY` instruction associated with task boundaries. This means of generating the `VERIFY` live-in sets is slightly conservative, in that a value identified as live may be used in the future but not within the task. To facilitate the liveness analysis, I first match up register saves and restores to track liveness through stack spills.

Task Boundary Selection. Using the heuristics and the algorithm described in Section 3.2.4, the location of task boundaries are selected. Then the `FORK` instructions are inserted and the CFG transformed to include the in and out transition code blocks for entry and verification.

3.5.2 Instruction-level Optimizations

The second phase is local optimizations that typically modify or remove a single instruction, or, rarely, a small number of clustered instructions. These optimizations are largely independent and could be reordered without impacting their efficacy.

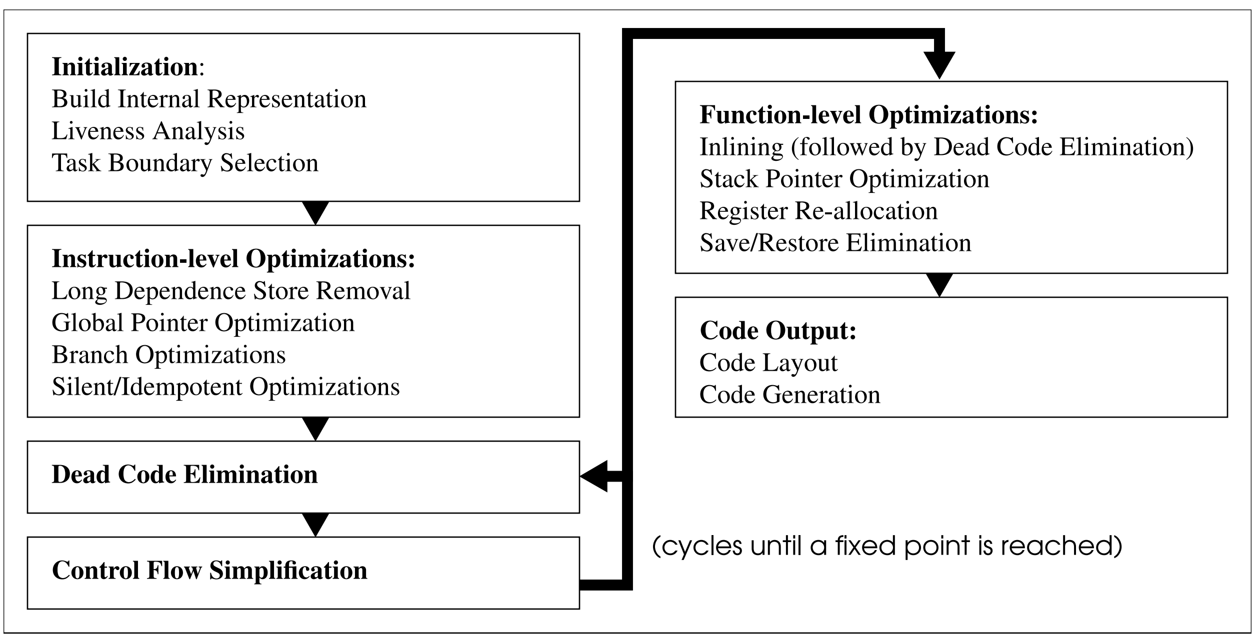


*Figure 3.9: **Overview of phases of program distiller implementation.***

Long Dependence Store Removal. As mentioned in Section 2.1.3, if the distance between a store and its first use exceeds the task window, then the distilled program need not perform the store, because, by the time the use is executed, the original code will have already updated architected memory. Using a memory dependence profile that includes a histogram of store-load distances, I identify candidates for removal. Two parameters are specified to control this optimization: 1) a specified distance, and 2) the fraction of store-load distances that must exceed this distance. If a store satisfies these criteria, it is removed from the distilled program.

Global Pointer Optimization. In the standard Alpha calling convention, the global pointer (GP) is not a callee saved register. This means that, unless the compiler verifies that the GP is unmodified in a called function, it must recompute it after a return before it uses it again. Perhaps due to separate compilation or from the use of libraries, the code studied is rife with GP re-computations, even though only one value for the GP is ever used. The approximator assures itself that the same GP value is computed at every instance, then removes the GP computations and ensures that the GP is never modified.

Branch Optimizations. Removing and simplifying branches is a major source of performance improvement as many branches are heavily biased to a single target. If the bias of a direct or indirect branch (as recorded in the edge profile) exceeds a specified threshold, the branch is removed along with the CFG edge to the non-dominant path. If this is the only path to the target block then the two blocks can be unified; otherwise, nothing is done at this time, and, if necessary, an unconditional branch is inserted at code layout time. If an indirect call calls a single function more than a specified fraction of the time, a direct call can be substituted.

Silent/Idempotent Optimizations. As was noted in [43], some stores frequently write a value to a memory location that is already holding that value. By profiling for this behavior, we can identify stores that are silent more than a specified fraction of the time. These stores can be removed. In addition, some static instructions consistently produce as output one of their input values. These instructions, which I refer to as *idempotent operations*, have not been previously studied to my knowledge. These sources of this behavior fall into three classes: 1) truncating a value to a smaller bit-width (*e.g.*, from 64b to 32b because the program casted a `long long` to an `int`) but the upper 32b bits were already zeroes, 2) logical operations where one operand is generally a superset of the other, and 3) conditional moves with highly biased predicates. If the important input operand uses the same architectural register as the output, the instruction can simply be removed. Otherwise it is replaced by a register move instruction, which generally can be removed by register re-allocation. Silent register writes that are not idempotent operations are not optimized because they interact poorly with register re-allocation (described below in Section 3.5.4).

3.5.3 Dead Code Elimination and Control-Flow Simplification

The removal of instructions by the above instruction-level optimizations often renders other instructions dead. By doing an inter-procedural liveness analysis, we can identify the dead code and remove it. The removal of dead code occasionally removes all instructions from a given basic block. When this occurs, the block is removed and all input arcs in the CFG are redirected to the block's single successor (if a block has multiple successors it must be terminated by a branch and therefore be non-empty).

Occasionally, the removal of blocks will cause all paths from a branch to lead to the same block. When this occurs, the branch is no longer necessary and can be removed. In this way, un-biased, hard-to-predict branches can be removed. Removing the branch will generally render new code dead, resulting in additional benefit from re-running dead code elimination. As the dead code elimination may result in new control-flow simplification opportunities, I alternate these optimizations until a fixed point is reached. Most benchmarks benefit from as many as 4 iterations of this process.

3.5.4 Function-level Optimizations

In this stage, I apply optimizations that have effects that span a function. I optimize each function separately, in a post-order traversal of the call graph (*i.e.*, leaf functions first).

Stack Pointer Optimization. Most functions use the stack to store locals and saved registers. Non-leaf functions (*i.e.*, ones that call other functions) allocate space on the stack by adjusting the stack pointer (SP) on entry and exit, so that called functions use a different region of the stack. During approximation, non-leaf functions are frequently converted into leaf functions, either by removing paths containing calls or through inlining (see below). In most of these cases, the pair of SP manipulations can be removed by adjusting the offset of loads and stores that are relative to the SP. This optimization is not applied if any unrecognized manipulations of the SP are present (*e.g.*, copying the SP to another register or other updates to the SP).

Register Re-allocation. After dead code elimination and inlining, there are often opportunities to make better use of the registers and remove register move instructions. To accomplish this, I first extract the register def-use webs [53] (*i.e.*, symbolic registers) using data-flow analysis to match the reaching definitions with the uses. This is greatly affected by the distilled program's structure, as entries to the distilled program must be considered as definitions and `VERIFY` instructions as uses of all live registers. The webs are used to build an interference graph, which encodes which webs cannot be allocated to the same registers. Then webs that are live-in or live-out to the function or are arguments or the return value of a called function are pre-colored to the architectural register they were assigned in the original program. Move instructions are identified. Iterated register coalescing [29] is then performed, which removes most of the unnecessary register moves.

Remove Saves/Restores. During register re-allocation, when assigning registers to the interference graph, I attempt to use the minimum number of registers and only use the callee-saved registers if necessary. Typically, because register pressure is lower in the approximate code, many saves and restores can be avoided. Because none of the currently implemented optimizations potentially increase register pressure, no spilling is necessary beyond what was performed in the original code.

Inlining. After processing a function, I consider inlining it into its callers. This is always applicable if the function is called from a single call site. Otherwise, because inlining would require code replication, it can only be considered for leaf functions that do not contain any task boundaries or non-return indirect branches (in the approximate version) due to limitations of mapping from the original program to the distilled program. I also limit the static size of functions to prevent code explosion and limit optimization time (for many of the optimization implementations time scales super-linearly with instruction count). When a function is inlined, it is copied into the CFG of the calling function. Return blocks are converted into unconditional jumps to the return target. The call instruction is removed and inserted in its place are a sequence of instructions (LDAH and LDA) that compute the expected return address, which is sometimes a live-in to the original code; when unneeded this computation is eliminated by dead code elimination.

3.5.5 Code Output

Once optimization is complete, the IR needs to be converted back to a new static image. This process involves code layout, recomputing branch target offsets, and generating code.

Code Layout. With the available profile information, it is natural to perform a profile guided layout of the code. I use simplified version of the Pettis and Hanson algorithm [59] that performs intra-procedural layout to minimize taken branches along the dominant control-flow paths. In the layout process, I can identify where unconditional branches will be needed and allocate space for them. At this time I construct the map that maps restart entries, indirect branch targets and return targets in the original program to their corresponding location in the distilled program. Transition code for the function is laid-out after the function to minimize fragmentation in the master's instruction cache.

Code Generation. Having computed the new program counters (PC) (in a separate region of memory from the original code segment) for each instruction of the IR, the next step is to compute the new branch offsets for control instructions. With these determined, the relatively straight-forward process of converting the IR back to Alpha instructions is performed.

3.5.6 Performance

Because the approximator can ignore the significant fraction of the program that is never used and typically reduce the remaining portion of the program to less than half its original static size, the approximator can be very efficient. Despite almost completely ignoring the performance of the approximator's implementation, it typically can generate a distilled program in a small number of seconds (< 5), even for non-trivial programs like `gcc`. I am optimistic that a real implementation can be constructed to have a sufficiently small overhead to be viable.

3.6 Chapter Summary

In this chapter, I presented the two main requirements of the distilled program: 1) to break its execution into tasks, and 2) to efficiently and largely accurately compute the program state expected at task boundaries. I discussed how placement of task boundaries can be critical to performance both with respect to impact on optimization and on task size. I described my control-flow based heuristics for selecting candidate task boundaries and my algorithm for balancing task sizes.

In addition, this chapter describes the structure of the distilled program and my prototype implementation of a program distiller. The distilled program structure includes features—program counter maps and transition code—that enable the distilled program to be started from the state of the original program and vice-versa. These features allow the state of the distilled program to be decoupled from that of the original program, increasing the flexibility with which the distilled program can be optimized. The prototype distiller is a binary-to-binary translator that is largely based on existing compiler techniques, except that some of the optimizations need not preserve correctness and the idiosyncrasies of the distilled program structure must be maintained.

Chapter 4

Implementing the MSSP Paradigm

In this chapter, I present the required mechanisms and one possible implementation of the MSSP paradigm. This implementation of the execution paradigm is by no means the best possible one, but it is complete and demonstrates all of the necessary functionality. Beyond completeness, the proposed implementation focuses on maximizing performance and effectively tolerating the latency of inter-processor communication.

First (in Section 4.1), I discuss the mechanisms the paradigm requires at a high level. Then (in Section 4.2), I present an analytical model of the execution paradigm to allow reasoning about the execution paradigm. Section 4.3 covers the themes that guided the design of the implementation that follows. Then (in Section 4.4), I describe the high-level architecture and some program data that validates some of the design decisions. Finally (in Section 4.5), I present some details on the mechanisms themselves, where they differ from prior work, and conclude with a chapter summary.

4.1 Required Functionality

Besides the construction of the distilled program—described in the previous chapter—the MSSP paradigm requires a number of mechanisms to function properly. The requirements listed reflect my selection of a "block reuse"-style mechanism for verification and commit; for alternative mechanisms some of these requirements may be different. These mechanisms are:

Timestamps. The relative logical ordering of tasks is vital to providing the appearance of a sequential execution. Tasks are assigned sequence numbers, which I call timestamps, by the master according to their order in a sequential execution. Slave tasks will have the same timestamp as the corresponding task executed by the master.

Checkpoint buffering. In executing the distilled program, the master processor performs writes to registers and stores to memory. These writes are not intended to be externally visible. Instead, they need to be buffered and associated with the timestamp of the task that performed the write. These checkpoints can be deallocated when the corresponding task in the original program is committed.

Live-out Buffering. Unlike the master, slave tasks update the architected state, but not until they have been verified. Because tasks may be significantly larger than the internal buffering available in most processor cores, I expect slave processors will "speculatively retire" instructions and buffer the resulting stores in a live-out value buffer. If task misspeculations always recover to the beginning of a task (as is done in my implementation), then speculative retirement can be allowed to retire registers normally (*i.e.*, irrevocably updating the local register file), by keeping a checkpoint of the state of register file as of the beginning of the task.

Live-in Buffering. As a slave task executes, it must keep track of the values that the task used that it did not create, so that these values can be verified. Not only do the names of these live-in values need to be buffered, but the values themselves as well.

Register Communication. Because checkpoint and live-out state includes register state, I need a mechanism to read register values from a processor and to load register values from a checkpoint.

Checkpoint Assembly. When a slave task performs a memory access, the returned value should reflect the most recent checkpoint value from a task with a timestamp that precedes or is equal to that of the requesting processor. If no checkpointed value is available, the value from the architected state is returned. Because most architectures support memory operations at multiple granularities (*e.g.* bytes, 32-bit words, etc.) this process can require assembling multiple, potentially overlapping, small writes from separate tasks along with architected data to satisfy a large read operation.

Live-in Verification. To see if a task can be committed (*i.e.*, its results reused), the hardware has to check to see if all of the live-in values are correct. I do this checking by comparing all of the live-in values to the architected value at the same storage location. This comparison has to appear as if it were done atomically, and the task is verified only if all live-in values match. After live-in comparison is complete, the live-in buffers can be cleared.

Live-out Commit. To avoid memory ordering violations in most consistency models, a task's speculative state has to be committed atomically (or have the appearance of being committed atomically) with the live-in verification.

Task End Condition. Each slave processor must know where its current task ends and the next task begins.

Maps. PC's of entries and indirect branches need to be mapped from the original program to the distilled program; PC's of transitions to the original program and link addresses need to be mapped from the distilled program to the original program.

Although all of these mechanisms are required for correct execution, they do not have equivalent impact on performance. As I show in the next section, only mechanisms that affect the throughput of the master processor and verification/commitment must perform well. The slave executions of tasks are tolerant of additional latency.

4.2 Analytical Model

I now present a simple analytical model to allow us to reason about performance. Our model makes the following simplifying assumptions:

1. All tasks are equivalent and have execution time **E**.

2. Distilling the program results in a speedup of α; distilled program segments execute in **E/α** time.
3. Verification of live-in values and commitment of speculative data can be done in less time than distilled task execution, so that verification/commit is never a bottleneck. In practice, there is an α beyond which no further speedup can be achieved because the execution is verification/commitment limited.
4. There is an initiation latency **I** between when a fork instruction is executed by the distilled program and when the task begins. This latency accounts for inter-core communication latency, time to execute transition code, and any additional execution latency incurred due to branch mispredictions or cache misses not observed by a sequential execution.
5. There is a binomial distribution with some probability **P** that a checkpoint received by a task will be correctly verified.[1]
6. Misspeculations are detected with a latency **D** after the previous task has been completed. This latency accounts for the time to update architected state and the inter-core communication required to check the misspeculated task's live-ins.
7. Restarting the distilled program takes a latency **R** after a misspeculation has been detected. This latency accounts for any inter-core communication to transfer architected state and for the time required to execute transition code.
8. Additional slave processors are always available. Thus, verification is on the critical path only for tasks that correspond to distilled program segments that produce incorrect checkpoints.

The original execution time for a program composed of **N** tasks is **NE**. The execution time of each task in the MSSP execution depends on whether its segment in the distilled program produced a correct checkpoint. If so, the task's execution time is that of the distilled program's segment, **E/α**. If not, the task's execution time is its latency, **E**, plus the initiation, detection, and restart latencies, **I+D+R**. (For algebraic simplicity, I group these terms into a single normalized overhead term, **O = (I+D+R)/E**). These events occur at a frequency of **P** and **(1 - P)**, respectively. Thus, the total execution time is **N(PE/α + (1-P)E(1+O))**, and speedup is given by:

$$speedup = \frac{time(sequential)}{time(parallel)} = \left(\frac{NE}{N\left(\frac{PE}{\alpha} + (1-P)E(1+O)\right)}\right) = \frac{1}{\frac{P}{\alpha} + (1-P)(1+O)}$$

The resulting equation has three free variables: α, **P**, and **O**. Figure 4.1(a) shows that if I assume the normalized overhead, **O**, is 1 (*i.e.,* equal to the task execution time), then ***speedup is super-linear with prediction accuracy***. As is expected, at low prediction accuracies slow-downs are incurred. At high accuracies (*i.e.,* **P** > 0.98), ***performance closely tracks the performance of the distilled program***. Sensitivity to normalized overhead is shown in Figure 4.1(b). This plot demonstrates that ***the architecture is largely insensitive to inter-core latency*** when prediction accuracy is high.

To summarize, the accuracy of the distilled program is paramount, as it decouples the throughput of the execution from the latency of the individual tasks. In turn, this decoupling allows the paradigm to be tolerant of communication latency if checkpoint construction and verification/commitment are centralized (explored in the next section). With an accurate distilled program, the rate that checkpoints can be constructed by the master and the verification and commitment throughput become the main limiters of performance.

1. The correctness of checkpoints is assumed to be independent and identically distributed (IID).

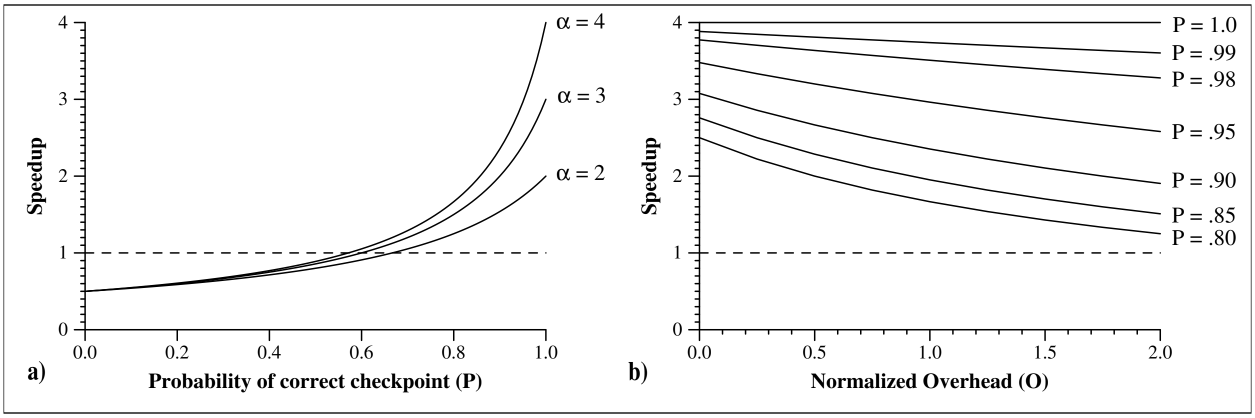


Figure 4.1: Performance predicted by the analytical model. *(a) Speedup is super-linear with checkpoint prediction accuracy, and, at high prediction accuracy, performance tracks that of the distilled program (results shown for O=1). (b) The architecture is insensitive to inter-processor communication latency (captured by parameter O) when checkpoint prediction accuracy is high (results shown for* α *= 4).*

4.3 A Guiding Theme: Tolerating Inter-processor Communication Latency

Having described the implementation's requirements (in Section 4.1) and abstractly explored the performance characteristics of the paradigm (in Section 4.2), I am prepared to discuss the guiding theme that I used to make implementation decisions: the ability to tolerate inter-processor communication. Since the MSSP paradigm breaks an inherently sequential program to execute it on multiple processors, there is invariably going to be some communication. With this fact in mind, I quote David Clark:

> *"Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God."*

With clock frequencies increasing, communication latency between processors on the same die is likely to be in the tens of cycles [48]. Short of designing smaller cores, to allow them to be placed closer, there is little that can be done about this latency. Hence, one major goal of this execution paradigm is to be very tolerant of this inter-processor communication latency.

Latency can be effectively tolerated when it is not on the critical path. As we saw in the previous subsection, assuming the distilled program is accurate, there are two main critical paths in the execution paradigm (Figure 4.2a): (1) the master's execution of the distilled program, and (2) the rate at which tasks can be verified and committed. **All other latencies only slow down the execution of individual tasks on slave processors** (Figure 4.2b). This slowdown will increase the occupancy of the slave processors, which is solvable by making more slave processors available[2]. The task misspeculation detection time is also increased, but this latency is not a major concern with low task misspeculation rates.

The impact of communication latency on the master's execution is minimized by performing it all on a single processor. By caching the speculative memory image in the master processor's local data cache, the master generally stalls for off-core communication only when a traditional uniprocessor execution would. In addition, the MSSP related tasks are performed in the background. For instance, check-pointing is done completely in hardware without additional instruction overhead. Only the special fork instruction is used in the distilled program to indicate where in the execution the checkpoint should be taken. Collection of the

2. There is a subtle interaction between task commit latency and task misspeculation accuracy. When the distilled program is constructed such that it does not produce the value for a long dependence (Section 2.1.3), misspeculations can occur if the commit latency exceeds the dependence length.

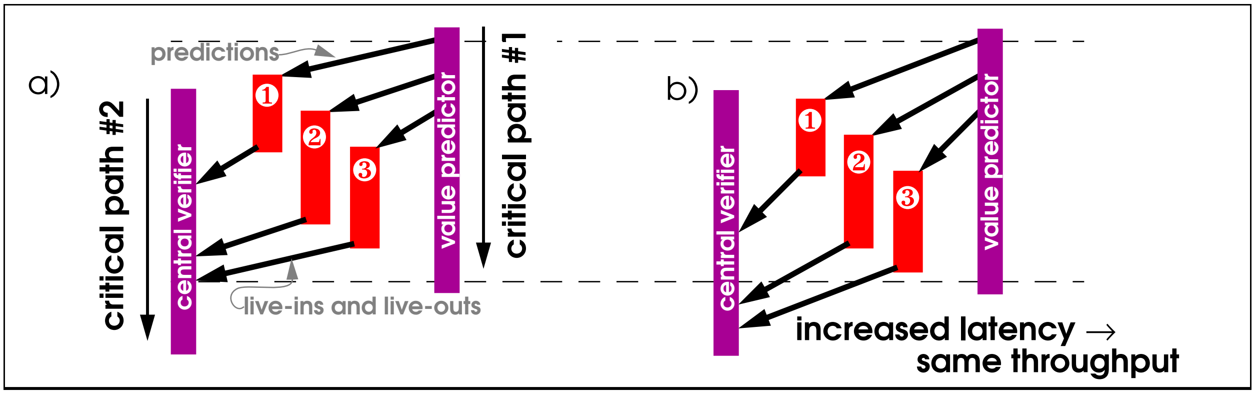


Figure 4.2: Critical path through the MSSP execution. *a) there are two critical paths through an MSSP execution: 1) the rate the master can execute the distilled program to compute the necessary value predictions, and 2) the rate that tasks can be verified and committed. b) If an implementation can avoid serializing latency on these two critical paths, increasing the communication latency should not affect task throughput, if live-ins are accurately predicted.*

checkpoint information should be done in a manner that minimally interferes with distilled program execution; when data is read from storage arrays (*e.g.*, the register file), reads of checkpoint values should be done in the background to avoid stalling the execution of the distilled program.

Similarly, inter-processor communication should be kept out of the verification and commit critical path. I accomplish this act—in the same manner as for the master's execution—by centralizing the process. I propose a centralized verification and commit unit at the first shared level of the cache hierarchy (the L2 cache in the proposed implementation). The slave processors will have to communicate their live-in and live-out values to this central commit unit, but **the communication from multiple slave processors can be performed in parallel**, overlapping the latencies. Once the data is present at the L2 cache, it can be verified and committed without crossing the interconnect.

To summarize, a major goal of my implementation is to tolerate inter-processor communication latency in the two critical paths of the execution paradigm: the master execution and the verification/commitment unit. In the next section, I describe the architecture that resulted from this approach.

4.4 An MSSP Implementation

As mentioned earlier in this dissertation, the MSSP paradigm can be implemented on an enhanced chip multiprocessor. So it should not be a surprise that the proposed implementation looks much like previously proposed chip multiprocessor architectures [7]. As seen in Figure 4.3(a), it consists of a number of processors, each with local first-level instruction and data caches, a unified L2 cache (banked to provide sufficient bandwidth), and an interconnection network to connect them all. Except for the addition of a global register file (GRF), it appears to be a standard CMP, at least at this level of detail.

In this implementation, I made the following design decisions:

- All processors are physically equivalent; a mode bit distinguishes their role (either master or slave) in the execution. This decision avoids the complexity of designing a second core.
- Checkpoint, live-in, and live-out data must be stored in a central place with the architected state; I selected the L2 cache as it already contains much of the active architected data. The L2 is enhanced to include the hardware necessary to assemble checkpoints and verify and commit task data. Since a task's live-ins may be physically distributed across multiple L2 cache banks and must be verified simultaneously, a distributed commit process is required.

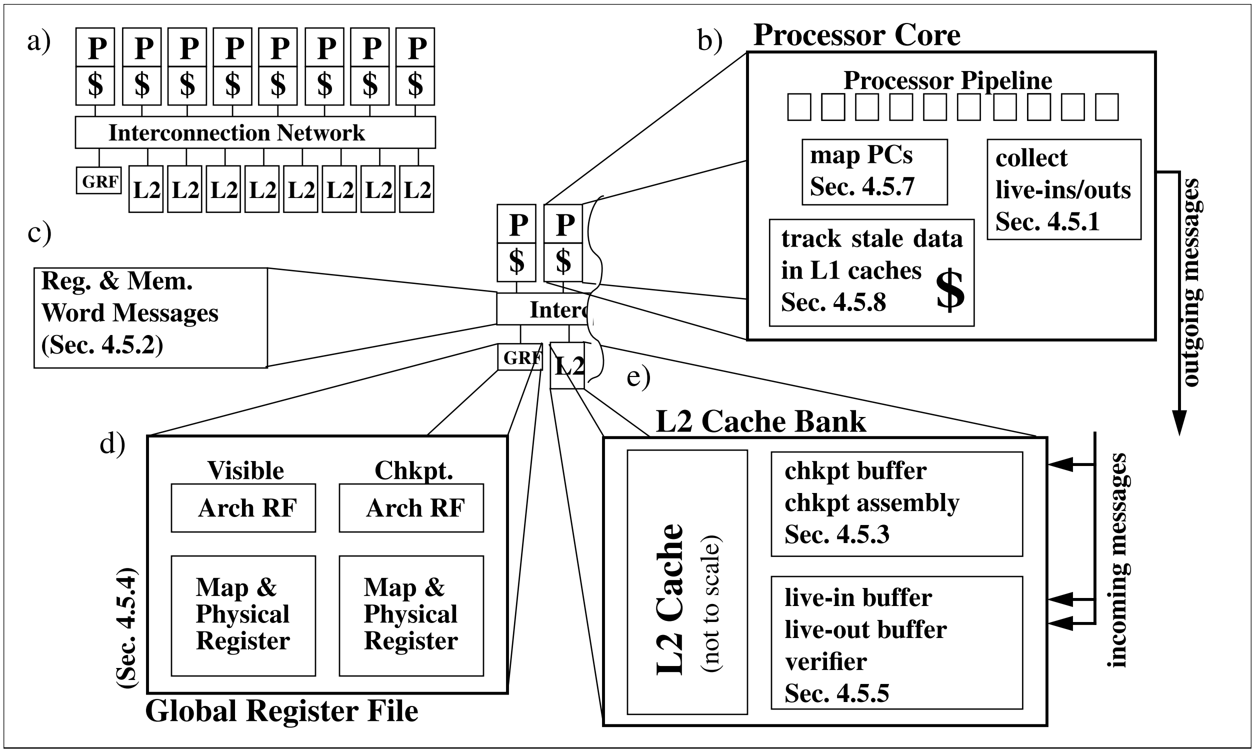


Figure 4.3: ***Block diagram of the MSSP hardware.*** *(a) chip multiprocessor consisting of processors with private instruction and data caches, a shared banked L2 cache, and a global register file (GRF), (b) the processor core has been enhanced to map program counters, collect live-in and live-out values, and track speculative data in the private caches, (c) the interconnection network supports new kinds of messages for efficiently moving register values and parts of cache lines, (d) the GRF stores retired and speculative versions of registers for the original and distilled program, and (e) the L2 cache banks include MSSP specific hardware for managing checkpoint data, live-in and live-out values.*

- There is no direct inter-processor communication. The master sends checkpoint data to the L2 cache banks where it can be used to satisfy requests by slave processors. Slave processors send all live-in and live-out values to the L2 cache banks where the data is verified and committed. Processors can cache speculative data in their local (L1 data) caches. In this way, a slave need only cache data currently relevant to itself.
- Task live-outs and checkpoint differences are “written through” to the L2. To more efficiently use interconnect bandwidth, multiple values are accumulated for a particular bank before a message is sent.

In the following sub-sections, I overview the MSSP-specific mechanisms (Section 4.4.1), referencing the detailed descriptions at the end of this chapter, describe the operation of the implementation (Section 4.4.2), and provide some application data to justify these design decisions (Section 4.4.3).

4.4.1 Mechanism Overviews

Figure 4.3.b-e shows an exploded view of the elements of the MSSP implementation, highlighting the differences in each element. I overview these differences here.

Processor Core. Besides support for the `FORK` and `VERIFY` instructions and for tracking the task number, the processor core has been enhanced in three major ways: (1) MSSP adds support in the branch resolution path for mapping indirect branches from the original program to the distilled program (Section 4.5.7). (2) At the retirement stage, MSSP adds structures for tracking the registers and memory locations that have been read or written, to capture task live-in and live-out values (Section 4.5.1); these values are sent across the interconnection network using special register and memory word messages (Section 4.5.2). (3) MSSP requires a mechanism to load register file contents at the beginning of a task and to read them at the end of a task (Section 4.5.9). In addition, the private caches track which lines contain speculative data; blocks holding speculative data are purged between tasks (for slaves) or periodically "refreshed" (for the master) to ensure that the processors see recent updates to architected state (Section 4.5.8).

Global Register File. The global register file (Section 4.5.4) keeps two distinct register files: *actual* and *checkpoint*. The actual registers are the true architected state, which is programmer visable. The checkpoint registers are those computed by the execution of the distilled program. These register files are distinct because register re-allocation may have been performed by the distilled program. In addition to an architected register file that holds the view of these registers as of the end of the last retired task, each register file speculatively buffers the writes by in-flight tasks. These speculative updates are held in a structure much like the register map/physical register file found in most out-of-order processor cores. A verifier included in the global register file, compares the two files for each task to validate the task's register live-ins (Section 4.5.5).

L2 Cache Banks. The L2 cache banks are similar in function to the GRF, but are organized differently due to the differences between the register and memory name-spaces. In addition to the storage for architected memory state found in normal caches, the L2 cache banks include buffers for holding checkpoint data, live-in values, and live-out values. Each type of data is stored in a separate buffer because the data needs to be accessed in different ways. The checkpoint data is merged with architected data by hardware that performs *checkpoint assembly*, the process of constructing the view of memory expected by a slave task (Section 4.5.3). The verifier hardware gathers the architected value for each buffered live-in memory value and ensures that the necessary coherence permissions are available to perform the verify/commit atomically (Section 4.5.5). The GRF and the L2 cache banks, which are physically distributed across the chip, use a two-phase commit algorithm to simultaneously agree that the task is verified.

4.4.2 High-level Operation:

The steady state operation of the implementation is shown in Figure 4.4. As the master executes a task of the distilled program, it notes which registers were written by retired instructions and collects retired stores into message buffers. As buffers fill up they are sent to the appropriate L2 cache bank. When the master processor retires a `FORK` instruction, the contents of the remaining message buffers are sent. In parallel, the final values of the written registers are read out of the register file and shipped to the GRF, along with the program counter (PC) specified by the `FORK` instruction. This check-pointing generally takes multiple cycles, during which the master continues executing subsequent tasks, and all of the data is tagged with the current checkpoint timestamp. When it can be assumed that all of the data has been received, it notifies the centralized task logic (located at the GRF) that the checkpoint is available and the task should be allocated to a processor when one is available.

The task logic then selects an idle processor and sends it the timestamp of the task to be allocated. In addition, it sends a register checkpoint (including a PC) constructed by overlaying all of the partial register file checkpoints over the architectural checkpoint register file. When the slave requires a cache block not currently in the L2—recall that all stale blocks have been invalidated—it sends a request to the L2 cache that includes its timestamp. The data returned is assembled from the architectural memory state and the check-pointed memory state. For each byte, the most recent value that is not newer than the requesting timestamp is selected. If the returned cache block includes any checkpoint data, this information is indicated in the message from the L2 cache bank, and the stale bit is set when the data is loaded into the cache.

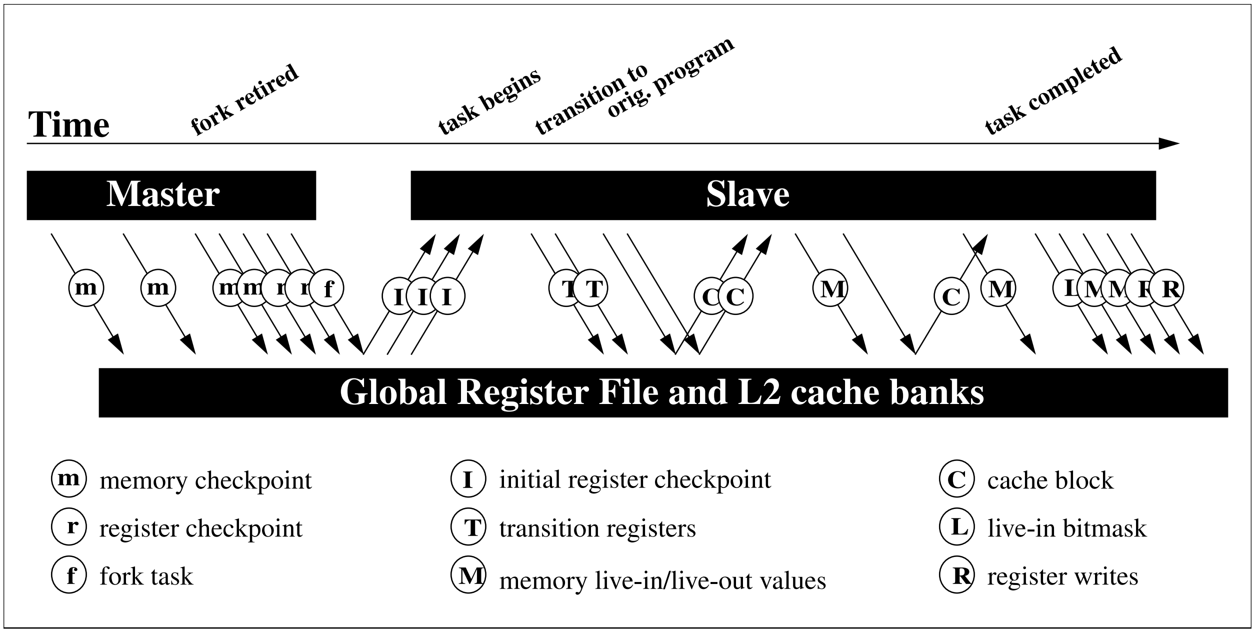


Figure 4.4: Life of a single task (steady state operation of MSSP). *The distilled version of a task is first executed by the master processor; the master periodically sends messages when its message buffers fill, until a fork instruction is retired, at which point it sends any partially complete messages before sending a fork message. The task is allocated to a slave, which first receives its initial register file then begins executing the transition code. Upon retiring the* `VERIFY` *instruction, it sends the written transition registers. When the task is complete, the slave sends a live-in register mask and any remaining partially complete live-in/live-out memory messages.*

The slave executes two pieces of code: the transition code and the task from the original program. The slave first executes the transition code that rearranges the state data provided by the master program into the locations that the slave task (*i.e.*, the original program) expects it. Because this code is part of the distilled program (*i.e.*, not part of the task from the original program) the processor need not track live-in values, but it does track writes. Register writes are sent to the GRF, tagged as transition register writes. Transition stores are treated as checkpoint stores. The program distiller marks the end of the transition code by making its last instruction a `VERIFY` instruction. The `VERIFY` also encodes the starting PC of the task in the original program; this PC is also sent as a transition register to the GRF.

The retirement of the `VERIFY` instruction marks the true beginning of the task, where the slave processor must begin tracking live-in and live-out values. Task live-outs are tracked the same way that checkpoint writes are tracked, by recording stores into message buffers and noting register writes. Tracking live-ins is similar to tracking live-outs, except that values read are recorded, but only those that have not yet been written. So, on the register side, I only set bits in the read register bitmask if the register has not yet been written by this task (as indicated by the written register bitmask). For loads, it means I need to keep track of the task's memory writes at the processor. This tracking can be done with a structure that tracks store addresses and which bytes were written. This structure is searched to determine if part or all of a load was satisfied by a store from within the task.

Generally, tasks are allowed to complete before they are fully verified. When a slave processor reaches an instruction that is tagged as being the beginning of another task, it stops fetching additional instructions. When this last instruction retires, the live-in register bitmask is sent to the GRF and the message buffers are purged to their respective L2 cache banks. The registers written by the task are read out of the retirement register file into a register message buffer to the GRF. When all data has been sent, stale blocks are invalidated and the processor is free to be allocated to another task.

When all previous tasks have completed, verification of a task can begin in earnest. Verification requires that all L2 cache banks and the GRF to simultaneously agree that all live-in values are valid and coherence permission to write the live-outs has been attained. For each of the task's memory live-in values, the L2 cache bank ensures that it has read permission to the cache line (requesting the block if it is not present) and reads the corresponding architectural value into the live-in buffer. The live-in buffer hardware compares the value against the recorded live-in value, signalling a misspeculation if the value does not match. Issues derived from maintaining these buffer entries coherent and overlapping the verification of multiple tasks are discussed in Section 4.5.5.

Once an L2 cache bank has successfully verified all of its live-in values and acquired write permission to all of the cache blocks with live-out values, it can signal that it is ready to commit the task. These signals are used in a two-phase commit algorithm (described in Section 4.5.5). If, for whatever reason, it is difficult to acquire the necessary blocks simultaneously, it is correct to signal a misspeculation and execute the task non-speculatively (as a traditional processor would). Policies for maximizing performance in the presence of multiprocessor coherence are beyond the scope of this dissertation.

When a task misspeculation is detected, the master processor and all slave processors executing later tasks are reset (including an invalidation of all stale cache lines). Furthermore, all speculatively buffered (un-committed) data for later tasks is cleared. To restart the execution, first, the actual register file is copied to the checkpoint register file. Then, a processor is selected to be the master and is sent the checkpoint register file. The master processor maps the architected PC to the PC of an entry into the distilled program and begins execution. Then a processor is selected for the non-speculative execution of the task, and the GRF sends that processor the architected register file.

Task sizes vary and generally some tasks are (dynamically) longer than can be buffered in the limited resources of the L2 cache banks. In these cases, I perform what is called an *early* verification. In effect, the slave processor pretends that it has finished the task. Logically, at an arbitrary point in the task, the slave flushes all in-flight instructions and sends all live-out values for verification. In parallel with verification, it purges all stale blocks from its cache and requests the GRF to send it the new architected register file. Once this new register file is received and the task has been verified, the slave processor can now execute in non-speculative mode. In Section 4.5.10, I describe how this early verification can be done without stopping the execution of the task.

4.4.3 Program Data That Validates This Approach

Although this implementation is justified somewhat by the performance results in the following chapter, I find that more intuition can be gained from doing back-of-the-envelope calculations. In this section, I present estimates of storage and bandwidth requirements for this implementation to give some intuition about its feasibility. I have collected some empirical data about the register and memory usage of various size fragments of program's execution (shown in Figure 4.5). Approximate values are shown in Table 4.1 for tasks of 100, 200, and 800 instructions, for which I calculate storage and bandwidth requirements. It should be noted that these are not tasks selected by any task selection algorithms, but merely segments broken at fixed length intervals, so these estimates are intended to taken with a grain of salt.

I assume the cores to be comparable to modern processors and would achieve an average IPC of 1.5 on these workloads if executed in normal uniprocessor mode (my baseline 4-wide machine ranges from 0.7 to 2.0 IPC for SpecInt2000). I would like the implementation to be able to support distilled programs that execute three times faster than the original program (*i.e.*, an equivalent IPC of 4.5). Thus the master completes 100, 200, and 800 instruction tasks in 22, 44, and 178 cycles, respectively. It is expected that execution of the tasks by the slave processors will take longer than they would if they were part of a traditional uniprocessor execution (due to inter-processor communication and incomplete branch history). In spite of this fact, in the bandwidth calculations that follow, I assume no elongation to do a worst case analysis. Thus, the slave completes the 100, 200, and 800 instruction tasks in 66, 133, and 533 cycles.

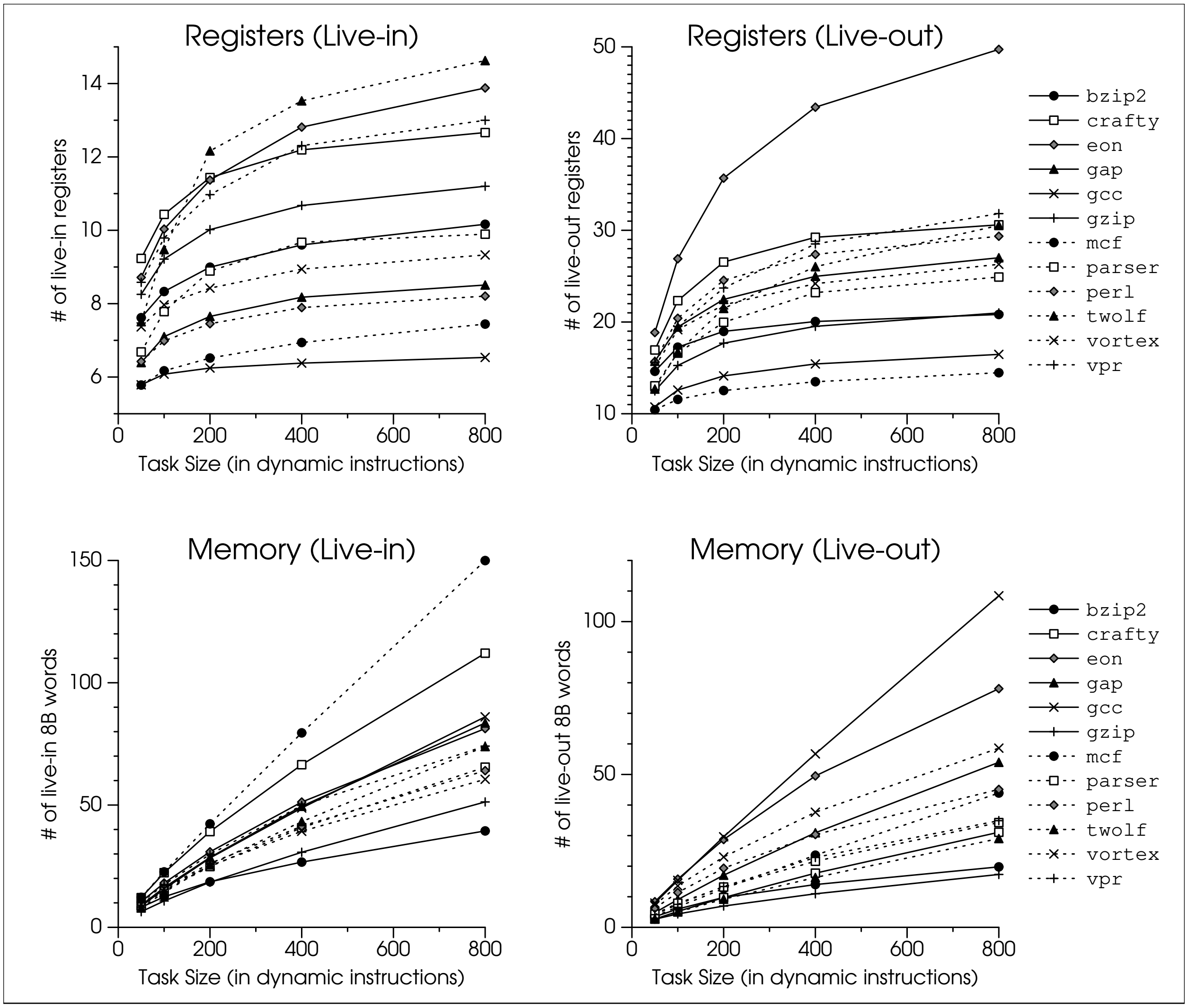


Figure 4.5: Characterization of task live-in and live-out set sizes. *Register and memory live-in and live-out set sizes are shown for tasks of 50, 100, 200, 400 and 800 dynamic instructions. Register set sizes are almost constant, while memory set sizes are slightly sub-linear with task size.*

TABLE 4.1 Raw data on average number of 64-bit registers, 64B cache lines for refreshes, and 8B memory words moved for tasks of different sizes, estimated from Figure 4.5.

Task Size	Registers				Refreshes (blocks)		Memory Words		
	M-out	S-in	Trans.	S-out	M-in	S-in	M-out	S-LI	S-LO
100	32	63	14	32	3	7	10	15	10
200	32	63	14	32	5	15	20	30	20
800	32	63	14	32	13	40	50	80	50

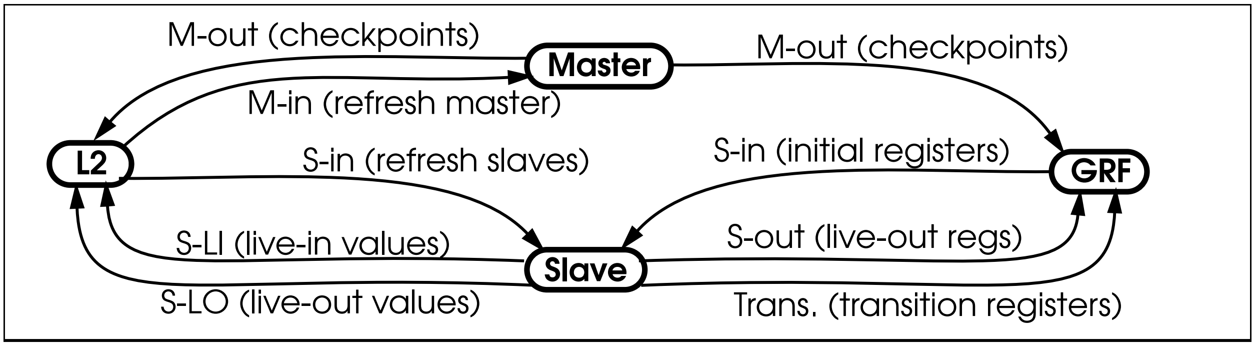


Figure 4.6: Graphical representation of flow of register and memory data. *For use as a key for data in Table 4.1*

Bandwidths. I walk through the calculation of the 200 instruction tasks in detail, but initial and final data for 100, 200, and 800 instruction tasks is provided in Table 4.2. I consider the bandwidth required at 4 different endpoints: the master, the slave, the GRF, and the L2 cache bank.

Master. In steady state, the master has to communicate any register changes since the last forked task; I assume that the master writes the same number of registers as the original program (32 for 200 instruction task), which I believe to be conservative. In addition, any stores performed by the task need to be flushed down to the L2 cache banks; again, I conservatively assume that the master does no more than the original program (~20 memory words for 200 instruction tasks). Lastly, I periodically refresh the master's stale cache blocks to remove old checkpoint stores and get committed stores by the original program. I estimate that perhaps one-fourth as many blocks should be refreshed as memory words were stored to by the task; the factor of four accounts for the fact that multiple words of the same cache block might have been written and there is likely some overlap between the blocks written by sequential tasks (as is suggested by the memory live-out data in Figure 4.5). I assume 5 blocks are refreshed per 200 instruction task.

Master Bandwidth:

32 registers	(at 7 regs/block)	= 5 blocks
20 memory words	(at 4 words/block)	= 5 blocks
memory refresh		= 5 blocks
Total		**= 15 blocks**

Over a 44 cycle task, that rate is a little more than 1 block every 3 cycles.

Slave. Slaves are first sent an initial copy of the register file (62 registers plus PC, assuming no optimization to avoid sending unchanged registers, 9 blocks). Then the slave processor can begin refreshing the invalidated stale blocks in its cache; I assume that 15 blocks will be refreshed for 200 instruction tasks

TABLE 4.2 **Estimated bandwidths calculated from raw data in Table 4.1.**

Task Size	Data Transferred per Task				Cycles per Task		Bandwidth messages (bytes) per cycle			
	M	S	L2	R	M	S	M	S	L2	R
100	11	30	30 / 8	21	22	66	.50 (32)	.45 (29)	.17 (11)	.95 (61)
200	15	44	43 / 8	21	44	133	.34 (22)	.33 (21)	.12 (8)	.47 (30)
800	31	89	99 / 8	21	178	533	.17 (11)	.17 (11)	.07 (4)	.12 (8)

(estimated as half the number of memory live-ins). When the slave retires the `VERIFY` instruction, the written transition registers are sent to the GRF (slightly less than half of the integer register file (14 registers) can be re-mapped with 2 blocks). By the end of the task, the slave will have sent the final register updates (32 registers, or 5 blocks for 200 instruction tasks), memory live-ins (about 30 memory words for 200 instruction tasks, or 8 blocks), and the stores (about 20 memory words for 200 instruction tasks, or 5 blocks).

Slave Bandwidth:

63 initial registers	(at 7 regs/block)	= 9 blocks
memory refresh		= 15 blocks
14 transition registers	(at 7 regs/block)	= 2 blocks
32 written registers	(at 7 regs/block)	= 5 blocks
30 live-in memory words	(at 4 words/block)	= 8 blocks
20 written memory words	(at 4 words/block)	= 5 blocks
Total		**= 44 blocks**

Over a 133 cycle slave task, that rate is a little above 1 block every 3 cycles.

GRF/L2 Cache Bank. The GRF and L2 cache banks have to absorb and produce all of the register-related and memory-related (respectively) traffic required by the master and slave processors. So from the above:

GRF Bandwidth:

32 checkpoint registers	(at 7 regs/block)	= 5 blocks
62 initial registers	(at 7 regs/block)	= 9 blocks
14 transition registers	(at 7 regs/block)	= 2 blocks
32 written registers	(at 7 regs/block)	= 5 blocks
Total:		**= 21 blocks/task**

Because the task throughput is one task every 44 cycles, this communication rate requires bandwidth of just less than one block every other cycle. This rate is perhaps an over-estimate because I don't expect the master to write as many registers as the slave, nor that so many transition registers will be needed all of the time (right now the distiller uses at most one for the stack pointer). Nevertheless, this unit requires the most bandwidth, suggesting the optimizations to avoid sending a full initial register file (described in Section 4.5.9) will be beneficial.

L2 Cache Bank Bandwidth:

20 memory words	(at 4 words/block)	= 5 blocks
memory refresh (Master)		= 5 blocks
memory refresh (Slave)		= 15 blocks
30 live-in memory words	(at 4 words/block)	= 8 blocks
20 written memory words	(at 4 words/block)	= 5 blocks
Total:		**= 43 blocks**

Because this traffic is spread across multiple banks, it is not a major bottle neck. Even if a third of the traffic was directed at one bank (a rather severe hot spot for a system with 8 banks), that would only be 13 blocks every 44 cycles, or less than one block every three cycles.

Storage. With the same data I estimate the amount of storage space that will be needed to hold the checkpoint, live-in, and speculative store data. My default implementation includes eight processors, so I consider storage for eight tasks. For 800 instruction tasks (80 double-word live-ins, 50 double-word live-out stores/checkpoint stores):

Live-ins:

80 live-ins/task * 8 tasks	= 640 live-ins
640 live-ins * 16B storage/live-in	= 10240B = 10KB storage

Checkpoints/Live-outs:

50 live-ins/task * 8 tasks	= 400 live-ins
400 live-ins * 16B storage/live-in	= 6400B = 6.25KB storage each

10KB + 2 * 6.25KB = 23KB storage

For a 2MB L2, which is available on existing processors and will likely be small for future processors, this amount of storage is only about 1% of the L2 storage resources.

Although it is one of our goals to minimize the amount of MSSP specific hardware, in this case, overloading existing storage arrays appears to be inefficient. For example, the speculative data takes such a small amount of storage relative to the size of the L2, I do not think it makes sense to use a unified structure, as is demonstrated by a quick thought experiment. If L2 cache blocks stored non-architectural data, at minimum the current array would have to be augmented to store the timestamp, at least an additional 4 bits of state. Assuming that these bits were added to every block uniformly, a cache with 64B blocks (512b data + 40b tag = 552b) would have 0.7% overhead (not counting the higher utilization of the cache) a significant fraction of the whole amount of storage necessary for all of the speculative data. Furthermore, designing specialized arrays for this data allows it to be searched and manipulated more efficiently. I discuss the design of these structures and other implementation details in Section 4.5, after a brief interlude about power consumption.

4.4.4 Power Consumption

Although power consumption is a serious implementation constraint, it is not one of the main focuses of this dissertation. As a result, the proposed implementation is not particularly power efficient. In this section, I intend to demonstrate that this inefficiency is a characteristic of the proposed implementation and not a characteristic of the execution paradigm and that power efficient implementations are likely.

The two major features of the MSSP paradigm that would likely be the root of power inefficiency are: (1) the use of multiple processors to execute a sequential program and (2) the increased frequency of communication required between processors and the L2 cache. I will discuss each of these issues in turn.

Using N processors need not use N times as much power, especially if the processors do not have to be equivalent. Because the architecture is tolerant of the slave processor's execution rate— performance is largely determined by the execution rate of the master processor—the slave processors do not need the microarchitectural complexity used to maximize the performance of modern processors or the power burden that goes along with it. The slave processors can be designed to be simpler, narrower, and with shorter pipelines, and they can be clocked at lower frequencies to enable lower voltages. Furthermore, the unique role of the master processor as a value predictor leads to other power optimizations. Similar to what was proposed with the DIVA checker [4], the master processor can potentially be executed at a voltage with minimal noise margins, if it can be engineered so that all noise-induced faults will manifest as bad checkpoint writes that will be detected during task verification.

Although I find reasoning about the impact of the additional communication bandwidth to be more difficult, I am reassured by the fact that this communication is rather latency tolerant. If the dynamic power consumption of the interconnection network can be modelled with the traditional logic power equation $P = aCV^2F$ (where P is power, a is an activity factor, C is capacitance, V is voltage, and F is frequency) and voltage scaling can be applied, then increases in latency yield a cubed reduction in power consumption. Realistically, the power benefits will not be quite this high as some measure must be taken to maintain the required bandwidth, either by increasing the width of the interconnect or its degree of pipelining.

Finally, consider that the MSSP paradigm can only be considered power inefficient where there is an alternative means of achieving the same level of performance that uses less power. The traditional architectural approaches to improving sequential program performance (*e.g.*, deeper or wider pipelines, more and larger predictors) have their own power costs and, as the distance signals can travel in a single cycle decreases, monolithic processor designs will become even less efficient requiring more information to be replicated and cached.

4.5 Mechanism Details

In this section, I describe a number of MSSP specific mechanisms in detail. These descriptions are not fundamental to the MSSP paradigm, but are meant to serve two purposes. First, they explain concretely one way that the required functionality could be implemented. Second, they document the implementation that was used to collect the results presented in Chapter 5.

4.5.1 Live-in/Live-out Collection

Master and slave processors both must track the registers and memory locations they write, and the same mechanisms are used for both processor roles, although the different characteristics of registers and memory led me to develop separate mechanisms for tracking register and memory writes. Memory live-outs are captured at retire time, by recording a store's address, size, and value. These values are held in a structure organized like a store buffer until they are sent to the appropriate L2 cache bank. Since registers are frequently over-written, no register values are sent to the GRF until the end of the task. Thus, when an instruction that writes a register is retired, the hardware only needs to record the architectural register identifier. The set of written registers is tracked using a bitmask that has one bit for each architected register. When the task ends, the final values of these registers are read[3] from the physical register file and sent to the GRF.

In addition, slave processors need to track their live-in values. Tracking live-in values is similar to tracking live-out values, except that values read are recorded, but only those that have not yet been written. So, for register reads, I filter updates to a read register bitmask with the written register bitmask. For loads, it means I need to keep track of the task's memory writes at the processor. This tracking can be done with a structure that tracks store addresses and which bytes were written. This structure is searched to determine if part or all of a load was satisfied by a store from within the task. Figure 4.7 shows the structures for tracking live-in and live-out values and their storage requirements.

3. In the case of the master, it is important to be able to read out these registers without stalling execution; reading these values can be accomplished by exploiting unused register read bandwidth or by creating a retirement register file specifically for this purpose.

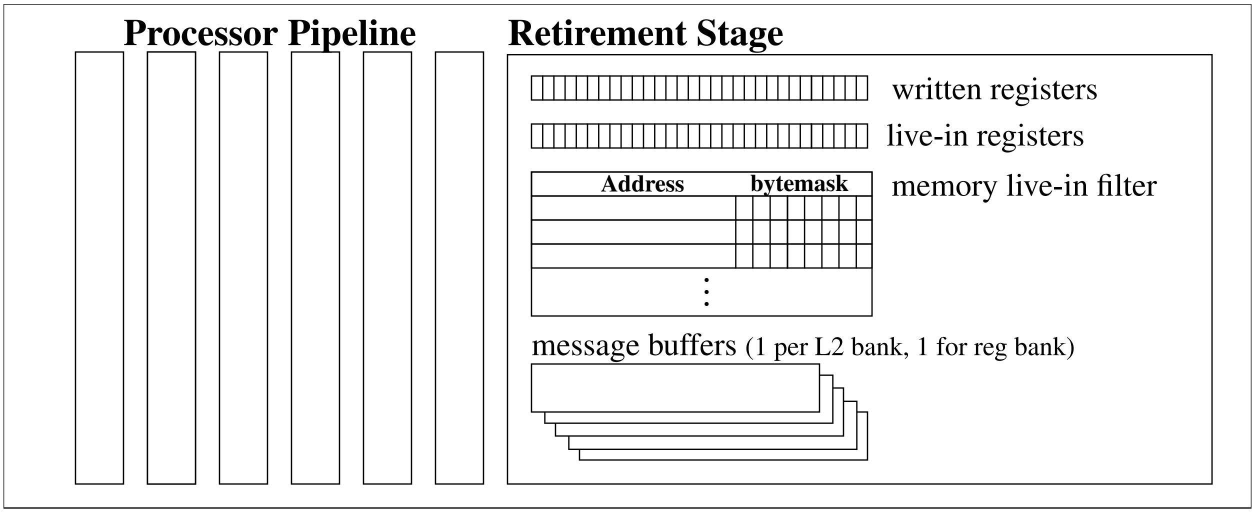


Figure 4.7: Hardware structures for collecting live-in and live-out values. *Register writes and reads are recorded in bitmasks; the final values of these registers are read at the end of the task. Memory values are recorded at retire time into message buffers, to be sent to the L2 bank. As a load or a store is recorded, the location of the recorded bytes are tracked in the memory live-in filter, which is used to filter future memory live-ins.*

Since the live-in and live-out values are stored at the L2 cache banks and the GRF, they must be sent over the interconnection network using special message types (described in the next sub-section). Processors periodically send memory live-in and live-out values as they accumulate. Register live-outs are packed into messages as they are read out of the register file. Only the bitmask indicating which registers were live-ins need to be sent; register live-in values are available in the architected checkpoint register file (part of the GRF, described in Section 4.5.4). Live-in memory values are captured at the processor when first used; the possibility that architected memory values can be modified during the execution of a task—via task commitment or coherent writes—makes other value tracking schemes challenging to implement.

Two subtleties of the implementation deserve mention: 1) the opportunity for different live-in values for the same memory location, and 2) the window of vulnerability between capture of live-out values and when they are received by the L2 cache banks. Whenever cache lines are displaced from a slave processor's cache, the contents of the cache line could be different if it is brought back in, because the architected state may have been updated in the mean time. For this reason, two loads to the same address from a given task could receive different values without an intervening store in the task. Since tasks are retired atomically one of the values must be incorrect. I delegate the detection of this situation to the verifier hardware. When a cache block is replaced in a slave processor's cache, any corresponding entries in the live-in filter are cleared, ensuring that the slave will send all copies of the value observed to the verifier.

Because non-architected state is created by the processors, but buffered and merged with architected data remotely at the L2 cache banks, there is a window of vulnerability when cache data could potentially get lost. It occurs when a data block is in transit to a processor's L1 data cache when the processor retires a store to the block. This vulnerability is largely a result of the combination of my selections of write-through and no-write-allocate policies. It is both a correctness issue for non-speculative tasks and a serious performance issue for the master. Because this window results from data in transmission being unobservable, the implementation keeps a copy of all non-architectural data retired by the processor until it has been acknowledged by the L2 cache bank. This data is merged with cache lines received from the L2; the local data will always be more recent than the received data. This buffering also allows the L2 cache bank to send negative acknowledgements ("nacks") when no storage is available, without risking losing data.

4.5.2 Register/Memory Word Messaging

To efficiently communicate register file and memory image differences, the interconnection network interfaces support two additional types of messages: register messages and memory word messages. Figure 4.8 shows a pair of packet formats assuming 64B messages: register messages can hold 7 registers (64b value and 8b identifier) and memory messages hold 4 memory words (48b physical address, 64b value, 8 valid bits (one per byte), and 8 bits of flags). Live-in and live-out values can be sent in the same message by flagging each with its type. For example, transition registers and live-out registers could be packed in the same message using four bits to specify that the first N values are transition registers. In addition, all messages are tagged with the processor's current timestamp.

4.5.3 Memory Checkpoint Assembly

Memory checkpoints—the view of memory required by a particular task—are constructed by the L2 cache banks by layering the difference data over the architected data in increasing timestamp order, as described in Section 2.2.1. As mentioned in Section 4.4.3, this implementation stores the checkpoint data in a special structure and not in the L2 cache arrays. Each entry of the checkpoint buffer holds an aligned 64-bit value, eight valid bits (one per byte), a timestamp, and an address. Storing the checkpoint data in this way allows it to be deallocated trivially (with a flash invalidate of entries with matching timestamps) when the corresponding task commits.

My implementation requests checkpoint data at the granularity of cache blocks, so that the data can be cached in the slave processor's L1 data cache. To construct a cache block's worth of the memory checkpoint, the differences that match the cache block have to be merged with the architected data. All entries that match the cache block, which is larger than the granularity at which checkpoints are stores, must be considered. As there may be contributions from multiple timestamps, potentially many entries in the checkpoint buffer may need to be read. In practice, 85% of the time there are two or less matching entries. As a result, the hardware can be designed to read them out serially and merge them with the architectural data as shown in Figure 4.9. First, the tags are probed to find all matching entries. Second, the matching entries are read out one by one and merged in the *merging block*. Each byte is merged independently, and a byte is only kept if it has the most recent timestamp in that location. Third, the architectural data read from the L2 cache array is merged; architectural data is only kept if no checkpoint value was found for that byte.

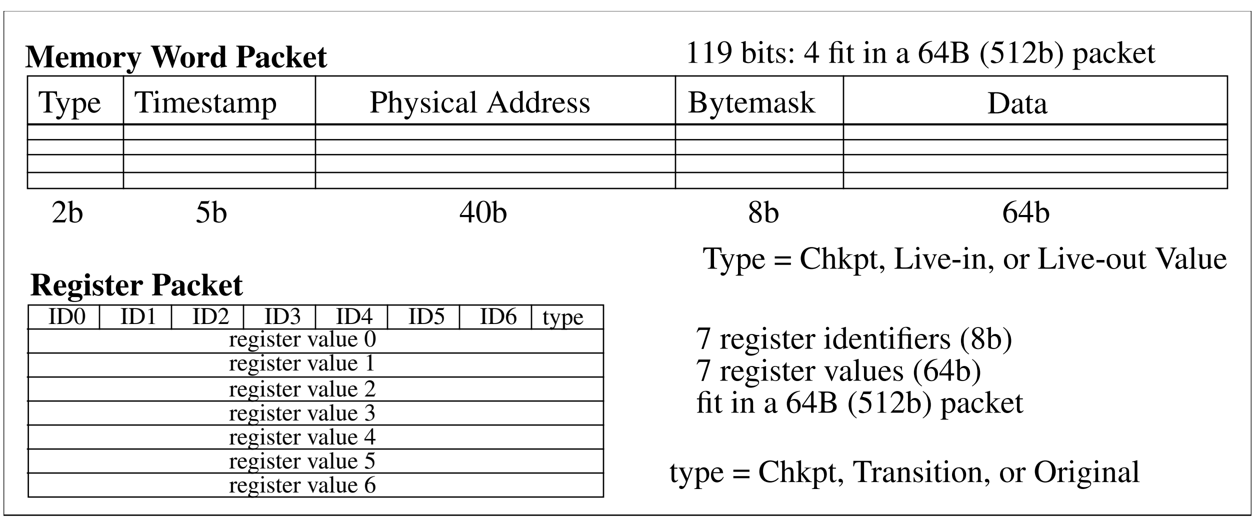


Figure 4.8: Packet formats for sending memory words and register values through interconnection networks. *Memory packets send 4 64b double words with address and bytemasks indicating which bytes contain data. Register packets send 7 64b register values and their identifiers.*

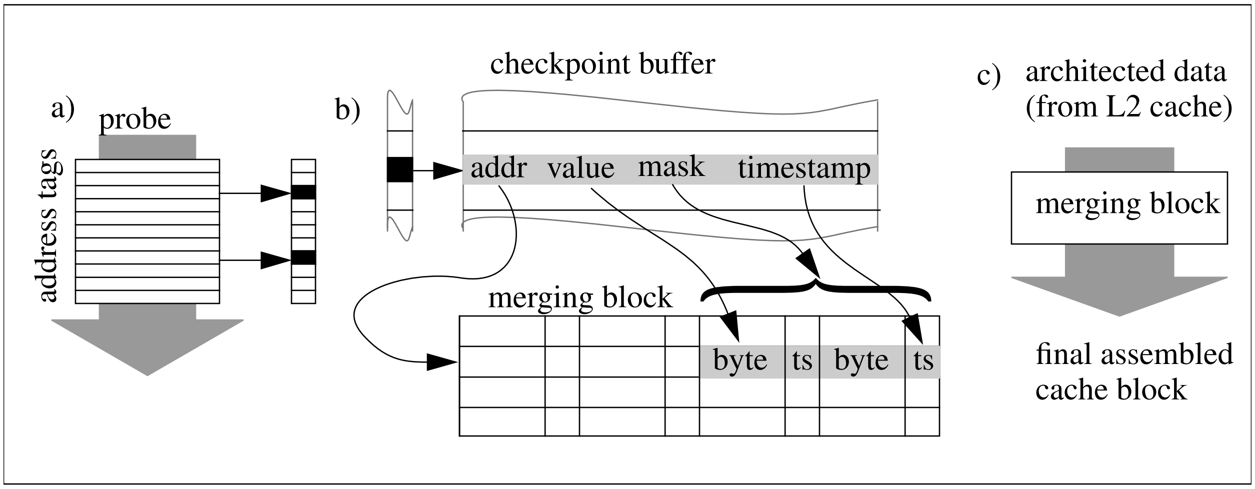


Figure 4.9: Checkpoint assembly from entries of the checkpoint buffer. *(a) Matching entries of the checkpoint buffer are identified, (b) the matching entries are read out serially and merged in a merging structure that keeps valid bits and timestamps on a per byte basis, and (c) the checkpoint data is merged with the architectural data read out of the L2 cache array (checkpoint data always takes precedence over architectural data).*

When few checkpoint entries match, they can be read out in parallel with the reading of the architectural data. In these cases, the L2 hit time is only increased by the time to merge the architectural and checkpoint data.

4.5.4 Global Register File

The global register file holds the architected register values and the checkpoint, transition, and task live-out registers. Register file checkpoints are simpler to manage because—at least in the Alpha architecture—there are no partial register writes so no merging is required and the architectural register name space is small. These characteristics lead to a different organization for the global register file (GRF) than for the L2 cache banks.

There are three independent name spaces tracked by the GRF: visible state, checkpoint state, and transition state. Visible state is the register state associated with the original program's execution, and the committed version of visible state is the architected register state. The checkpoint state is that produced by the master processor's execution. The transition state is produced by the transition code executed by the slave processors. As described in Section 3.3, the transition code is necessary because state, especially registers, may be mapped differently in the distilled program than the original program. Thus, the transition registers hold the state of the distilled program after it has been mapped back to the original program's naming scheme.

Functionally, the GRF is like an ARB [26] for registers: when queried, the GRF returns the newest (in program order) value that is older than the requesting task. The organization of the GRF (shown in Figure 4.10), which differs from the ARB for efficiency reasons, is similar to the organization of the register file in some out-of-order processors, like the HP-8000 family [41]. For each namespace, the GRF has a committed register file, a set of register maps, and a set of physical registers. For each architectural register, the GRF has one map entry for each active task. If the task wrote a register, the map entry points to the physical register that holds the value, otherwise the map entry is invalid. When a register for an uncommitted task is requested, the register's map entries are read (in parallel) and the most recent valid entry is selected. If no uncommitted value older than the requesting task exists, the value is read from the committed register file.

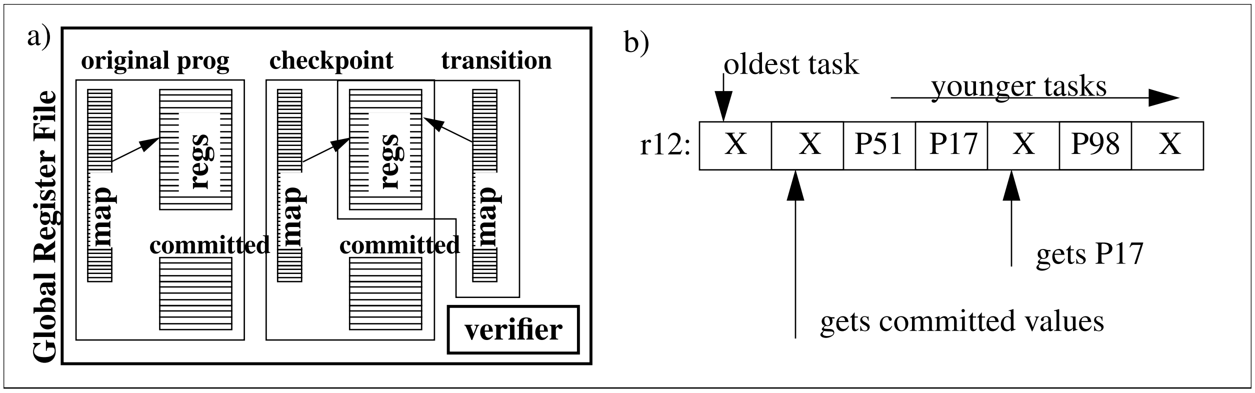


Figure 4.10: Detail of global register file implementation. *a) The GRF tracks three independent namespaces (visible state, checkpoint state, and transition state). Checkpoint and transition state share a physical register file. b) Each register has a map table entry for each in-flight task, which can be invalid (indicated by an 'X') or contain a pointer to a physical register. The GRF supplies the youngest register write from a task older than the requesting task.*

Because the complete register images are maintained in the GRF, the slave tasks can communicate the set of live-in registers read by sending a bitmask, with one bit per architectural register. Register verification is performed by comparing the transition registers to the original program registers for a given task. Only the registers indicated by the live-in register bitmask must match for the task to be validated. Because transition registers are the mapping of a given task's state, only transition registers associated with the requesting task are considered. If no matching transition register was written for the requesting task, the register must have been mapped similarly in both the distilled and original program and the request proceeds as a request for a checkpoint register for the same task. When a task is committed, its register values are copied to the committed register file.

4.5.5 Verification/Commitment

As previously mentioned, verification and commitment is like performing instruction reuse at the granularity of a task and must appear atomic to ensure no consistency model violations. MSSP provides the appearance of atomicity by acquiring the necessary coherence permissions (read for lines with live-ins, write for lines with live-outs) and delaying coherence requests while live-outs are committed. Though logically centralized, the verification and commitment process is physically distributed across the GRF and L2 cache banks requiring a two-phase commit protocol. Also, to avoid putting the inter-processor communication latency on the critical path, it is important to be able to verify tasks in parallel; initially, though, I will describe a sequential commit process.

The GRF and each L2 cache bank independently verify their portion of live-ins. Register verification is straightforward; the original program register file is compared to the transition register file, and if all registers indicated by the live-in mask match, the GRF raises a verify signal. Memory verification is more complicated, in part due to coherence. When a memory live-in value is received by the L2 cache bank, read access to the corresponding block is acquired and the architected value is stored with the live-in value in a live-in buffer. If the block in the cache is invalidated (either due to a cast-out or coherence request), so is the architected value. When all live-ins for the oldest timestamp have been received and match their architected value, the L2 raises its read verify signal. When the L2 receives live-outs, it tries to acquire write permission to the necessary cache blocks and tracks this permission with a bit in the live-out buffer entries, which are kept coherent with the cache. A write verify signal, raised when all live-out values with the oldest timestamp have permission, is ANDed with the read verify signal to form the bank's verify signal.

To verify a task, all of the verification signals have to be set simultaneously, which is hindered by these signals coming from distant parts of the chip and the fact that they can transition from set to clear if a coherence transaction steals away a necessary block. In our two-phase commit protocol, these verify signals are routed to a central "arbiter" on the chip. This arbiter ANDs these signals together and broadcasts the result back to the register and L2 cache banks. When a bank sees this resulting signal go high, it stops processing coherence requests to avoid having to lower its own signal. If the signal remains high for more than the round trip latency of the longest path, then all parties know that verification has been successful. Committing a task involves copying live-out data with the oldest timestamp into architected state, both at the L2 cache banks and in the global register file. In addition, buffered live-in values and memory checkpoint values can be invalidated.

To avoid sequential communication latency in the verification process, verification of multiple tasks can be overlapped. Overlapping the verification requires that the live-in and live-out buffers search each other when new entries are written (much like load/store queues) to keep the live-in value's architected value up-to-date. In addition, each byte of the architected value needs to be tagged with the timestamp (if any) of its current source, because live-outs can come out of order and write a subset of a live-in value. Lastly, I need the ability to source data from the live-out buffer while it is lazily copied to the architected state. A task can begin verification as soon as all of its live-ins have been received and all of the preceding tasks' live-outs have been received. Tasks must commit in order, but, if data can be sourced from the live-out buffer, commitment can be as simple as setting a *committed* bit for the task's entries in the live-out buffer; movement of committed data to the L2 cache array can be done in the background.

If a live-in does not match architected state, a misspeculation is signalled. This signal is sent to the central arbiter and forwarded to all processors and all banks. Also, if, for some reason, all of the necessary coherence permissions cannot be acquired simultaneously, the L2 cache bank signals a misspeculation; policies for deciding *when* to signal a misspeculation are beyond the scope of this dissertation. After recovery, the task will execute non-speculatively, enabling forward progress, using the existing mechanisms to ensure forward progress in multiprocessors.

The peak bandwidth at which live-in and live-out data can be received by an L2 cache bank may outstrip the rate at which it can be written into the buffer structures. If this outstripping can occur, some buffering may be necessary to smooth out bursts of traffic. If these buffers are full and another message is received, it can be dropped and a re-send requested, because copies of the messages are kept at the processor until acknowledged.

4.5.6 Misspeculation Detection/Recovery Path

As mentioned in the previous section, misspeculations are detected in the L2 cache banks or the GRF and signalled to the other storage entities through the central arbiter. Because the live-outs of all previous tasks must have been received for the misspeculation to have been detected, there is usually only a small number of cycles between when a misspeculation is detected and when its task becomes the oldest task. As a result, for simplicity, this MSSP implementation waits for all previous tasks to commit before beginning recovery.

Failed verification is not the only means of instigating a recovery. For MSSP to be resilient to any faults in distilled program construction, a watch dog timer is necessary to handle cases that would otherwise cause deadlock. One such case occurs when the master enters a state where it ceases to fork new tasks (*e.g.*, it enters an infinite loop that includes no fork instructions); this situation is remedied by signalling a recovery when no slave tasks are in flight. Another case occurs when a slave task is forked, but it fails to transition to the original program; this case is handled by restarting if the head task fails to transition for a given amount of time. The case where the master provides a speculative slave task with a set of input parameters that prevent it from ever reaching a task boundary (*e.g.*, an infinite loop) is detected by the early verification mechanism (described in Section 4.5.10).

A recovery entails squashing all processor executions and invalidating all non-architected state (including any stale data in processor caches). Then, within the GRF, the architected register file is copied to the committed checkpoint register file. For the checkpoint, the PC is mapped to the PC of an entry (see Section 4.5.7 below). Processors are selected to be the new master—typically the same one as the old master for cache affinity—and a new slave. Each is sent a copy of its register file and directed to begin execution. The master processor executes the entry transition code before executing the next task in the distilled program proper. The slave processor executes its task non-speculatively since all of its live-in values will come from architected state.

Failed verification can also be used to provide a simple mechanism for handling traps and synchronous interrupts. If a slave task encounters a trap or synchronous interrupt, it can signal to the L2/GRF that it should fail verification. After recovery, the task will be executed non-speculatively. The fault should occur again when the faulting instruction is re-executed, and can be handled using the conventional uniprocessor mechanisms. Asynchronous interrupts can be handled similarly: when the interrupt is received by the CMP, all speculative tasks are forced to fail verification. After recovery or if the head task was already non-speculative, the interrupt is delivered to using conventional techniques.

4.5.7 Mechanisms for Mapping Between Programs

As described in Section 3.3.1, the distinctness of the distilled and original program necessitate mapping PCs between programs under certain circumstances. The implementation uses different techniques depending on the direction of the mapping. When mapping from the distilled program to the original program, the statically mapped PC can be embedded in the distilled program text. When mapping the other way, table lookup is employed.

Distilled program PCs to be mapped, be they link PCs in call instructions or task start PCs in `VERIFY` instructions, are fixed for a given instruction in the distilled program. As a result, they can be statically translated during the construction of the distilled program. To avoid any unnecessary hardware tables, the mapped PCs are encoded directly into the distilled program. To encode the full PC instruction slots after the mapping instruction can be used to provide a large enough immediate. These inlined immediates require a change in the way return addresses are computed for insertion into the return address stack.

Mapping from the original program to the distilled program is performed by table lookup so the implementation can leave the original program text unmodified. The two cases that need to be supported are 1) indirect branches and 2) entries into the distilled program.

Mapping indirect branch targets is one of the most intrusive changes to the processor core, but one that is latency tolerant. Since it performs a similar duty, the hardware is organized much like the hardware used to accelerate virtual memory. Most mapping requests can be satisfied by a small (*e.g.*, 64 entry) mapping lookaside buffer (MLB) that holds pairs of PCs <original program, distilled program>. As is shown in Section 5.2.12, most programs are insensitive to the addition of a few cycle delay to perform this mapping.

When an MLB lookup results in a miss, a page table structure in memory is accessed. As my results show that performance is insensitive to the time to access this page table, my implementation tries to minimize the size of this page table. A sparse page table is used because only about one-half to two percent of instructions are active indirect branch targets and they are not evenly distributed. My implementation, shown in Figure 4.11, uses two tables created by the program distiller: the L1 table encodes the range of L2 entries associated with a page of original program text, and the L2 table encodes the map entries themselves. The benchmarks have an average of 2 to 10 entries per thousand static instructions (16 4-byte entries fit on a 64-byte cache line). By ordering the entries sequentially, the mechanism could often guess

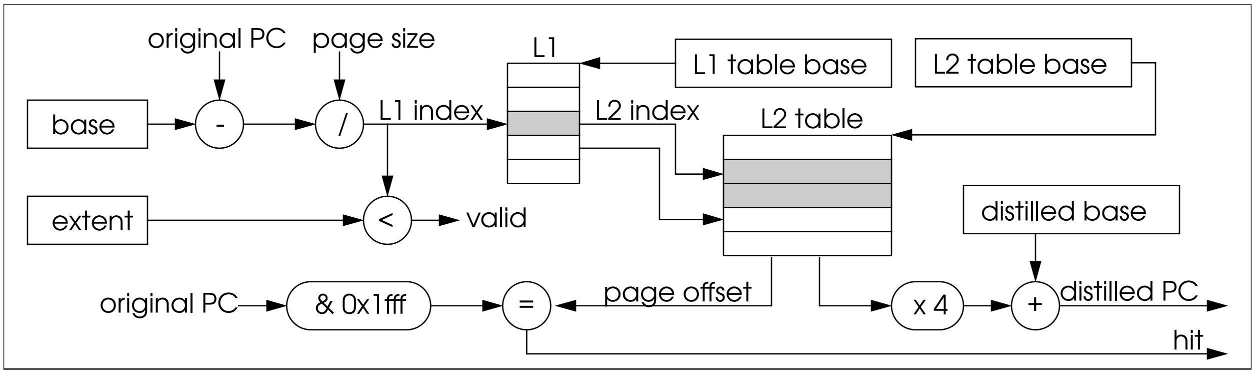


Figure 4.11: ***A sparse page table implementation for mapping.*** *The L1 table has one entry per (8kB) page of original program text. These entries hold a (16-bit) pointer to the first L2 entry associated with the page. Thus, by looking at two consecutive L1 entries, a state machine can identify the range in the L2 table to search. Each L2 entry concatenates the (11-bit) original program page offset and a (21-bit) offset from the beginning of the distilled program. The index in the first table is found by subtracting off a base pointer and dividing by the page size.*

which block holds the correct entry on the first try. Because of its latency tolerance the state machine could be implemented either at the processor—buffering the tables in private cache—or at the L2 cache banks—buffering the tables in the L2 cache arrays.

Entry PCs can be mapped with a similar mechanism, but the entry MLB is located at the GRF. As the set of entry PCs is smaller and exhibits better locality than the set of active indirect branch targets, the same hardware structure has a smaller miss rate (generally by a factor of two).

4.5.8 Tracking Stale Data and Refreshing

Unlike at the L2, (speculative) checkpoint data and speculatively written data is intermingled with architected data in the L1 data caches. Since the checkpoints are complete before the task is allocated to a slave processor—unlike other speculative multithreading paradigms—it is unnecessary to annotate data in the L1 with the timestamp of the task that created it[4]. Instead, a single *stale* bit is kept per block that indicates when the block is out-of-sync with current architected state. A block is marked stale if (1) it contains checkpoint data, (2) a speculative task stores to it, or (3) a write has been committed to this block since it was loaded by this processor. Even this single bit is just a performance optimization, which is rendered unnecessary if the data cache is flushed after every task. With the stale bits, only the tagged blocks need to be invalidated.

Setting the stale bit is trivial in the first two cases; the third is accomplished by extending an invalidation coherence protocol. When a cache block is fetched, the L2 cache bank knows if checkpoint data was used and communicates this information using a bit in the message. When a retired store by a speculative task is written into the cache, the block's stale bit is set. The most difficult case is when a store is committed at the L2 cache bank, but this case is very similar to need to invalidate a block in a traditional coherence protocol. Thus, as previous research has done [15, 32, 33, 73, 74], I extend the existing invalidation protocol to send "invalidate" messages when a store is committed. These messages are interpreted by MSSP processors as an indication to set their stale bits.

Although not necessary for correctness, it helps performance—by 20 percent on average, as shown in Section 5.2.11—if the master periodically "refreshes" stale blocks in its cache. A refresh is just the retrieval from the L2 of a block that the processor already has in its cache. The refresh brings the current view of the block (from the most recent timestamp) and enables the master to shed old checkpoint state and

4. This statement assumes that each processor is executing a single task at a time.

see architected state updates by slave processors. The frequency of refreshes has to be selected to balance conflicting desires; frequent refreshes ensure that the master has the most recent data available, while less frequent refreshing uses less bandwidth. Currently my simulations use a relatively naive mechanism: a simple queue that is filled on “invalidation” requests and drained by one entry periodically when there are few cache misses outstanding.

4.5.9 Efficiently Communicating, Reading and Writing Register Files

The MSSP paradigm requires us to communicate register values between processors, necessitating mechanisms to read values out of and write values into the processors register files. To avoid impacting cycle time, I try to avoid adding additional special ports to the register file, by harnessing unused bandwidth on the existing ports. Also, to minimize the interconnect bandwidth used to communicate register file contents between the GRF and the processor I keep a reference register file at the processor. I briefly detail these notions in this sub-section.

The processor core’s register file is written under three circumstances—beginning of a slave task, restart of the master, and an early verification—and read under three—checkpoint registers at the end of a master’s task, transition registers at the end of the transition code executed by the slave, and live-out registers at the end of the slave’s task. Of these tasks, reading of the checkpoint registers from the master processor is the only performance critical one.

Filling the processor’s register file can be done by using the existing write ports, by adding a path from the interconnection network to the register file. In two of the cases, task start and master restart, there is no competition for these resources, in the third, early verification, execution can be stalled. Reading the register file can be done similarly by using the existing read ports. Live-outs are generally read at the end of the task when there is no competition for the read ports. It would be preferable not to have to stall the processor to read transition registers, but it would be tolerable as slave execution is rarely on the critical path. On the contrary, master execution is frequently on the critical path.

Initializing the processor’s register file at the beginning of a task or after a restart of the master can be done using the existing write ports (by adding a path from the core’s interconnection network controller) because no instructions will be executing while the register file is being filled. Reading the register file is more common and needs to be done with little impact on the execution; these reads can either be accomplished by adding additional read ports, harvesting unused read bandwidth, or creating a separate retirement register file solely for the use by the MSSP logic.

4.5.10 Early Verification

If a task is particularly long, then some finite buffer may become full, which would prevent further execution. In such a situation, the MSSP implementation performs an early verification. An early verification can be implemented by squashing all in-flight instructions, temporarily stopping fetch, sending any remaining live-ins and live-outs to the appropriate banks, and requesting verification. If verification succeeds, the architected register file is sent to the slave and all of its stale data is invalidated. At this point the task can execute non-speculatively and can continue without the need for further buffering.

The processor need not halt execution if it can begin the early verification process while some buffering is still available. The challenge of performing verification concurrently with execution is that execution can produce new live-in values that need to be verified. To avoid this situation, the slave purges all checkpoint data from its cache and register file; only architectural data and speculatively written data will remain in the cache and register file. At this point, no new task misspeculations can be introduced: only existing misspeculations can be propagated, but these will be detected by verification of the existing set of live-in values. Once all checkpoint data has been purged, the slave no longer needs to capture new live-in values, just live-out values. When the task has been verified all live-outs can be immediately committed.

The early verification mechanism also gets overloaded to support task synchronization, where a speculative slave task waits until it becomes non-speculative before executing. Although not required frequently, support for synchronization enables the distiller to substitute a stall for cases that would almost certainly otherwise cause a task misspeculation. One such case is described at the end of Section 6.1. The required support is an additional special instruction that I can SYNC, which is inserted in the task's out-transition code. When the SYNC is encountered, an early verify is requested before the task begins; as no live-ins have been recorded, the verification should never fail. The result is that the task can begin executing non-speculatively, as soon as it becomes head and has retrieved the necessary architected state.

4.6 Chapter Summary

In designing an implementation of the MSSP paradigm, I focused on two key bottlenecks of the paradigm: (1) the master's execution of the distilled program, and (2) the verification and commitment of task data. To minimize overhead on the execution of the distilled program, I elected to perform all checkpoint construction and communication in hardware. To avoid communication latency on the verification and commitment of tasks, I perform this process centrally at the L2 cache banks. These design decisions enable an implementation that is remarkably tolerant of communication latency, because many of these latencies can be incurred in parallel, off the critical paths.

Chapter 5

Experimental Evaluation of the MSSP Paradigm

In this chapter, I present a simulation-based evaluation of the performance of the MSSP paradigm. I first describe the three components of my evaluation methodology (in Section 5.1) and then present performance results, supporting data, and sensitivity analysis (in Section 5.2). The major findings of this experimental analysis are summarized at the end of the chapter in the chapter summary.

5.1 Experimental Methodology

In this section, I describe the experimental infrastructure I used to evaluate the performance of the proposed paradigm. This infrastructure consists of three elements: 1) the prototype program distiller, 2) a timing simulator, and 3) the SPEC2000 integer benchmarks. These are the topics of the three sub-sections that follow.

5.1.1 Program Distiller Implementation

As described in Section 3.4, the preliminary evaluation in this dissertation is performed using a static off-line implementation of the program distiller. This implementation simplifies the construction of the distiller, because all of the profile information is available at the beginning of its execution.

All distilled programs are created automatically. The distiller is a binary-to-binary translator, which takes the original program and extensive profile information and generates the distilled program image and the necessary maps. This distiller prototype only implements a subset of the optimizations that I have identified as profitable. Currently, the distiller eliminates highly-biased branches, dead code, stack pointer updates, and branches whose taken and un-taken paths are the same. It also re-allocates registers, inlines functions, and performs code layout to minimize the frequency of taken branches. I expect results to improve as additional optimizations are implemented. Some parameters specified to the distiller are shown in Table 5.1.

5.1.2 Simulation Infrastructure

The performance results presented below were generated by a detailed cycle-level simulation model. The simulator I used was derived from the SimpleScalar toolkit [3], but the timing model has been constructed from scratch. Like SimpleScalar, only the user-level instructions are simulated; all system calls are emulated by calls to the machine running the simulation. .

TABLE 5.1 Baseline parameters supplied to the program distiller.

Parameter	Value	Explanation
Target Task Size	250 instructions	The task size the task selection algorithm considers optimal.
Branch Threshold	99% biased	The fraction of instances that must go to the dominant target for the branch to be removed.
Idempotent Opt. Threshold	99% correct	The fraction of times that an instructions output must match an input for it to be removed.
Silent Store Threshold	not applied	The fraction of times that a store must have overwritten an identical value in memory for it to be removed.
Long Store Definition	1000 instructions	The store to load distance at which a dependence is considered a long dependence.
Long Store Threshold	99% long	The fraction of store instances that have to be long for the static store to be considered long.

Unlike the traditional SimpleScalar timing simulators, functional simulation is *not* decoupled from timing simulation. Instead, the simulator tries to perform operations as the simulated machine would, when the simulated machine would. For example, registers are renamed to hold speculative state and misspeculation recovery is performed by rewinding to a pre-misspeculation register map. This truly execution-driven methodology is vital in simulating speculative parallelism architectures, because, like a traditional parallel processor, the timing of an operation (*e.g.*, a load) can alter its result.

Since the simulator is organized in this manner (*i.e.*, not decoupled), I have some confidence that the results presented are correct for the system as modelled. Concurrently with the timing simulation, I run a verification simulator, which has its own distinct copy of registers and memory, that verifies the functional correctness of instructions as they are retired. The second simulator ensures that the correct path through the program is taken and that the correct program values are computed. Because it is the timing model generating the functional results, it is difficult to accidentally construct an incorrect timing model that consistently generates correct functional results. Despite this validation of the timing model, conclusions drawn from these results are subject to the implicit assumptions made when constructing the timing model.

The simulator models a chip multiprocessor (CMP) that supports the Alpha AXP instruction set. The baseline model reflects the implementation described in Chapter 4, with 8 processors and a shared 2MB L2 cache with 8 banks, as shown in Figure 5.1.

Each processor core is configured to look similar to the HP Alpha 21264; details can be found in Table 5.2. The superscalar width, pipe depth, and functional unit latencies simulated are similar to those published for the 21264 [37]. In addition, the line predictor-based front-end is modelled—the machine fetches aligned groups of four instructions so the compiler-inserted nops for padding are important—but larger, more advanced branch predictors are used. Many engineering details of the 21264 (*e.g.*, functional unit clusters, troll traps, etc.) are not modelled, but it is my belief that the flexibility inherent in the distilled program would allow it to better avoid their negative impacts than traditional code can.

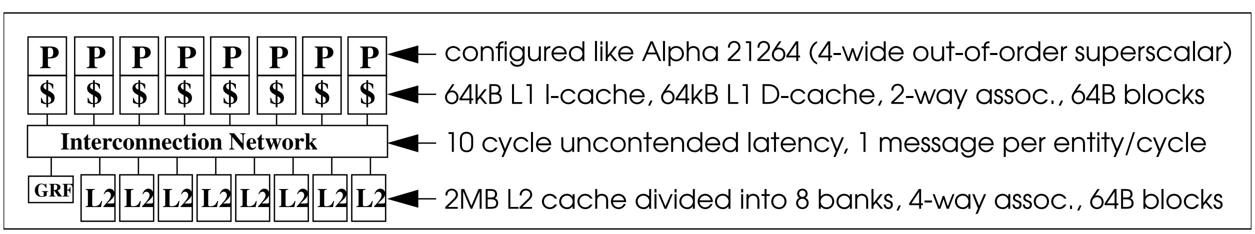


Figure 5.1: Diagram of machine model.

TABLE 5.2 Simulation parameters approximating a CMP of Alpha 21264 cores

Front End	A 64kB 2-way set associative instruction cache, a 64kb YAGS branch predictor, a 32kb cascading indirect branch predictor, and a 64-entry return address stack. The front end fetches aligned blocks of 4 instructions. A line-predictor à la the 21264 is implemented.
Execution Core	4-wide (fetch, decode, execute, retire) machine with a 128-entry instruction window and a 13 cycle pipeline, includes a full complement of simple integer units, 2 load/store ports, and 1 complex integer unit, all fully pipelined. The first-level data cache is a 2-way set-associative 64kB cache with 64B lines and a 3-cycle access latency, including address generation.
System	The system on a chip includes 8 processor and 8 L2 cache banks (each 4-way set-associative with 64B lines, 2MB total), and one GRF. An interconnection network is modelled that can move one 64B message per cycle to or from each processor and cache. One-way network latency is 10 cycles, unless stated otherwise. Minimum memory latency (after an L2 miss) is 100 cycles.

My experiments compare the performance of the MSSP paradigm to a traditional uniprocessor execution of the program on a single processor of the same CMP. Although clearly not an "equivalent resource" comparison, this comparison is valid. As we move into the billion transistor era, processor cores will be less resource constrained and more communication latency constrained [1, 48]. Thus, the resource inequality is less important than whether communication latencies are modelled sufficiently accurately. By default, the one-way communication latency across the interconnection network (*i.e.*, between processors, and between processors and L2 cache banks) is 10 cycles; in Section 5.2.6, I explore the sensitivity to this parameter.

5.1.3 Benchmark Programs

I performed this evaluation using the integer benchmarks from the SPEC2000 benchmark suite. These programs exhibit the control-flow and memory access behaviors representative of non-numeric programs. Generally, these behaviors make these programs harder to analyze with a compiler and more difficult to efficiently execute than numeric programs.

The benchmarks were compiled for the Alpha architecture using the Digital Unix V6.21 compiler with optimization levels and flags specified for producing PEAK executables. The executables were compiled for the 21264 (EV6) implementation and include implementation specific optimizations. The compiler inserts nops primarily to avoid more than one branch per group of four aligned instructions (to facilitate line prediction) and to align branch targets. Because our simulator models a fetch architecture very similar to that of the 21264, these nops are useful in the simulated architecture. Like the 21264, our simulator drops nops at decode time, so they do not consume scheduling, window or retirement resources. The results include nops, but they are unimportant as it is the relative performance (MSSP vs. sequential) that is important.

Because my detailed timing simulator incurs a slowdown of a factor of 10,000 from native execution, simulating the full reference inputs to completion is impractical. In an attempt to make simulation of these programs practical, I have modified the input parameters or data sets of the reference inputs to reduce their run time. I chose this approach, rather than using the training inputs, because in some cases the data working sets of the reference inputs are significantly larger. In all cases, I tried to modify the input sets to maintain the size of the working set. Some benchmarks perform similar operations multiple times, and this reduction can be performed trivially by specifying that fewer operations be performed. In a few cases (`twolf`, `vortex`, and `vpr`), the operations performed were simplified to reduce the run time. The set of input parameters used are shown in Table 5.3.

5.2 Results

In this section, I present the experimental results. The most important results can be summarized as follows:

TABLE 5.3 Input sets used for simulation of Spec 2000 integer benchmarks

Benchmark	Input Parameters	Notes
`bzip2`	input.source 10	reference input, first 10 MB
`crafty`	crafty.in	reference input truncated to only play two games
`eon`	chair.control.kajiya chair.camera chair.surfaces chair.kajiya.ppm ppm pixels_out.kajiya	reference input
`gap`	-l ./input -q -m 64M < train.in	train input
`gcc`	integrate.i -o integrate.s	reference input
`gzip`	input.source 10	reference input
`mcf`	train.in	train input
`parser`	2.1.dict -batch < train.in	train input
`perl`	diffmail.pl 2 350 15 24 23 150	train input
`twolf`	ref	reference input without "slow 10" directive
`vortex`	lendian2.raw	reference input, with number of tasks reduced: LOOKUPS 1000, DELETES 2000, STUFF_PARTS 2000
`vpr`	net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 100 -initial_pres_fac 1000 -max_router_iterations 1	reference input for routing phase, modified to increase the overuse penalty and limited to a single iteration

- Speedups up to 1.75 (1.25 harmonic mean) can be achieved on full runs of the reduced inputs. The two largest speedups are achieved for `gcc` and `vortex`. `Gcc` has a number of loops with low iteration counts where (in the common case) the iterations are independent from each other and the code that follows. In `vortex`, many functions have error checking code that is removed and a single dominant path that allows a significant degree of distillation.
- Distilled programs can accurately predict task live-in values with the necessary coverage. Task mis-speculations typically occur only once every 10,000 instructions or more.
- These accurate predictions limit the execution paradigm's sensitivity to inter-processor communication: raising the communication latency from 5 cycles to 20 cycles only reduces the MSSP speedup 10%.
- The MSSP-specific hardware storage requirements (beyond a chip multiprocessor) are modest. For example, only 24kB of storage resources are required at the L2 to hold checkpoint, live-in, and live-out values.
- The root optimizations only account for about one-third of the effectiveness of distillation (as determined by speedup); dead code elimination and the other supporting optimization each account for one-third of the speedup in our current distiller.
- The program distiller is largely insensitive to root optimization thresholds. Most of the benefit of the root optimizations is achieved at the lowest threshold settings because it comes from removing never-observed behaviors.

I first present some supporting numbers exploring the effectiveness of optimization (Section 5.2.1), task selection (Section 5.2.2), hardware resource utilization (Section 5.2.3), and indirect branch target mapping (Section 5.2.4). Performance results are presented in Section 5.2.5 and correlations are drawn to distillation effectiveness. In Sections 5.2.6 through 5.2.12, sensitivity analysis is performed to explore the implementation's sensitivity to variables including communication latency, bandwidth, task size, and number of processors.

The performance results in Section 5.2.5 are derived from complete runs of the benchmarks. The supporting results—where multiple MSSP simulations are compared—are simulations of a 5 billion instruction sample started after 5 billion instructions. This sample is representative for all of the benchmarks except `gcc`, which performs particularly well in this sample.

5.2.1 Distilled Program Optimizations

Distilled program optimizations can be divided into two categories: 1) the speculative base optimizations, and 2) the non-speculative supporting optimizations. The root optimizations are pictorially represented in Figure 5.2. The results shown in Figure 5.4 and Figure 5.3, demonstrate the effect of the root optimizations. For each optimization, I vary the correctness threshold provided to the distiller (plotted on the X axis); as the threshold is decreased there are more opportunities for application of an optimization, because the distiller is less demanding that an optimization preserve correctness. I have plotted both the the number of times each optimization was applied to the static program and the resulting number of dynamic instructions removed for a range of different input threshold parameters (allowing from 0.01 to 2.0 percent incorrect results). For example, the point in the first graph at (99.9%, 350) indicates that with a 99.9% threshold, 350 static branches were removed from `gcc`.

The first thing to notice is that biased control-flow elimination—the first two optimizations—have a qualitatively different shape than the four other optimizations; they have increasing opportunity with higher threshold values. This shape is due to a distribution of branch biases. The effectiveness of the second optimization is not monotonically increasing because the two optimizations interact and non-dominant entries are eliminated after non-dominant branch targets. For the most part **the other four optimizations are independent of threshold**, indicating that most exploitable instructions are always or almost always safe to remove.

From the dynamic instruction counts in Figure 5.4 and Figure 5.3, one can see that a non-trivial number of instructions are removed. The larger number of static idempotent operations removed is partly an artifact; if an instruction always writes the value zero into a register it is counted in these results. For some optimizations and benchmarks, the numbers range into the billions of dynamic instructions: a substantial number for these executions that range from 9 to 40 billion instructions. Nevertheless, the base optimizations by themselves are seldom sufficient to get sizable performance gains. However, they also create opportunities for the non-speculative supporting optimizations.

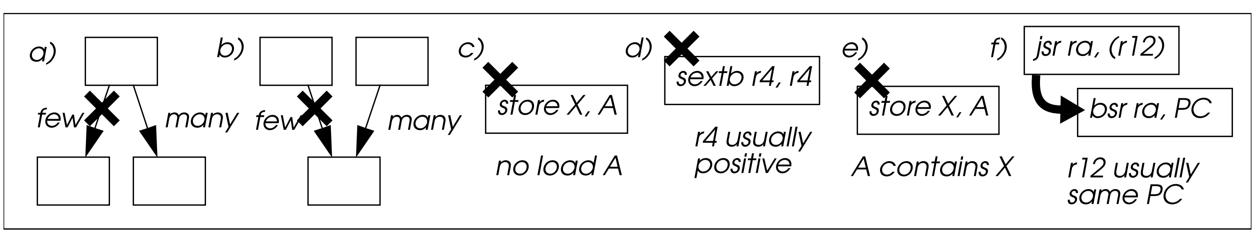


Figure 5.2: Root optimizations represented pictorially. *Optimizations involve removing or transforming instruction and are shown with their pre-condition. (a) branch elimination removes non-dominant branch targets, (b) entry elimination removes non-dominant control-flow sources, (c) long dependence store elimination removes stores with distant first uses, (d) idempotent operation elimination removes instructions that mostly produce one of its inputs as its output, (e) silent store elimination removes stores that mostly write the value already present in a memory location, and (f) indirect-to-direct call conversion is applied when an indirect call has a single dominant target.*

In Figure 5.5, I compare the relative effectiveness of the root and supporting optimizations. I plot the *distillation ratio* the number of dynamic instructions executed by the distilled program relative to the original program, and the average distance (in original program instructions) between task misspeculations for three configurations: 1) only root optimizations, 2) root optimizations and dead code elimination (DCE), and 3) all root and supporting optimizations. **In general, DCE and the set of other supporting optimiza-**

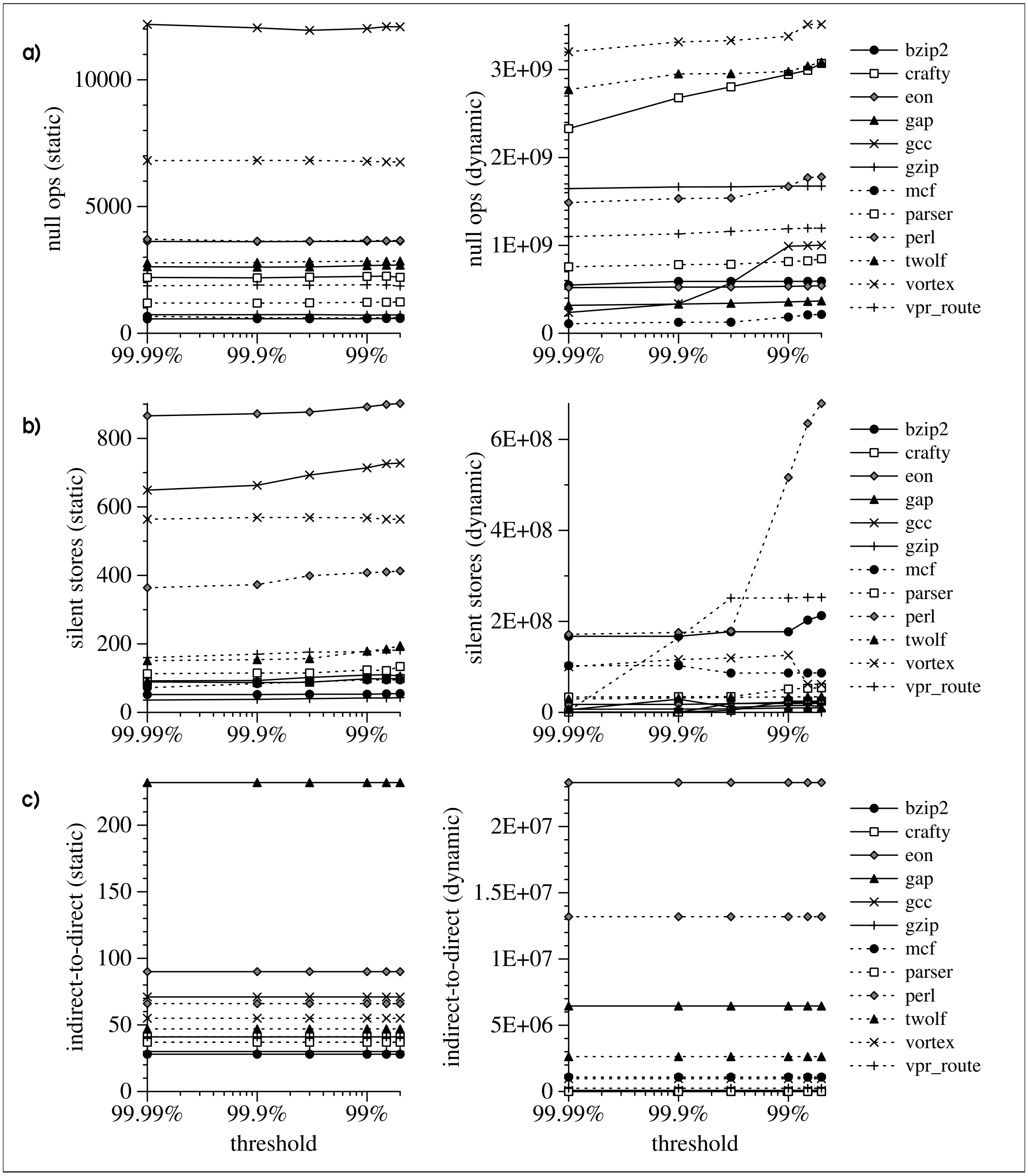


Figure 5.3: Effectiveness of root optimizations(2). *Idempotent operation elimination (a), silent store elimination (b) and indirect-to-direct call conversion (c) are all largely independent of threshold.*

tions each reduce the distillation ratio about as much as the root optimizations. For the most part this reduction in distillation ratio comes with little impact on misspeculation frequency. Only `perl` observes significant impact: the cause appears to be due to the master thread getting too far ahead of the slaves (causing what were long dependences to become short dependences).

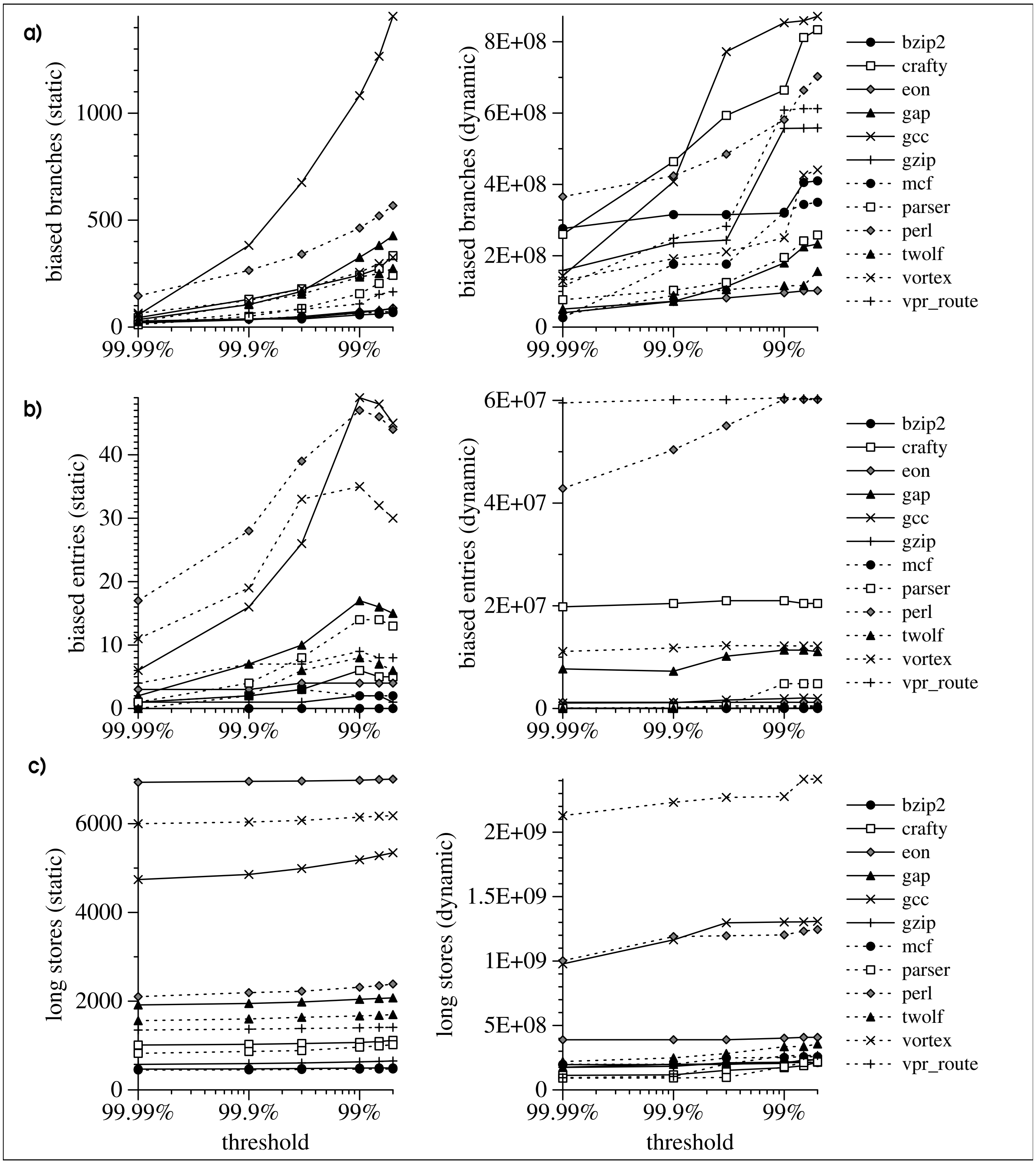


Figure 5.4: ***Effectiveness of root optimizations(1).*** *Biased branch (a) and biased entry (b) elimination have more opportunities for optimization with higher thresholds, while long dependence store elimination (c) is largely independent of threshold.*

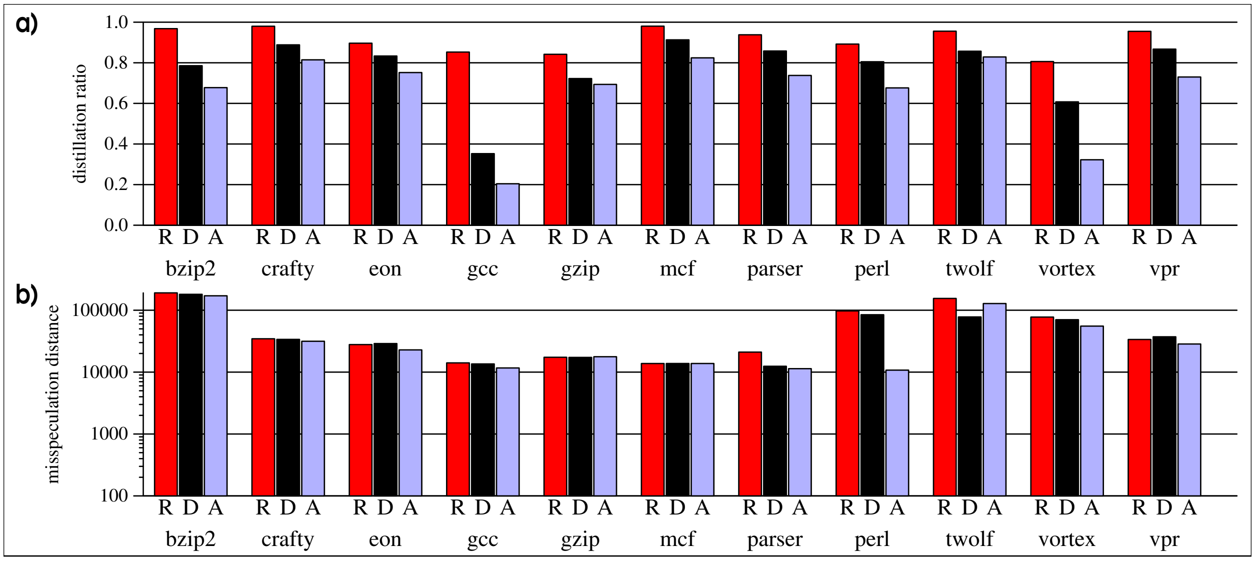


Figure 5.5: ***Relative effect of root and supporting optimizations.*** *a) distillation ratio (relative number of dynamic instructions executed in distilled and original programs, and b) average distance between task misspeculations. Data shown for (R) only Root optimizations, (D) root optimizations and Dead code elimination, (A) All root and supporting optimizations.*

5.2.2 Task Selection

By default, the program distiller is requested to create tasks of approximately 250 instructions. There are not always suitable task boundaries at this interval, so the distiller satisfies this task size goal as best it can. The resulting distribution of task sizes is shown in Figure 5.6a, as a cumulative distribution. For most benchmarks the average task size (shown in Table 5.4) is just under 200 instructions. Without task boundary suppression (Section 3.2.3), the average task size is smaller, as shown in Figure 5.6b. Five benchmarks—`bzip2`, `gcc`, `gzip`, `mcf` and `parser`—suffer from a large number of small tasks; without task boundary suppression these benchmarks experience significant slow-downs relative to a uniprocessor execution.

Additional data about the constructed distilled programs is shown in Table 5.4. **For most benchmarks, the static distilled program is substantially smaller than the original program**; *vortex* is a

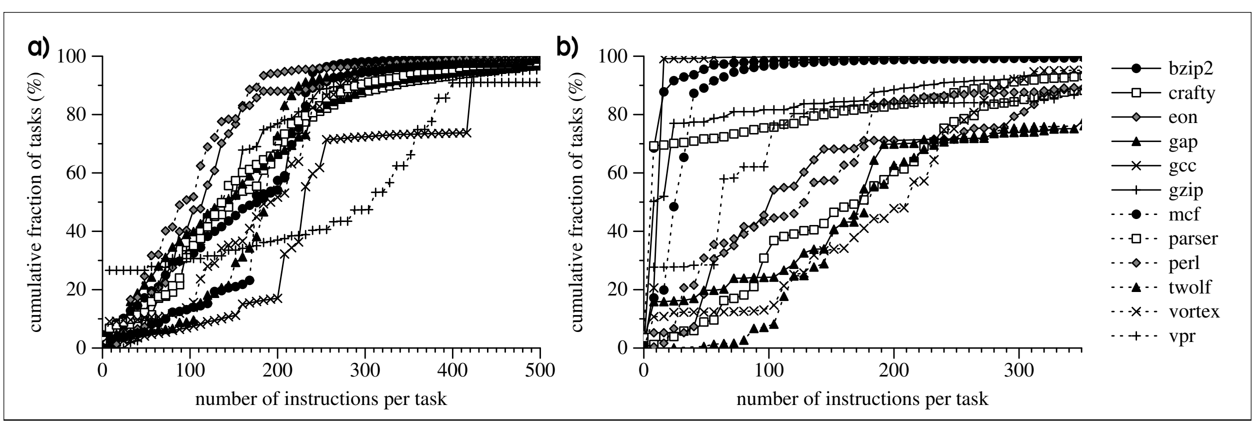


Figure 5.6: ***Task size distributions, with and without task boundary suppression.*** *a) Most benchmarks have most of their tasks in the range of 100 to 300 instructions. b) Without task boundary suppression benchmarks* `bzip2`*,* `gcc`*,* `gzip`*,* `mcf` *and* `parser` *are dominated by small tasks (less than 50 instructions).*

TABLE 5.4 Distilled program statistics

Bench mark	original program size (insts)	distilled program size (insts)	number of static task bound's	avg. dyn. task size	indirect map size (kB)	loads / 1k inst orig/dist	stores / 1k inst orig/dist	discont. fetch / 1k inst orig/dist
bzip2	39,000	9,000	188	140	1	216/267	46/82	62/110
crafty	93,000	41,000	364	182	4	242/296	45/55	35/84
eon	161,000	51,000	952	129	9	210/266	157/215	32/83
gap	215,000	35,000	844	213	8	194/236	61/89	57/134
gcc	452,000	133,000	2968	195	30	109/280	89/135	46/112
gzip	43,000	6,000	113	280	1	155/190	34/69	33/74
mcf	34,000	4,000	99	190	1	266/321	33/62	79/153
parser	67,000	27,000	459	196	4	167/242	53/112	59/115
perl	187,000	47,000	894	112	14	214/279	108/167	54/109
twolf	99,000	27,000	732	188	4	192/233	60/74	48/78
vortex	166,000	109,000	2161	186	14	108/272	56/173	11/106
vpr	75,000	25,000	488	344	4	247/313	78/107	31/58

notable exception because it performs a lot of function in-lining, which involves code replication. The number of static task boundaries correlates strongly to the size of the static distilled program; for each task boundary there is one static `CHKPT` instruction, one in-transition entry into the distilled program, and one out-transition block with a `VERIFY` instruction. The average size of a task in (original program) dynamic instructions indicates how often the `CHKPT` and `VERIFY` instructions are encountered during execution. The number of indirect branch targets used by the distilled program, which need to be in the indirect map table, is also correlated to the size of the static distilled program. I also present the distillation ratio for loads and stores demonstrating that my optimizations can be effective at reducing the number of required cache accesses. Lastly, I show the number of discontinuous fetches (taken branches) for the distilled and original program. This metric is improved by the profile-driven code layout. The benefits of `vortex`'s aggressive inlining can be seen by its **factor of 10 reduction** in this metric.

5.2.3 Hardware Resource Utilization

Next, I present the hardware resources required to make the paradigm work, namely interconnection network bandwidth, non-architectural data storage, and processor utilization.

Communication Bandwidth. Measured bandwidth utilization corroborates our estimates in Section 4.4.3. Figure 5.7, shows the average number of bytes of data communicated per instruction for each major type of entity: master processor, slave processor, L2 cache bank, and GRF. This data may be slightly pessimistic due to a naive cache refreshing policy. Cache refreshing (discussed in Section 4.5.8) is responsible for many of the fills requested by the master. Currently, my refresh policy errs on the side of refreshing too often, naively requesting a refresh for a block at the end of a task in which it was written. I am confident that a more intelligent technique could achieve the same results with less bandwidth.

Non-architectural State Storage. A moderately small amount of storage is required for speculative data. In our current model, the amount of such data is currently not limited, allowing us to measure the requirements of each workload. Figure 5.8 shows the amount of storage that is sufficient for 98% of the cycles;

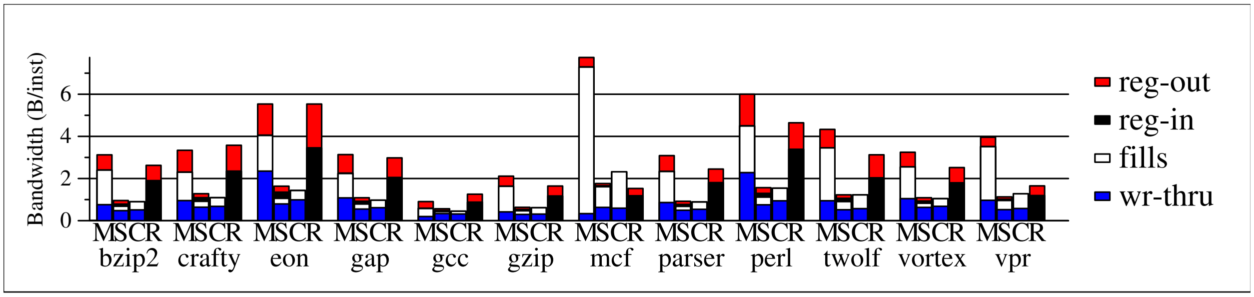


Figure 5.7: Bandwidth utilized. *Although "bandwidth can be bought", our MSSP implementation uses quite a bit. Utilization is shown in bytes per original program instruction for each type of entity (M = Master, S = Slave, C = L2 Cache bank, R = global Register file). So far little has been done to optimize bandwidth utilization.*

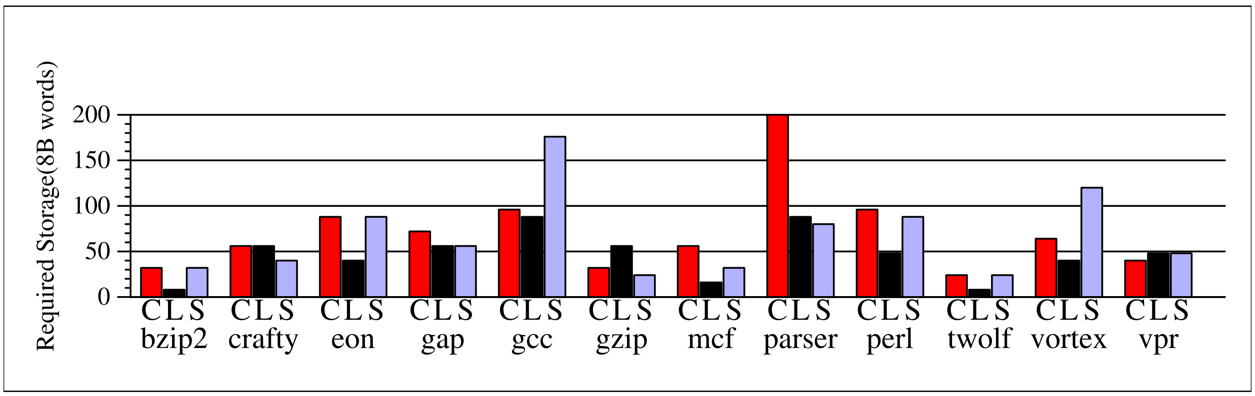


Figure 5.8: Amount of speculative state storage required. *Little speculative storage is necessary. The number of 8B memory words sufficient to buffer state in 98% of cycles is shown (b). Rarely are more than 128 words of storage required of any type (C = checkpoint, L = live-in, S = live-out stores).*

storage for each type of data (memory checkpoints, live-in values, and live-out values) is plotted separately. When a lot of storage is required, it is to keep many tasks in flight simultaneously (as will be shown next, in Figure 5.9). Since this behavior is desired, the implementation should have sufficient resources to

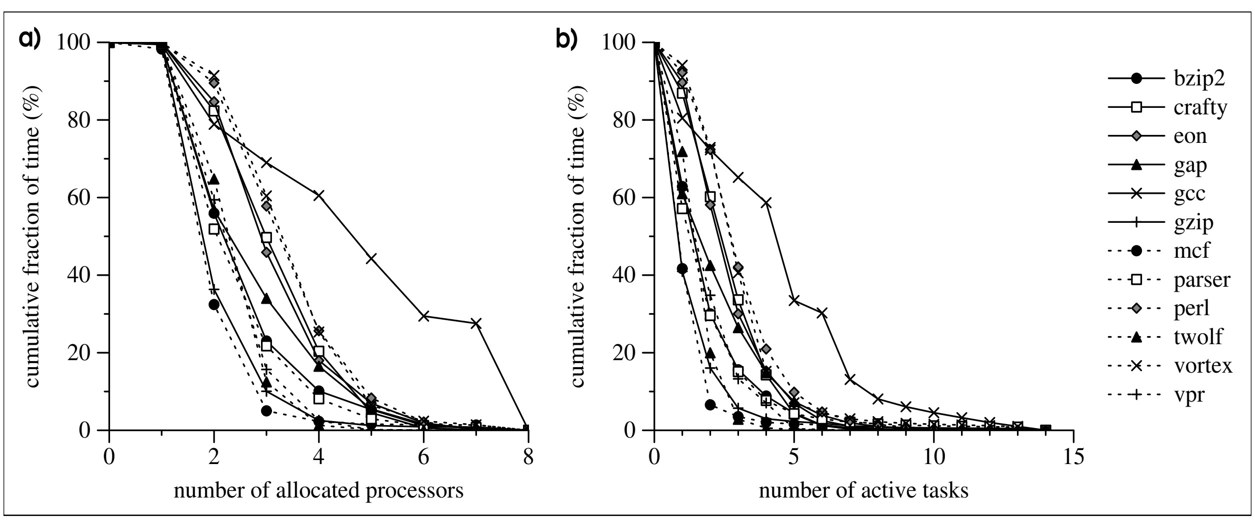


Figure 5.9: Processor utilization and task activity distribution: *(a) distribution of number of processors allocated at any given time, and (b) distribution of number of tasks active (i.e., executing or completed and waiting for commitment).*

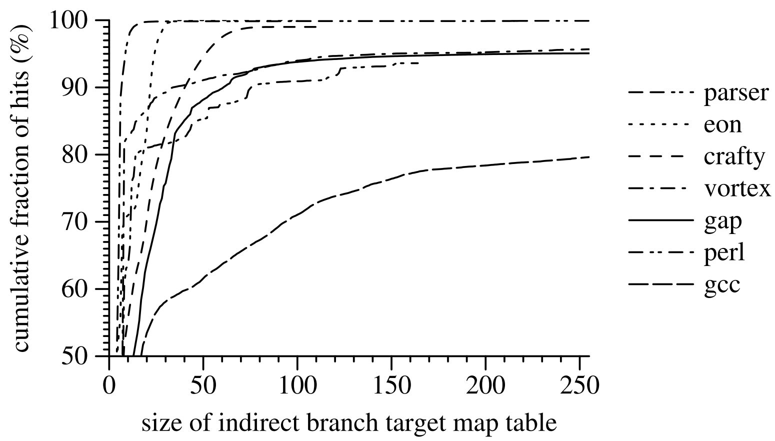


Figure 5.10: Hit ratio of as a function of indirect branch target map size. *The lines that end before 256 entries have no capacity misses, only compulsory misses. Benchmarks* `bzip2`*,* `gzip`*,* `mcf`*,* `twolf`*, and* `vpr` *are not shown as they have a negligible number of misses.*

support it. Taking the load imbalance into account, providing each bank with 64 entries of each type should be sufficient for these workloads. Since checkpoint, live-in, and live-out entries require about 16B of storage each, **a structure of this size would require about 24KB of storage, or a little more than 1% of a 2MB L2 cache**.

Processor Utilization. Although the simulated machine has 8 processors available, they are rarely being used simultaneously. Figure 5.9a shows a cumulative distribution of cycles of the number of processors (either master or slave) that were allocated. Because distillation ratios from the current distiller prototype are reasonably low and the implementation uses processors efficiently—processors can be reallocated when a task is completed, not when it commits—for most of the benchmarks **4 or fewer processors were necessary 90% of the time**. Only `gcc` can regularly keep more than 4 tasks active simultaneously. Performance sensitivity to number of processors is measured in Section 5.2.10.

5.2.4 Mapping

As I described in Section 4.5.7, a mechanism is required for mapping between the original and distilled programs. In this sub-section, I demonstrate that there is a working set of indirect branch targets in the distilled program. Figure 5.10 shows the hit ratios for a range of fully associative tables with least recently used (LRU) replacement, up to 256 entries. For the most part, the working sets would fit in a 64-entry fully associative cache with a least-recently-used policy. All benchmarks except gcc have at least an 85% hit ratio; indirect branches are rare enough in gcc's distilled program that it only misses every 3,000 instructions with a 64-entry cache. I believe these results demonstrate that indirect branch target maps can be effectively cached.

One reason that the working sets are small is that some indirect branches that are present in the original program have been removed from the distilled program. If an indirect branch has a single dominant target, it can either be converted to a direct branch or removed altogether. Furthermore, returns are a big source of indirect branches and they are frequently removed by function inlining. Table 5.5 shows the average reduction of dynamic instances of indirect branches, which ranges from 35 to 100 percent.

TABLE 5.5 Number of indirect branches per 1000 dynamic (original program) instructions

Benchmark	bzi	cra	eon	gap	gcc	gzi	mcf	par	per	two	vor	vpr
original	15	19	35	37	6	6	34	26	36	13	28	10
distilled	0	5	10	26	2	1	1	9	24	1	4	0

5.2.5 Performance

The MSSP performance simulations are compared to a base case in which a single processor of the CMP is used to execute the program as a traditional uniprocessor. Speedups are computed from execution times (in cycles). All references to retired instructions per cycle (IPC) relate to original program instructions; such comparisons are meaningful because the same original program instructions are committed—in the same order even—by both execution paradigms.

I first present data from the whole program's execution; Figure 5.11 shows data on task misspeculation frequency, distillation effectiveness, and speedup together to allow correlations to be drawn visually. In Figure 5.11a, I show the average distance between task misspeculations (in dynamic instructions). This distance ranges from 10,000 to over 100,000 instructions, demonstrating that task misspeculations can be made rather infrequent. Figure 5.12 plots the misspeculation detection latency. While there is wide distribution of detection latencies, most misspeculations are detected in less than 300 cycles of starting a task.

The misspeculation frequency data is meaningless in isolation, in Figure 5.11b I try to quantify the effectiveness of the distillation that corresponds to this misspeculation frequency. Quantifying distillation

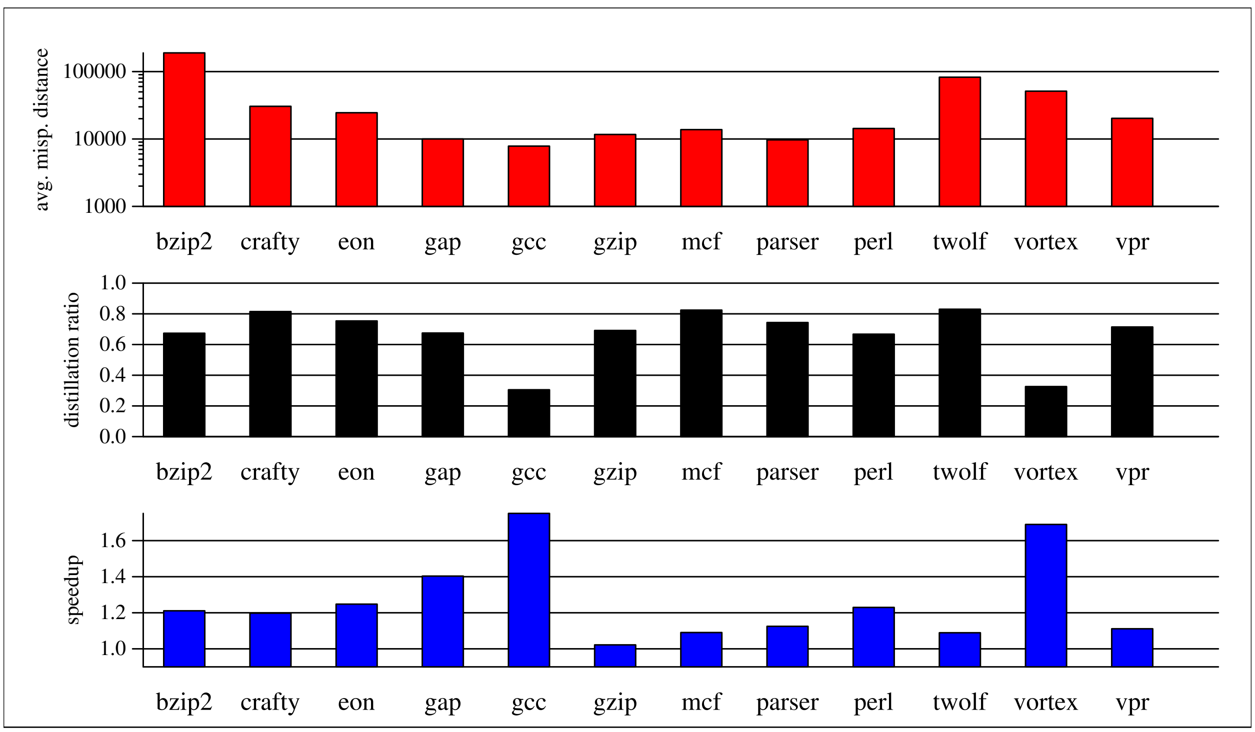


Figure 5.11: MSSP Performance across the whole program's execution. *Task misspeculations (a) are infrequent; the smallest average distance (in dynamic instructions) between misspeculations is just under 10,000 instructions, and for some benchmarks it is more than 100,000 instructions. The distillation ratio (b), the ratio of the number of non-nop instructions in the distilled program vs. the number of non-nop instructions in the original program (smaller is better), estimates the effectiveness of distillation. This figure correlates well to the speedups (c) achieved, which range from 1.0 (no speedup) to 1.75.*

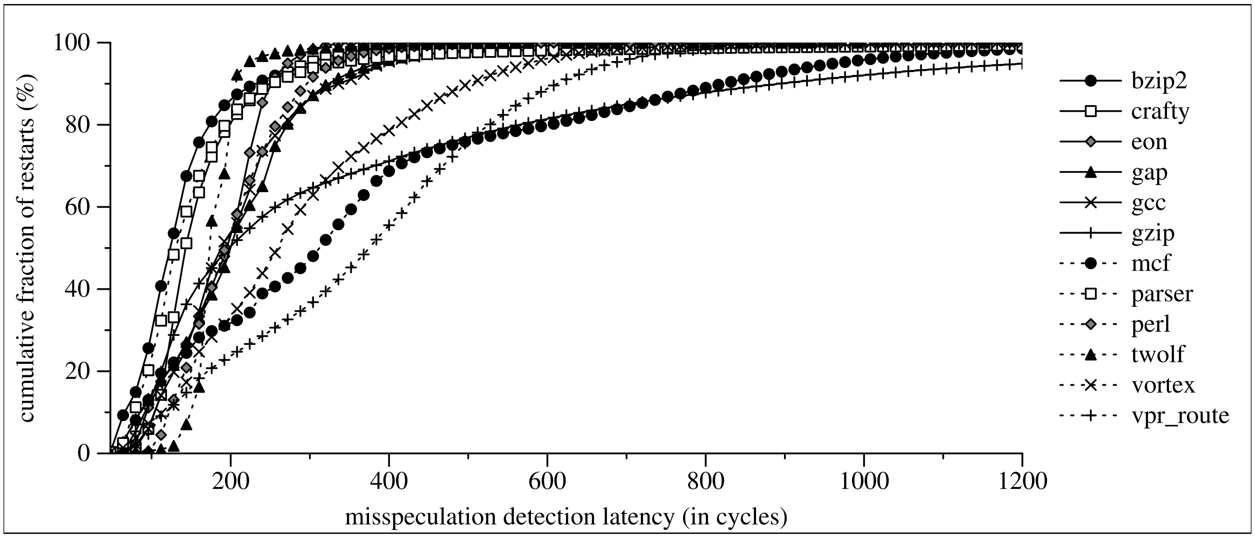


Figure 5.12: Task misspeculation detection latency. *Most task misspeculations are detected within 300 cycles of starting the slave task. Longer detection times are due to long tasks preceding the misspeculation.*

independent of an implementation model is difficult; as an attempt I have defined the *distillation ratio* to be the number of non-nop instructions "retired" by the master processor relative to the number of non-nop original program instructions retired by the slaves. Although this metric cannot quantify the benefit of IPC improving optimizations (*e.g.*, pre-fetching, if-conversion, global scheduling, etc.), it is a decent metric for my current distiller implementation as few IPC improving optimizations are implemented.

The distillation ratio varies greatly between benchmarks. My current distiller only effectively distills two of the benchmarks `gcc` and `vortex`. In these two benchmarks the master executes 70% fewer instructions than the original program; the rest of the benchmarks only observe a 20% to 35% reduction. `Gcc` has a number of loops with small loop bodies whose iterations are independent from each other (in the common case), like the one shown in Section 2.2.5. Most of the code in these loops can be removed by the distiller leaving only the updates of the induction variable. Then the loop in the distilled program can be unrolled by just scaling the values added to (or subtracted from) the induction variables. The result is that the master executes about 5 instructions for each task of 200-250 instructions.

`Vortex` has a number of factors contributing to the effectiveness of its distillation. First, it has a lot of error checking code, much of which is removed by biased branch and dead code elimination. Second, many small functions have a single dominant path, allowing all branches to be removed from them. This action both leads to a significant reduction in the computation performed by the function and sometimes transforms non-leaf functions into leaf functions. Third, `vortex` is very call intensive; the function call overhead is greatly reduced by inlining leaf functions (some of which were created by non-dominant path elimination). If all of a function's calls are inlined, it too becomes a leaf function, and is considered for inlining. In `vortex`, such inlining is often carried three or four levels up the call graph.

In Figure 5.13, I show the same metrics as in Figure 5.11, but break the simulations in five billion instruction segments to explore the variations between segments of the execution. My reduced input sets for the benchmarks have runs that range from just under 10 to almost 45 billion instructions, resulting in between 2 and 9 segments, inclusive, per benchmark. As can be seen, most benchmarks have little variation over their execution at this granularity, although some benchmarks demonstrate variation in their initiation and/or termination segments from the rest of the segments. The main exception is `gcc`, which performs significantly better in its second segment (the segment starting after five billion and ending at 10

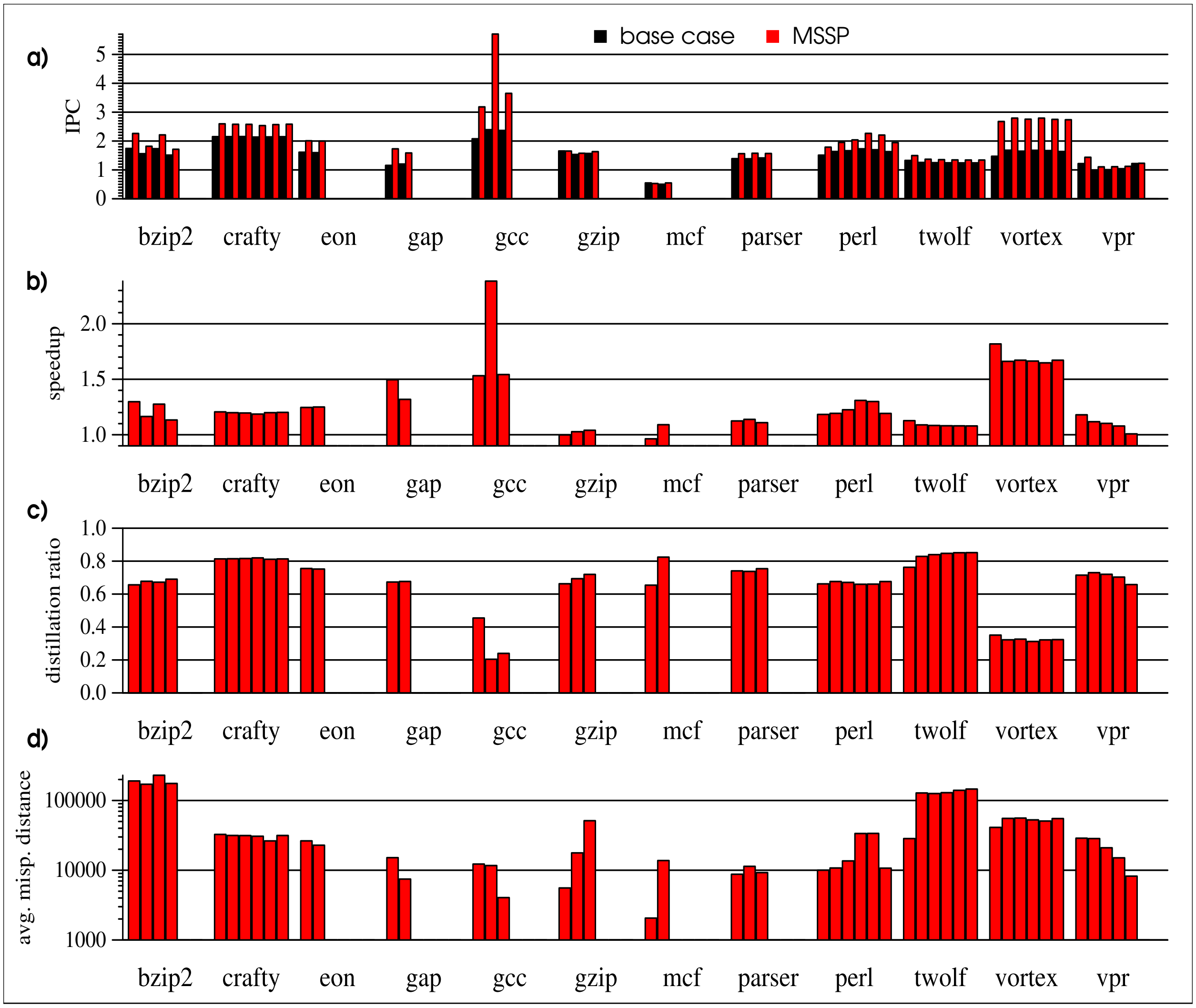


Figure 5.13: Performance variation across execution. *Each benchmark's execution is broken into five billion instruction segments. IPC (a), speedup (b), distillation ratio (c), and average misprediction distance (d) data for is shown for each segment. Shorter running benchmarks execute fewer instructions. For many benchmarks there is little variability between segments. Gcc is a noteworthy exception, where the second segment (the one I used for sensitivity analysis) achieves a significantly higher speedup.*

billion instructions). I point this fact out because it is this second segment that I plot for all of the supporting and sensitivity analysis.

5.2.6 Sensitivity to Interconnect Latency/Bandwidth

One of the goals of the MSSP paradigm is to tolerate inter-processor communication latency, and, for the most part, this goal is achieved. Figure 5.14 compares the baseline and MSSP implementation for on-chip one-way communication latencies of 5, 10, and 20 cycles. Both base and MSSP executions are affected as the L2 hit time has increased from 12 to 22 to 42 cycles. For the most part there is only moderate impact on the MSSP speedups. Only `gcc` and `vortex` observe a significant relative performance

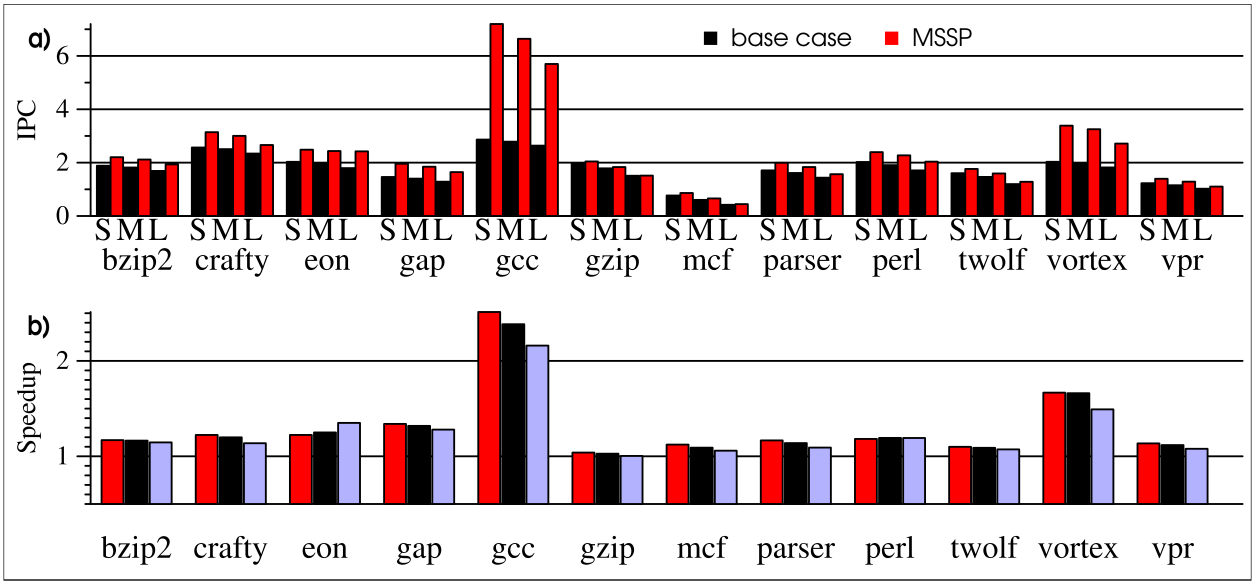


Figure 5.14: Limited sensitivity to inter-processor communication latency. *As the network communication latency between processors and the L2 cache is scaled from 5 cycles (S = short) to 10 cycles (M = medium) to 20 cycles (L = long)—causing the minimum L1 miss penalty to go from 12 to 42 cycles—relative performance on most benchmarks diminishes slightly, and performance relative to the base case improves with increased latency on* `eon`.

reduction. The MSSP execution for the benchmark `eon` is actually more tolerant of inter-processor communication latency than the baseline uniprocessor execution.

The implementation is even less sensitive to the amount of bandwidth available, at least around this operating point. Figure 5.15 shows the performance sensitivity as I reduce the inter-processor bandwidth from 1 message per cycle, to 1 message every 2 and 3 cycles.

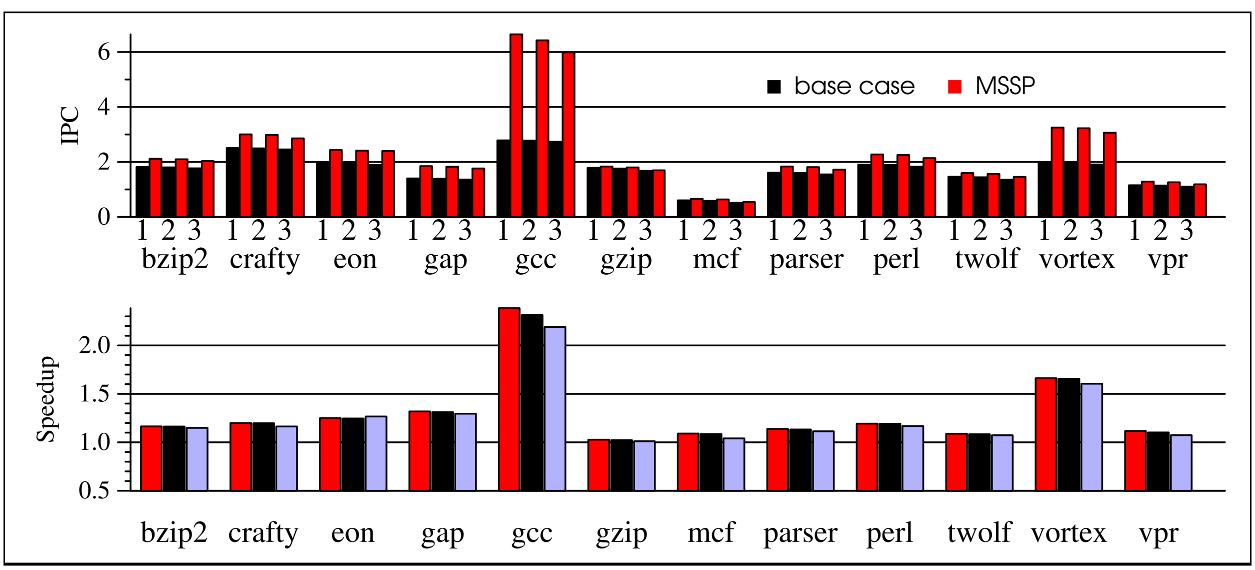


Figure 5.15: Limited sensitivity of inter-processor bandwidth on performance. *IPC (a) and speedup (b) are shown for configurations that can communicate a message every **1**, **2**, and **3** cycles.*

5.2.7 Sensitivity to Task Size

With task boundary suppression in place, the execution is somewhat insensitive to task sizes. In Figure 5.16a, I show the resulting average task sizes when the distiller is charged with creating tasks of 100, 250, and 500 instructions. As can be seen, the distiller usually undershoots its mark, largely because of frequently executed loops below the target task size. Figure 5.16b shows that performance on most benchmarks is mostly insensitive to task size—with performance slightly increasing on average with larger tasks—but gcc and vortex have distinctly better performance with 250 instruction tasks. The sensitivity of performance on task size is a complex interaction of a number of forces. In Figure 5.17, I demonstrate that distillation improves slightly with task size (recall that a smaller distillation ratio is better), and generally

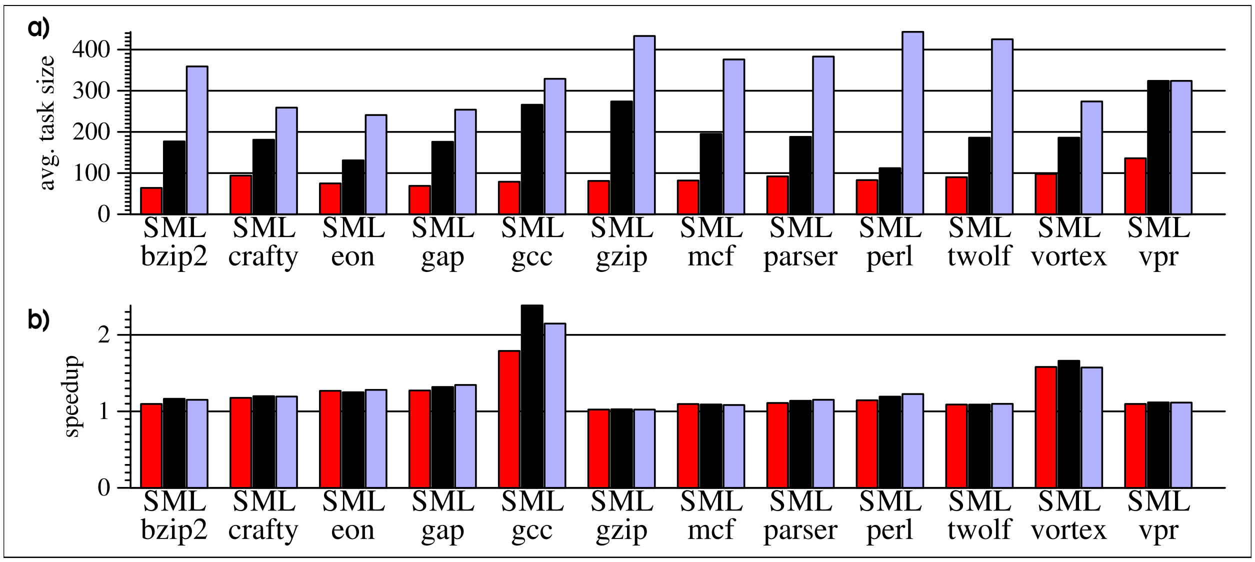


Figure 5.16: Sensitivity of task size on performance. *When directed to construct tasks of three different sizes—100 instruction (S=short), 250 instruction (M=medium), and 500 instruction (L=long) tasks—the program distiller obliges but generally constructs undersized tasks (a). For the most part performance is insensitive to task size (b).*

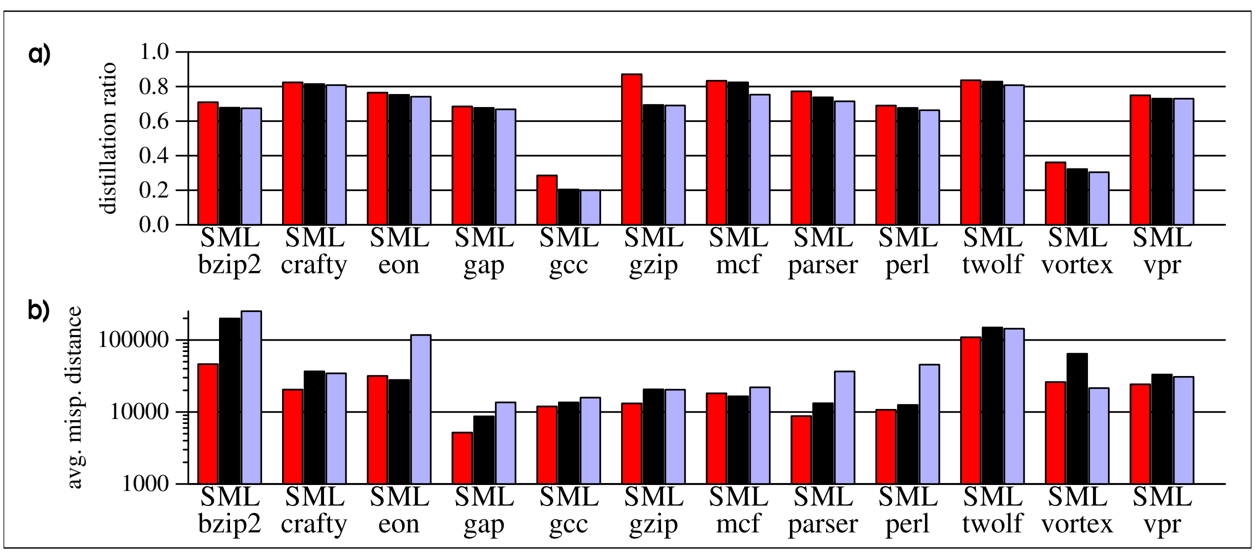


Figure 5.17: Impact of task size on distillation ratio and misspeculation frequency. *As task size increases, distillation is slightly more effective (a) and misspeculations are less frequent (b), because there are fewer tasks to be misspeculated.*

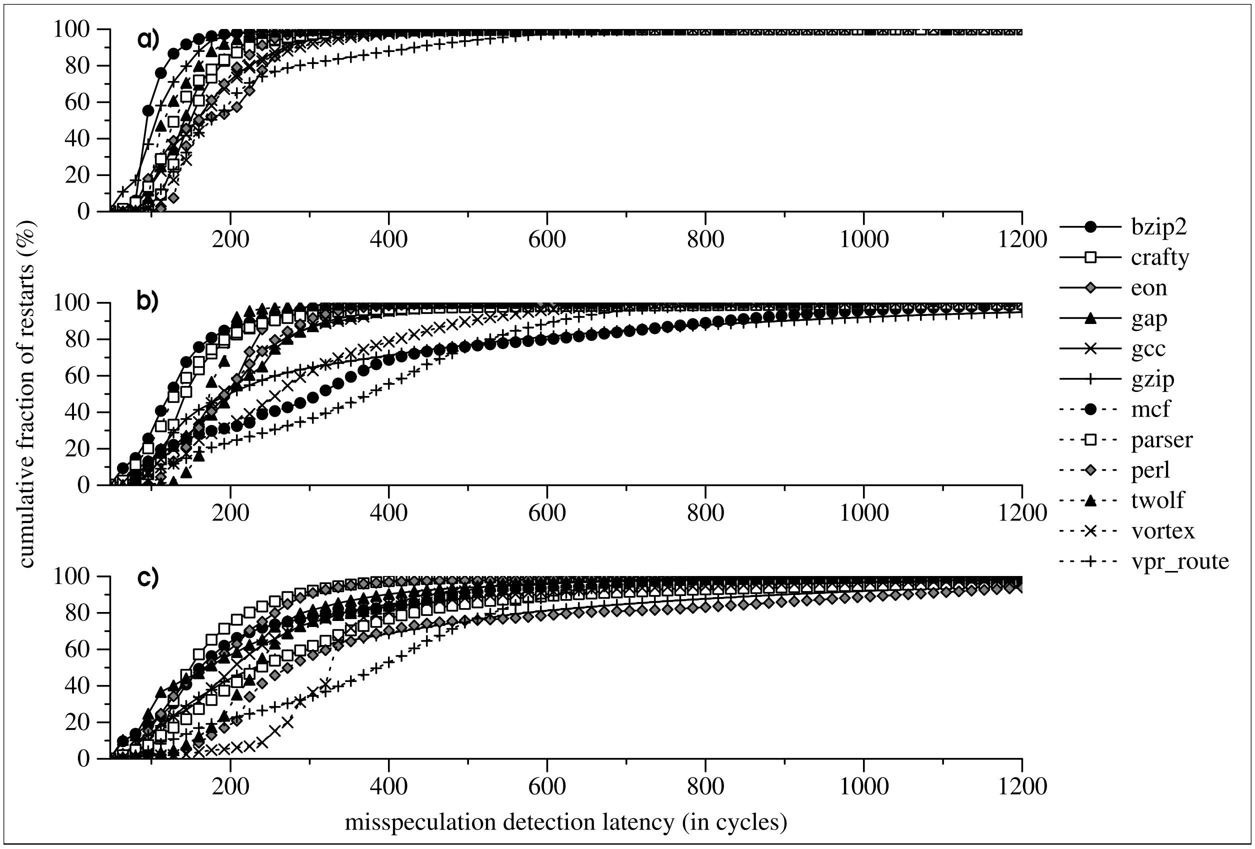


Figure 5.18: Sensitivity of task size on task misspeculation detection latency. *Longer tasks generally have longer lifetimes (i.e., time between allocation to a slave processor and commitment) than short tasks. Data shown for 100 (a), 250 (b), and 500 (c) instruction tasks.*

the distance between task misspeculations is increased, although this is mostly an artifact resulting from there being fewer tasks to be misspeculated. The factor that has the largest negative impact is the task misspeculation detection latency (shown in Figure 5.18). As the desired task size is scaled from 100 to 500 instructions, the median latency to detect a misspeculation increases from 150 cycles to 300 cycles, on average.

Much larger than its effect on performance, variations in task size impact the bandwidth/storage tradeoff. As is shown in Figure 5.19, larger tasks use less inter-processor bandwidth (on a per instruction basis), but require more buffering for non-architectural data, as the larger tasks have more state per task.

5.2.8 Sensitivity to Task Boundary Selection

To explore the sensitivity to the heuristics used to select task boundaries, I ran three experiments. The first experiment, rather than always putting loop task boundaries before the loop header, picked a random block on the dominant path of the loop. The second experiment ignored the after loop and after call heuristics for placing task boundaries; instead potential task boundaries were placed before every basic block. The third experiment varied the location within a basic block task boundaries were place; instead of always inserting the task boundary before the first instruction of the block, a random instruction in the block was selected.

The performance of these experiments is compared to the baseline task selection algorithm in Figure 5.20. For the most part, these alternate task selections perform within 5% of the baseline algorithm

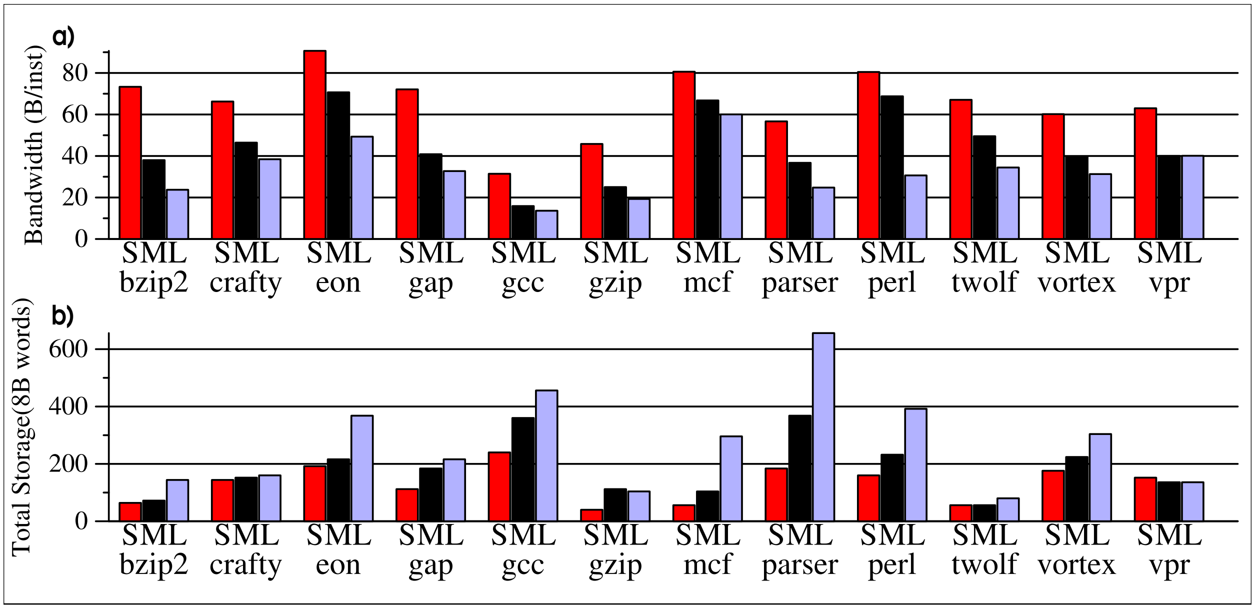


Figure 5.19: Sensitivity of task size on bandwidth and storage requirements. *a) aggregate inter-processor bandwidth consumed per instruction, and b) aggregate non-architectural data storage sufficient for 98% of cycles. Data shown for three task size configurations: 100 instruction (S=short), 250 instruction (M=medium), and 500 instruction (L=long) tasks.*

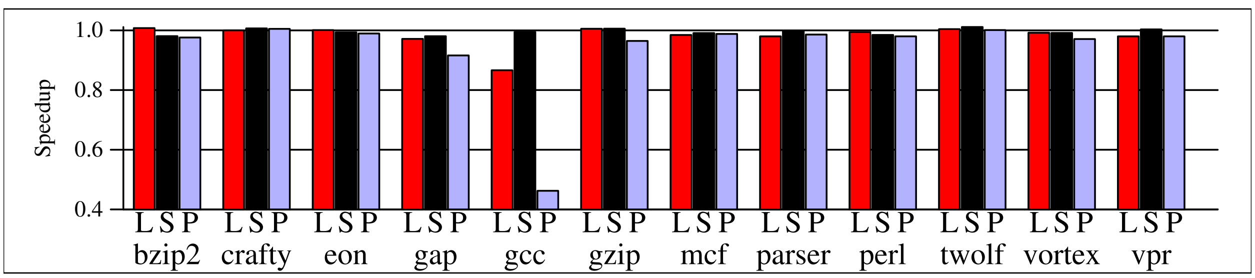


Figure 5.20: Sensitivity of performance on task boundary selection. *The performance of three alternate task selection heuristics are compared to the baseline task selection algorithm.* ***L****: insert boundary before random block for* ***L****oop tasks,* ***S****: make all basic blocks equally likely in* ***S****traight-line code, and* ***P****:* ***P****erturb task selection within a basic block by picking a random instruction to be the task boundary.*

and in a few instances give 1% speedups. In two benchmarks `gap` and `gcc` the experiments that affect loop task selection (**L** and **P**) causes the unrolling optimization to fail on a few important loops.

5.2.9 Sensitivity to Optimization Thresholds

My prototype distiller requires (as an input) threshold parameters to guide the application of the root optimizations. In experimenting with these parameters, I have found that, for the most part, the distiller is not very sensitive to their settings. In Figure 5.21, I show the speedup of a variety of configurations relative to baseline MSSP configuration. With the exception of `gcc`, these changes to the threshold values affect performance less than 5%. `Gcc`'s performance is significantly impacted by the first four configurations, because they specify a higher (99.9% vs. 99%) threshold for the number of instances that must be long for a store to be characterized as a long store, a critical parameter for `gcc`. No one configuration is best for all of the benchmarks, suggesting that these thresholds should not be specified directly. Rather, an approach that evaluates the benefit of the enabled optimizations against the cost of the induced misspeculations is likely to provide both good and robust performance.

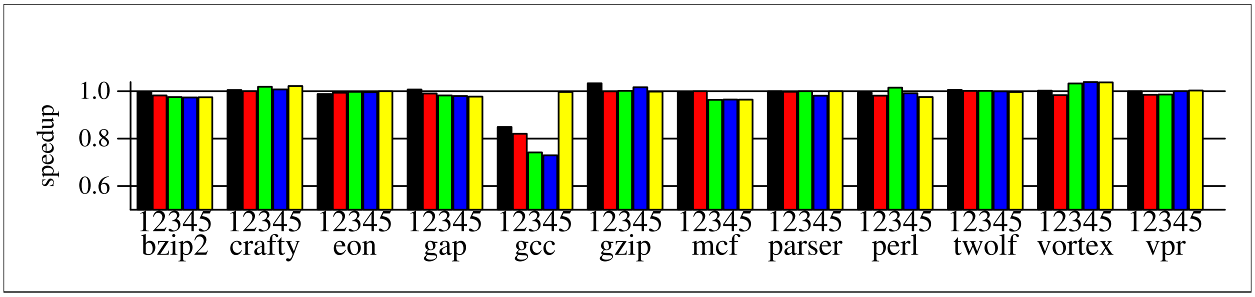


Figure 5.21: Sensitivity of distillation thresholds on performance. *Speedups are relative to the base MSSP configuration. The simulations are provided with the following input parameters:*

	Configuration					
Threshold	**base**	**1**	**2**	**3**	**4**	**5**
`branch threshold`	99%	99.5%	99%	98.5%	98%	98%
`long store threshold`	99%	99.9%	99.9%	99.9%	99.9%	99%
`idempotent threshold`	99%	99.5%	99.9%	99.9%	99.9%	99%

5.2.10 Sensitivity to Number of Processors

Earlier, I showed data (in Figure 5.9) indicating that rarely are all eight processors utilized in the default MSSP configuration. In Figure 5.22, I vary the number of available processors from three to eight. The benchmarks with little or no speedup can be satisfied with four processors, but some slowdowns occur with three. The rest of the benchmarks benefit from up to six processors and `gcc`, which demonstrated a high utilization in Figure 5.9, benefits from all eight.

5.2.11 Sensitivity to Refreshing

Periodically refreshing the master's cache (as described in Section 4.5.8) is important. Without refreshing, performance can drop as much as 20 percent, as shown in Figure 5.23.

5.2.12 Sensitivity to a Realistic Mapping Lookaside Buffer (MLB)

There is a tiny performance difference between a realistic implementation of an MLB and an ideal MLB. Figure 5.23b shows the performance of an implementation with a 64-entry MLB that has a 2-cycle access time (misses fetch page table entries from the L2 cache banks) relative to one with a perfect 0-cycle MLB. In eight cases the performance difference is negligible; the other four lose a half of a percent.

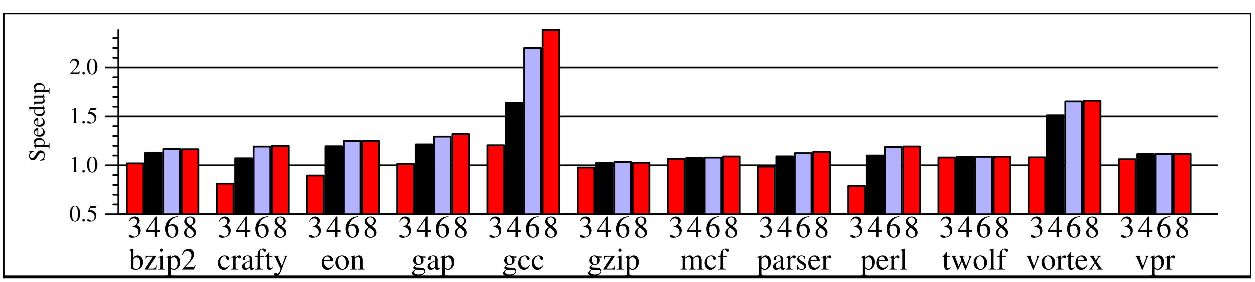


Figure 5.22: Sensitivity of performance on number of available processors. *Speedup is compared among configurations with three, four, six and eight processors. Many benchmarks benefit from up to six processors, only* `gcc` *really benefits from more than six processors.*

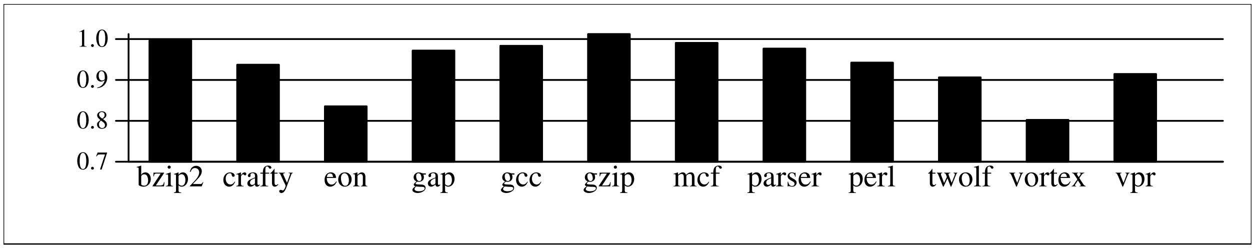


Figure 5.23: Sensitivity to refreshing. *Refreshing the master's cache periodically is important; failing to do so can drop performance by as much as 20%.*

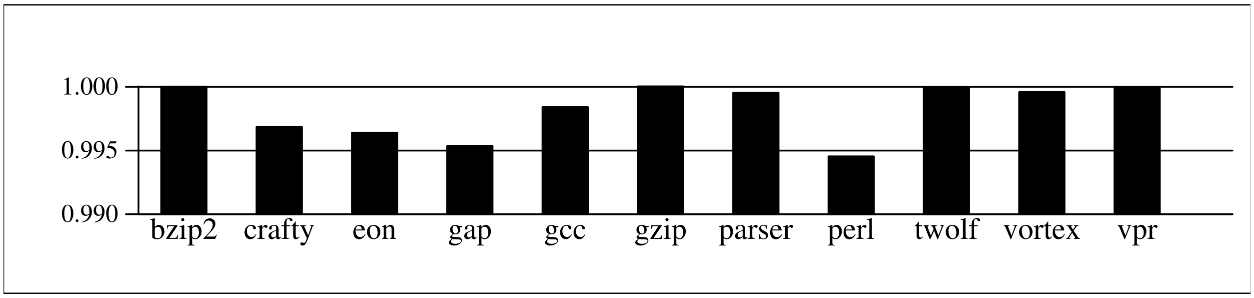


Figure 5.24: Sensitivity indirect branch target mapping. *A 64-entry indirect branch target map lookaside buffer with a 2-cycle access time gives most of the benefit of a perfect mapping mechanism; four of the 12 benchmarks are slowed by about a half of a percent with the realistic scheme.*

5.3 Chapter Summary

In this chapter I described the infrastructure I used for the experimental evaluation of the MSSP implementation and the resulting data. I briefly summarize the results as follows:

- My distiller prototype yields distilled programs that can accurately predict task live-ins (misspeculating less than once every 10,000 original program instructions retired).
- The performance of the distilled programs varies significantly between benchmarks. Although some programs may be fundamentally harder to distill, this result is in part due to the applicability of the optimizations that I have implemented to these programs.
- The performance of the MSSP implementation largely correlates to the effectiveness of distillation. Speedups ranging from 1.0 (no speedup) to 1.75 were achieved with a harmonic mean speedup of 1.25.
- The architecture is tolerant of inter-processor communication latency: only 10% slowdown is observed when the communication latency is scaled from five cycles to 20 cycles.
- The distiller is largely insensitive to both the specified optimization thresholds and task selection (both task size and task boundary locations), but the data suggests that further research could lead to improvements on both these fronts.
- The hardware requirements can be modest; for example, on the order of 24kB of speculative storage at the L2 appears to be sufficient.

Although the MSSP paradigm has some desirable characteristics, I find the overall performance of my current implementation to be disappointing. Some benchmarks—notably `gcc` and `vortex`—hint at what can be achieved, but most of the benchmarks are not effectively distilled by my current infrastructure. Still, I remain optimistic; performance appears to be limited by the effectiveness of optimization and there are many optimizations that I have not yet implemented. In short, I believe that the MSSP paradigm is deserving of continued research. In the next chapter, I describe some of the avenues I have identified for follow-on research.

Chapter 6

Conclusions

In this conclusion chapter, rather than merely summarize the definition, implementation, and performance of the Master/Slave Speculative Parallelization, I pause to reflect on some unexpected results of the research and some directions that future research should take. Specifically, I describe lessons learned during the implementation (in Section 6.1), classify mechanisms with regards to their correctness requirements (in Section 6.2), and present some open questions (Section 6.3).

6.1 Lessons Learned

I have learned many lessons in the process of studying this execution paradigm, not the least of which is that it is perhaps a reasonable way to organize computers in the future. Of course, I had that hunch or I never would have embarked on this journey. Not all of the lessons I learned were expected. In this section, I briefly summarize some factors that were not obvious to me when I initially envisioned the execution paradigm.

Program Information Content. I had expected that distillation would be more successful on some programs than others, but I did not know what the important characteristics of a program would be that would determine the effectiveness of distillation. The result of this study leads me to believe there is an inherent information content to a program and the process of distillation is akin to compression. The information content of a program is high if it performs a diverse set of computations in an unpredictable order and low if it repeatedly performs a set of operations in the same manner every time. Much like compression, distillation is most effective when its input has low information content.

The information content of programs varies in part due to the computation being performed and in part due to the way the program is written. The compression programs (`bzip2` and `gzip`) unsurprisingly consist of computation with high information content (loops with highly variant iteration counts and little redundancy), which to date my distiller cannot effectively optimize. In contrast, the benchmark `vortex` is written in a very low information content style that is call intensive and has significant amounts of error checking code. As a result, `vortex` can be effectively distilled. Perhaps the style in which `vortex` is written is more indicative of large software projects than hand-tuned programs like `bzip2` and `gzip`.

Optimization Effectiveness. Although I have forgotten exactly which optimizations I thought would be most effective, I am sure that my predictions were not entirely correct. The Slipstream work [77] demonstrated that a significant fraction of instructions can be removed as a result of the elimination of predictable branches. Also, significant are the mutually beneficial optimizations of inlining and register re-allocation that can remove much of the overhead in call intensive programs. Two particularly surprising results were the number of store instructions that could be removed because they result in long dependences and that distillation can make both paths of an unbiased branch point to the same target allowing the branch to be trivially removed.

Tight Loops. I did not anticipate the impact on task construction of loops with small loop bodies and high iteration counts. The implementation is designed to efficiently execute when tasks are a moderate size—large enough to amortize communication costs, but small enough to buffer speculatively—perhaps between a few hundred and a thousand dynamic instructions. For simplicity, I originally proposed to use static annotations of the original program to specify the boundaries of tasks. These annotations are insufficient for loops with small loop bodies (*i.e.*, less than 20 instructions) that are executed for many iterations (*i.e.*, more than a thousand). The static annotation allow either a single iteration or the whole loop to be selected as a task. Many tiny tasks or a single enormous task are both sub-optimal. As a result, I implemented task boundary suppressors (described in Section 3.2.3).

Synchronization. Another unexpected wrinkle was the impact of non-leaf functions with a variable number of arguments. In the Alpha architecture, the first six arguments to a function are passed in registers. When a function calls another function, it must save its arguments (generally on the stack) so that it can pass arguments to the called function. Even if the second function has fewer arguments, all arguments are saved to the stack because the contents of these registers are not guaranteed to remain intact across calls by the calling convention. In a function with variable arguments, this process can result in the contents of all argument registers being saved before it is determined which actually contain valid arguments.

These saves are a minor problem in a normal execution, where some garbage value is saved and restored. In an MSSP execution, these garbage values become live-in values and must be correct for the task to be verified. Ensuring that these values are correct is possible—by performing inter-procedural liveness analysis on registers and avoiding re-allocating those registers—but difficult, because they could have been written long before in many possible locations. However this approach is inefficient, as it reduces the amount of dead code that can be removed and limits how registers can be reallocated. As a result our implementation takes a different approach: synchronization. Identifying the rare variable argument functions is trivial. When one is found the distiller automatically inserts a "synchronize" instruction before the `VERIFY` instruction in the distilled program. This synchronize instruction causes the slave to stall until it becomes the head task and to perform an early verification (as discussed in Section 4.5.10) immediately before any instructions are executed. This early verification causes the architected registers to be loaded and the task to be executed non-speculatively. The stall is a performance penalty, but a smaller one than a guaranteed misspeculation.

6.2 Requirements for Correct Execution

One of the interesting aspects of the MSSP paradigm is that, by construction, a non-trivial fraction of the mechanisms involved need not be verified. Results generated by these mechanism are only used for prediction, so if they are faulty, they will only result in mispredictions that will potentially hurt performance, but not correctness. In this section, I briefly outline the portions of the design that must be correct or can be flawed.

6.2.1 Mechanisms That Must Be Correct

Instruction Fetch, Execute, and Commit. The reuse test used to validate task live-ins does not re-execute instructions at commit time, so MSSP relies on correct execution of the task by the slave processor. Techniques like Diva [4] can be used to ensure this correctness.

Collection of Live-ins and Live-outs. The execution of the task is abstracted down to its live-in and live-out sets. An error in a live-in set could allow a misspeculated task to be committed and an error in a live-out set would incorrectly update the architected state.

Transport of Live-ins and Live-outs. The live-in and live-out sets must be transported correctly from the slave processor to the L2 cache banks and the global register file. This data movement can be accomplished using the techniques from cache coherence (*e.g.*, ECC). In addition, a slave processor must ensure that all live-ins and live-outs have been sent before a task is committed.

Capture and Transport of Transition registers. Transition registers, much like memory live-in values, must be correctly captured and transported to the global register file. Failure to do so could result in a misspeculated task being committed.

Transport of Non-speculative Data. Like a traditional architecture, tasks executing in non-speculative mode (*e.g.*, after a recovery) rely on (architected) memory data being correctly moved through the memory hierarchy.

Coherently Matching to Architected State. Live-ins buffered at L2 cache banks and the global register file must be correctly verified against the most recent value previous to the task being verified. This requirement is made difficult because live-outs from tasks can arrive out of program order. It is further complicated on the memory side by coherence requests from other processors. Failure to correctly match state could allow misspeculated tasks to commit.

Two-Phase Commit. The physically distinct L2 cache banks must all agree simultaneously that a task has been verified and can be committed. Failure to maintain this simultaneity can cause memory ordering violations in some memory models.

Tagging and Evicting Stale Data. Private L1 caches are used to hold non-architected data (e.g., data from predicted checkpoints). MSSP tags the non-architected data as such, so that it can be evicted before a non-speculative task executes on the processor. If non-architected data is incorrectly tagged or evicted, a non-speculative task, which will not have its live-ins verified, can potentially read an incorrect value that can result in the architected state being updated incorrectly.

Checkpointing Hardware. Although it is not necessary that the checkpoints be correct, it is important that the master processor is not allowed to update architected state in any way. Updates by the master can only modify its local cache and write entries of the checkpoint buffers.

6.2.2 Mechanisms That Don't Need To Be Correct

Program Distillation. If the above mechanisms are correct, the MSSP paradigm can be made impervious to faults in the distilled program. The master is prevented from directly affecting architected state, so the only faults it can induce are indirectly through faulty checkpoint values. By verifying all live-in values against architected state, such faults can be detected. The MSSP paradigm includes a set of watchdog timers to detect if the master enters an infinite loop and force a recovery to ensure forward progress.

Mapping. Mapping is only performed in four situations, none of which are during the slave's execution of the task in the original program. They are: 1) starting the master at an entry, 2) link operations performed by the master, 3) indirect branches performed by the master, and 4) the verify instruction executed by the slave to compute the PC of the first instruction of the task in the original program. All of these will only result in the master processor getting off track or bad values being part of checkpoints (as a transition register in the case of #4) that will be verified as live-ins if used.

Checkpoint Assembly. Checkpoint assembly need not be verified because the faults it introduces are equivalent to the distilled program computing bad checkpoint values in the first place.

Refresh. MSSP periodically evicts or re-fetches cache blocks that contain non-architectural data in the caches of the master and speculative slaves. Refreshes are purely a performance optimization; executing with stale data will only result in generation or consumption of incorrect checkpoints by the master and slaves, respectively.

Stop Bits. If the indications the slave processors use to decide where to stop executing a task are faulty, the task may end at an unexpected location. Bad stop annotations should not affect the ability of the current task to commit. If the following task was not specified to start where this task ended it will be detected when verifying the live-in PC or other live-in values.

6.3 Open Questions

In this section, I outline some of the open questions that remain in the study of the MSSP paradigm.

To what extent can programs be distilled? I have seen opportunities for if-conversion[1], additional register allocation, pre-fetch insertion, scheduling (both local and global), phase-based optimization, full unrolling of loops with few iterations (predicating the final iterations if necessary), more general constant folding, constant propagation, redundancy elimination, tail duplication, and others. How effective can these additional optimizations be?

What does the continuum of distilled programs look like? What is the relationship between performance when the code is correct and its accuracy? Which optimizations are most important? Which optimizations enable other optimizations?

How should distillation be performed? Can the run-time implementation proposed in this dissertation be implemented efficiently? Are some optimizations not worth the time they take to perform? What hardware support should the implementation include for profiling and distillation?

How can the original program be constructed to facilitate distillation and the MSSP paradigm? In this work, I focused on a binary-compatible approach. Can altering the original code improve the extent to which it can be distilled? Can such alterations simplify the hardware? In addition, an alternative paradigm

---

1. If support for predication is not available, conditional moves (CMOV) can be used. Stores in the distilled program can also be made conditional by using a CMOV to conditionally set the store's address to some garbage location (*e.g.*, the address 0) where the value will not be read. Like all stores performed by the master, the store will be deallocated when the slave task is committed and not affect architected state in any way.

where the following execution is not the original program, but the minimal subset of the program necessary to check the original program may be possible.

What improvements can be made to the MSSP implementation? In our evaluation the same processor core was used for the master and the slaves. How could the master processor be tailored to facilitate execution of distilled programs. Can slave processors be simplified (or dynamically reconfigured) to fulfill their role while using less power?

Can the architecture scale to larger (16-64) processor counts? Can a hierarchy of distilled programs be constructed that provide perhaps less accurate, but more distant predictions about execution. Can techniques like pre-execution be incorporated into the paradigm to further increase the master's execution rate. Does non-architectural state need to be organized and communicated differently to support such large processor counts?

6.4 Chapter Summary

In this dissertation, I have described Master-Slave Speculative Parallelization, an execution paradigm that parallelizes a sequential program's execution using an approximate copy of the program. This chapter describes the lessons learned, describes the correctness requirement of the paradigm, and outlines some questions left open by this work. To summarize, in studying this execution paradigm, I have observed many opportunities to effectively approximate programs. Of these, I've perhaps implemented half. These optimizations are an encouraging start, but to this point, during the implementation of each optimization, I discover another one or two possible optimizations. This dissertation demonstrates that this paradigm has a number of nice characteristics—not the least of which is the relaxed constraints on correctness of a number of mechanisms—but only begins the characterization of distilled programs and the performance potential of MSSP. There is much potential future work to explore the extent to which programs can be approximated, how the approximation should be performed, and how architectures should be organized to efficiently execute programs with the assistance of approximate code.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, November 1998.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[4] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.

[5] V. Bala, Evelyn Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, Hewlett Packard Labs, June 1999.

[6] Thomas Ball and James R. Larus. Programs Follow Paths. Technical Report MSR-TR-99-01, Microsoft, January 1999.

[7] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[8] J. L. Bentley. *Writing Efficient Programs*. Prentice Hall, Englewood Cliffs, 1982.

[9] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[10] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[11] B. Calder, G. Reinman, and D.M. Tullsen. Selective Value Prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 64–74, May 1999.

[12] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, March 1999.

[13] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Water, and Wen mei W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

[14] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: a Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, Mar 1998.

[15] Marcelo Cintra, Jose Martinez, and Josep Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[16] Lucian Codrescu, D. Scott Wills, and James Meindl. Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications. *IEEE Transactions on Computers*, 50(1):67–82, 2001.

[17] Robert Cohn, David Goodwin, and P. Geoffrey Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[18] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[19] R. P. Colwell, R. P. Nix, J. J. O' Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[20] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, June 1995.

[21] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[22] Richard Eickemeyer and Stamatis Vassiliadis. A Load-Instruction Unit for Pipelined Processors. *IBM Journal of Research and Development*, 37(4):547–564, July 1993.

[23] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2001.

[24] Joseph A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[25] Manoj Franklin and Gurindar S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 1992.

[26] Manoj Franklin and Gurindar S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[27] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–271, October 1998.

[28] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[29] Lal George and Andrew W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[30] J. González and A. González. The Potential of Data Value Speculation to Boost ILP. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 21–27, July 1998.

[31] James R. Goodman and Wei-Chung Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, July 1988.

[32] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[33] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[34] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control Flow Speculation in Multiscalar Processors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, February 1997.

[35] V. Kathail, Micheal Schlansker, and Bob R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, HP Laboratories, February 1994.

[36] Gregory A. Kemp and Manoj Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proceedings of the International Conference on Parallel Processing*, pages 239–246, August 1996.

[37] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[38] Alexander Klaiber. The technology behind crusoe processors. Transmeta Whitepaper, January 2000.

[39] Tom Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[40] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial Dead Code Elimination. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, June 1994.

[41] Ashok Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2):27–32, March/April 1997.

[42] Eric Larson and Todd Austin. Compiler Controlled Value Prediction using Branch Predictor Based Confidence. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[43] Kevin M. Lepak and Mikko H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, June 2000.

[44] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–237, December 1996.

[45] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[46] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), November 1993.

[47] Pedro Marcuello, Antonio Gonzalez, and Jordi Tubella. Speculative Multithreaded Processors. In *Proceedings of the 1998 International Conference on Supercomputing*, July 1998.

[48] Doug Matzke. Will physical scalability sabotage performance gains. *IEEE Computer*, 30(9):37–39, September 1997.

[49] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, and Wen mei W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 59–70, June 2000.

[50] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic Points-To Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, June 2001.

[51] Andreas Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[52] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the 2001 International Conference on Supercomputing*, June 2001.

[53] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, CA, 1997.

[54] Robert Muth, Saumya Debray, and Scott Watterson Koen De Bosschere. alto : A Link-Time Optimizer for the Compaq Alpha . *IEEE Transactions on Computers*, 31(1):67–101, 2001.

[55] Soner Onder and Rajiv Gupta. Dynamic Memory Disambiguation in the Presence of Out-of-order Store Issuing. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 170–176, November 1999.

[56] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[57] Subbarao Palacharla and James E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[58] Sanjay J. Patel and Steven S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[59] K. Pettis and D. Hansen. Profile guided code positioning. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.

[60] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[61] Eric Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 1999.

[62] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–35, December 1996.

[63] Eric Rotenberg, Quinn Jacobson, Y. Sazeides, and James E. Smith. Trace Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 138–148, December 1997.

[64] A. Roth and G.S. Sohi. Speculative Data Driven Sequencing for Imperative Programs. Technical Report CS-TR-2000-1411, University of Wisconsin, Madison, February 2000.

[65] Amir Roth. *Pre-Execution via Speculative Data-Driven Multithreading*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 2001.

[66] Amir Roth and Gurindar Sohi. Speculative Data-Driven Multi-Threading. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 37–48, January 2001.

[67] Y. Sazeides and James E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–258, December 1997.

[68] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[69] James E. Smith. Decoupled Access/Execute Computer Architecture. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 112–119, April 1982.

[70] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, May 1990.

[71] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[72] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[73] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, Carnegie Mellon University, June 1997.

[74] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[75] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, January 2000.

[76] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[77] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[78] D.M. Tullsen and J.S. Seng. Storageless Value Prediction Using Prior Register Values. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 270–281, May 1999.

[79] S. Vajapeyam and T. Mitra. Dynamic Vectorization: A Mechanism for Exploiting Far-flung ILP in Ordinary Programs. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 12–27, May 1999.

[80] Sriram Vajapeyam and Tulika Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, June 1997.

[81] T. N. Vijaykumar and Gurindar S. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, November 1998.

[82] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 281–290, December 1997.

[83] Catherine Xiaolan Zhang, Zheng Wang, Nicholas C. Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automated Profiling and Optimization. In *Proc. 16th Symposium on Operating System Principles*, pages 15–26, October 1997.

[84] Craig B. Zilles and Gurindar S. Sohi. Understanding the Backwards Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, June 2000.

[85] Craig B. Zilles and Gurindar S. Sohi. Execution-based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.