SELF-MONITORING OF THREAD INTERACTIONS FOR IMPROVED RESOURCE MANAGEMENT IN MULTITHREADED SYSTEMS

by

Tong Li

Department of Computer Science
Duke University

Date: _____

Approved:

_____

Professor Alvin R. Lebeck, Co-Advisor

_____

Professor Daniel J. Sorin, Co-Advisor

_____

Professor Pankaj Kumar Agarwal

_____

Professor Jeffrey S. Chase

_____

Professor Carla Schlatter Ellis

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2005

ABSTRACT

SELF-MONITORING OF THREAD INTERACTIONS FOR IMPROVED RESOURCE MANAGEMENT IN MULTITHREADED SYSTEMS

by

Tong Li

Department of Computer Science
Duke University

Date: _____

Approved:

_____

Professor Alvin R. Lebeck, Co-Advisor

_____

Professor Daniel J. Sorin, Co-Advisor

_____

Professor Pankaj Kumar Agarwal

_____

Professor Jeffrey S. Chase

_____

Professor Carla Schlatter Ellis

An abstract of dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2005

Abstract

Hardware-level multithreaded systems provide infrastructure capable of executing multiple software threads in parallel. Such systems range from conventional multiprocessors to the recent simultaneous multithreading (SMT) and chip-multiprocessor (CMP) systems. To ensure that application programs receive high "quality-of-service" (e.g., high throughput, low response time), it is important for multithreaded systems to manage their hardware resources efficiently.

Existing resource management solutions in operating systems and hardware are far from optimal. The major obstacle they face is the lack of accurate knowledge in the OS and hardware about application runtime behavior. For example, existing processors try their best to maximize metric values (e.g., instructions-per-cycle) meaningful to them, which, however, often do not correlate with metrics in which the applications are interested (e.g., transactions-per-second for a database application). Such lack of knowledge about application behavior often leads to inefficient resource usage in multithreaded systems.

In this thesis, we develop mechanisms that enable multithreaded systems to self-monitor application runtime behavior. To support efficient management of processor resources (e.g., functional units), we study monitoring of thread microexecutions and their correlations with application-level performance metrics. To support efficient management of system resources (e.g., entire processors), we study monitoring of thread interactions. We focus on monitoring thread synchronization behaviors, which reflect application performance, and thread deadlocks, which reveal errors in applications.

We make three contributions in this thesis. First, we develop a directed acyclic graph (DAG) model for quantifying the criticality of instructions on shared memory multithreaded systems. Second, we design a hardware mechanism that dynamically detects spinning synchronizations in multithreaded programs. Based on this design, we develop (1) hardware performance counters that more accurately reflect application performance than existing counters, (2) thread scheduling and power control policies that more efficiently

manage resources than existing policies, and (3) OS and hardware support for detecting application livelocks. The third contribution of this thesis is an operating system mechanism, called Pulse, which can dynamically detect deadlocks in user applications. By using OS-level speculative execution, Pulse can detect deadlocks involving consumable resources, such as synchronization semaphores and pipes, which, to the best of our knowledge, no existing tools can detect.

Acknowledgments

I am fortunate to have two great advisors: Alvin Lebeck and Daniel Sorin. Working with Alvy and Dan has been both pleasant and rewarding. I am grateful for their tremendous help in improving my ability to perform and present research. For all the skills I learned from Alvy and Dan, they will continue to benefit me in my future career and throughout my life.

I sincerely appreciate Carla Ellis for her guidance in the deadlock detection work that I present in this dissertation. Her insights, knowledge, and vision were essential to make this work successful. I thank Pankaj Agarwal and Jeffrey Chase for serving on my committee. Their comments and suggestions have greatly helped improve the quality of this dissertation. I also thank Jeff for his guidance in the Direct Access File System (DAFS) project. Although not included in this dissertation, the DAFS work was an eye-opener and had helped me sharpen my skills in systems programming, which were essential to the successful completion of this dissertation.

I would like to thank Judy Goldsmith, my former advisor at the University of Kentucky, where I obtained my Master's degree. I am grateful that I met Judy when I first came to the US and we continue to be close friends. I sincerely appreciate all the skills I learned from Judy on how to become a good researcher.

I can never say enough thanks to my wife, Nan. Throughout my Ph.D. study, she has been the strongest support for me to overcome the times of difficulty and frustration. I thank my parents, who continuously encouraged me in my academic pursuit.

Table of Contents

List of Tables

List of Figures

1 Introduction

Hardware-level multithreaded computers have been increasingly prevalent in recent years. Multithreaded systems provide hardware execution contexts that support multiple software threads to execute in parallel. These systems range from conventional multiprocessors (e.g., Sun Fire V1280), to the recent simultaneous multithreading (e.g., Intel Xeon) and single-chip multiprocessor (e.g., IBM POWER4) systems. Since multithreaded systems possess large computing power, they often provide infrastructure for important server applications, such as database and web servers. Not only at the high end, hardware-level multithreading has also gained popularity in low-end desktop computers such as those based on Intel's Pentium 4 processors. Due to technology limitations, processor clock speed improvements will soon be unable to sustain the same pace as what we have experienced in the past decade. Thus, hardware-level multithreading will continue to proliferate as parallelism becomes the alternative path for improved performance.

A typical multithreaded system consists of a large variety of hardware resources, ranging from system-level resources, such as processors and memories, to chip-level resources, such as functional units and issue queues. It is important to manage all these resources such that they work cooperatively to provide user applications the best "quality-of-service".

Resource management in multithreaded systems, however, is a complex task. First, different applications or even different phases in the same application often have different metrics for measuring quality-of-service. For example, a database application may be interested in the number of transactions it can process per second, while a windowing program may be concerned about the response time of a mouse click. Dynamically managing system resources to accommodate varying application needs is challenging. Second, application-level metrics, such as transactions-per-second, are often not attainable in the operating system and hardware. For example, the OS and hardware have no idea about what a transaction means for a database application. Existing OSes and processors try their best to max-

imize metric values meaningful to them, which, however, often do not correlate with application-level metrics. As an example, existing processors strive to execute every instruction as fast as possible. This is often unnecessary because some instructions, such as those executed in a spin loop, can be delayed without impacting application throughput. Similarly, at the OS level, execution of threads that are not on the critical path of a multithreaded program could also be delayed without increasing the program total runtime.

Existing resource management solutions in the OS and hardware are far from optimal. The major obstacle faced by designers is the lack of knowledge in the OS and hardware about how applications perform in terms of their specific metrics. To obtain such knowledge, applications could explicitly communicate it to the OS and hardware, or the OS and hardware could reason about it by monitoring aspects of application runtime behavior. This thesis takes the latter approach, because it releases the burden on application programmers and facilitates resource management policies that are general for all applications.

In this thesis, we develop mechanisms that enable the OS and hardware to self-monitor application runtime behavior. To support efficient management of processor resources, such as on-chip functional units, we study monitoring of thread executions at the microarchitecture level, i.e., microexecutions, and their correlations with application-level performance metrics. To support efficient management of system-level resources, such as entire processors, we study monitoring of thread interactions. We focus on monitoring thread synchronization behaviors, which reflect application performance, and thread deadlocks, which reveal errors in applications.

In Chapter 2, we study monitoring of thread microexecutions. Fields et al. [15, 16] proposed a directed acyclic graph (DAG) model for characterizing the microexecution of single-threaded programs on uniprocessors. Based on this model, they quantified the *criticality* of an instruction using its *global slack*, i.e., the maximum time that the instruction can be delayed without increasing the program's total runtime. Previous research [15, 16, 53, 58, 59] showed that allocating resources to instructions based on criticality enabled efficient resource usage in single-threaded uniprocessor systems. In this thesis, we extend the uniprocessor criticality model to multithreaded systems. We evaluate criticality for var-

ious multithreaded commercial and scientific workloads. Our results show that instructions in the same application often have a diverse range of criticality values, and some instructions have hundreds or even thousands of cycles of slack, suggesting large potential benefits of criticality-based resource management polices for multithreaded systems.

In Chapter 3, we study monitoring of thread synchronization behaviors. We focus on spinning (a.k.a., busy-waiting) synchronization. When a thread is spinning, it wastes resources, such as processors and power, while performing no useful work. We design efficient hardware that dynamically detects when a thread is spinning. Based on the proposed hardware, we develop techniques that enable accurate performance monitoring, efficient thread scheduling and power management, and support for detecting application livelocks.

In Chapter 4, we study monitoring of deadlocks in multithreaded applications. Deadlock is a serious error, which, at its extreme, can cause an entire system to stop functioning. Timely detection of deadlock is essential for resolving the error and maintaining forward progress. We develop Pulse, an OS mechanism that dynamically detects deadlocks in user applications. Pulse speculatively unblocks each process that has been blocked for a long time, and executes it ahead. Speculative execution allows Pulse to identify what events a blocked process would generate if it were not blocked. By comparing against the events on which each process is blocked, Pulse can discover the dependences among the processes and events; if the dependences form a circular chain, Pulse reports a potential deadlock. The ability to look into the future allows Pulse to detect deadlocks that involve complex (non-lock-like) resources, which, to the best of our knowledge, no existing tools can detect.

This thesis makes three contributions.

1. We develop a DAG model that quantifies instruction criticality for shared memory multithreaded systems.

2. We propose a hardware mechanism that dynamically detects thread spinning synchronization and supports improved performance monitoring and resource management.

3. We design Pulse, a generic OS framework that dynamically detects deadlocks in user applications.

2 Monitoring Thread Microexecutions

Modern processors consist of various hardware structures, such as functional units, reorder buffers, and issue windows. Traditionally, processors allocate these resources to instructions based on simple priority functions (e.g., oldest-instruction-first). When making resource allocation decisions, these policies do not take into account how the decisions would affect applications' overall performance. Consequently, applications may not always achieve their desired levels of performance; even when they do, resource utilization may not always be efficient (e.g., power consumption may be higher than necessary).

To efficiently manage on-chip resources, processors must be able to monitor application performance. From a processor's perspective, an application thread is simply a series of dynamic instructions executing on the hardware. We refer to the execution of a thread within a processor as the thread's *microexecution*. To accurately monitor application performance, hardware must determine how individual instructions in the application threads contribute to overall performance. *Instruction criticality* is a term that refers to the importance of an instruction with respect to overall application performance. Previous research [15, 16, 49, 53, 58, 59] has shown that criticality-based resource management policies can greatly improve resource efficiency for single-threaded uniprocessor systems.

Fields et al. [15, 16] proposed a directed acyclic graph (DAG) model for characterizing the microexecution of a single-threaded program. With this model, they can perform critical path analysis and quantify instruction criticality by using *global slack*, which is the maximum amount of time that the hardware can delay an instruction without increasing the program's total runtime. While prior research has demonstrated that instruction criticality is an effective metric for single-threaded uniprocessors, this

thesis is the first research to extend the uniprocessor criticality model and analysis to shared memory multithreaded systems.

We make three contributions in this chapter:

1. We develop the first fine-grain instruction-level criticality model for shared memory multithreaded systems.
2. We design novel graph compression techniques that significantly reduce the storage and time requirements for analyzing traces of complex multithreaded workloads.
3. We perform thorough evaluation that demonstrates instructions in multithreaded commercial and scientific applications have diverse ranges of criticality values.

The remainder of this chapter is organized as follows. In Section 2.1, we discuss in detail the motivation for modeling instruction criticality. To quantify the criticality of instructions, we present a DAG model for characterizing program executions in Section 2.2. In Section 2.3, we show how to map this model to programs running on single-threaded uniprocessors and multithreaded systems. In Section 2.4, we study how to compute instruction criticality under the DAG model. Since a DAG representing the entire execution of a program can be very large, in Section 2.5, we develop graph compression techniques that enable efficient criticality analysis. We evaluate instruction criticality for commercial and scientific workloads on shared memory multithreaded systems in Section 2.6 and discuss related work in Section 2.7. Finally, we conclude in Section 2.8.

2.1 Motivation for Modeling Criticality

Computer hardware frequently makes decisions about how to manage its resources. For example, dynamically scheduled (a.k.a., out-of-order) superscalar microprocessors must decide how to schedule machine instructions and how to share resources (e.g., functional units or fetch bandwidth) among them. Conventional microprocessors employ microarchitectural control policies (e.g., instruction issue policy) based on simple priority functions (e.g., oldest-instruction-first). These simple policies often lead to inefficient resource utili-

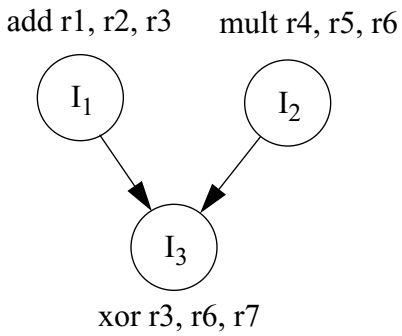


Figure 2-1. Dependence graph for an example program segment. Rightmost operands are destination registers.

zation because they do not take into account the criticality of instructions with respect to overall program performance.

In any program, single-threaded or multithreaded, it is often the case that certain instructions can be delayed without impacting the program's total runtime. Figure 2-1 shows a dependence graph of three instructions from an example program. In this figure, instruction I_3 depends on the results (r3 and r6) of instructions I_1 and I_2. Instruction I_3 can execute only if both I_1 and I_2 are finished. Suppose that I_2 finishes *t* cycles later than I_1. Thus, the execution of I_1 can be delayed for *t* cycles without delaying the execution of I_3. On the other hand, I_2 cannot be delayed because it directly impacts when I_3 can execute.

The above example demonstrates that, due to the dependences and latency differences between instructions, some instructions can have certain amounts of *slack* in their execution. The slack of an instruction provides a natural measure for instruction criticality: the greater an instruction's slack is, the less critical it is.

Determining the criticality (slack) of instructions as soon as they enter the processor pipeline allows hardware to manage resources based on instruction criticality. The following list shows potential advantages of criticality-based control policies for both single-threaded uniprocessor and multithreaded systems.

- *Resource utilization*. Resources (e.g., caches, memory bandwidth) can obtain better efficiency by prioritizing allocations and accesses based on instruction criticality.

- *Power efficiency.* Processors or thread contexts executing less critical instructions can run more slowly, thus saving power without sacrificing performance.
- *Mis-speculation reduction*. Speculation techniques, such as coherence prediction, can improve application performance. However, mis-speculations introduce performance penalty. Selectively applying speculation to only critical instructions avoids unnecessary speculations and thus can reduce the number of mis-speculations.
- *Dynamic scheduling*. Multiprocessor systems typically perform dynamic scheduling at the task level to obtain an efficient schedule. By incorporating the fine-grain instruction criticality information, the scheduling algorithm could achieve a better schedule.

In the next section, we present a model for characterizing the microexecution of a program. This model serves as a vehicle for us later in this chapter to formally define instruction criticality and design algorithms to compute it.

2.2 A DAG Model for Program Execution

We model any given execution of a program with a directed acyclic graph (DAG), in which each node represents a dynamic event during the execution and each edge represents a dependence between its source and target nodes. In our model, a dynamic event is a general term that can represent any event in the program's execution (e.g., fetching an instruction, executing an instruction, or executing a coarse-grain task). A dependence edge models the precedence constraints dictated by the program's semantics (e.g., data dependence) or the underlying hardware (e.g., resource dependence). Each edge is weighted by the time required to resolve the corresponding dependence during the execution. For an edge $e = (u, v)$, we say that e arrives at node v with arrival time t, where t is the time that the corresponding dependence is resolved. The arrival time of an edge is the real time during program execution (i.e., arrival times increase monotonically as the execution progresses). Among all edges arriving at v, we call the one that arrives last the *last-arriving* edge [16]. There can be more than one last-arriving edge when multiple edges arrive simultaneously.

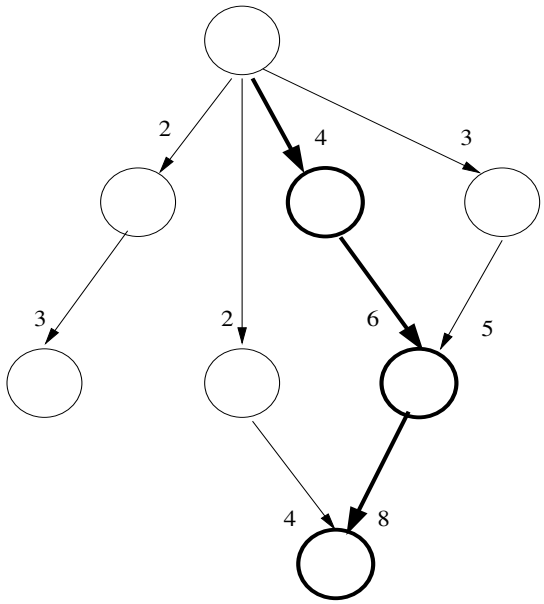


Figure 2-2. A simple DAG with critical path highlighted.

A *critical path* of a DAG is a longest weighted path in the DAG. The length of a critical path is the sum of the weights of all edges on the path, which is equal to the total time of the given execution. If an edge $e = (u, v)$ is on a DAG's critical path, an important property is that e must be a last-arriving edge sinking on node v. Conversely, if an edge is not last-arriving, it must not be on a critical path. When a node has multiple last-arriving edges, the DAG may possibly contain multiple critical paths.[1] For the purposes of our criticality analysis, we do not explicitly label each edge with its weight; instead, we label each edge with the time that it arrives at its target node. Figure 2-2 shows a simple DAG with its critical path highlighted. Note that a DAG need not have a unique sink node (i.e., a node that has no outgoing edges). For example, in a multithreaded program, every thread may invoke an exit call at the end of its execution. If we model these exit calls as events and represent them as nodes in a DAG, all of these nodes do not have outgoing edges.

The *slack* of an event is a measure of how long its start time can be delayed without affecting subsequent events. To distinguish types of slack, Fields et al. [15] introduce the

1. For brevity of terminology, we will refer without loss of generality to the *critical path* as encompassing these potentially multiple critical paths.

concepts of *local* and *global* slack. In our DAG model, the local slack of a node is the maximum time that the start of the corresponding event can be delayed without delaying any event in the descendent nodes.[2] The global slack of a node is the maximum time that the start of the event can be delayed without increasing the length of the DAG's critical path. By definition, instructions on the critical path have both local and global slack of zero.

2.3 Mapping DAGs to Systems

In this section, we first discuss how previous research maps the DAG model to single-threaded uniprocessor systems and then describe our contribution of extending the uniprocessor DAG model to shared memory multithreaded systems.

2.3.1 Uniprocessor Systems

Fields et al. [15, 16] use the DAG model to characterize any given execution of a single-threaded program on a dynamically scheduled superscalar processor. In their DAG model, a node represents one of three events: instruction dispatch, execute, and commit, each corresponding to a stage in an instruction's microexecution. An edge represents one of seven types of dependence.

1. In-order dispatch: Instructions must be dispatched in the order in which they appear in the program.
2. Finite reorder buffer: When the reorder buffer is full, an instruction can be dispatched to it only after another instruction commits and frees an entry from the reorder buffer.
3. Control dependence: The correct target instruction of a mis-predicted branch cannot be dispatched until the branch is resolved.
4. Execution follows dispatch: An instruction cannot execute until it has been dispatched.
5. Data dependence: An instruction cannot execute until the instructions producing its operands finish.

2. Node u is a descendent of node v if there exists a path from v to u.

6. Commit follows execution: An instruction cannot commit until it has finished execution.

7. In-order commit: Instructions must be committed in the order in which they appear in the program.

Types (1) and (7) collectively model the program order dependence required by the program's sequential semantics. The other dependence types reflect microarchitectural constraints that exist in most dynamically scheduled processors.

2.3.2 Multithreaded Systems

We extend the uniprocessor DAG model to shared memory multithreaded systems. To avoid confusion between *threads* (software constructs) and *thread contexts* (hardware for running threads in some systems), throughout this thesis, we refer to the latter as processors without loss of generality. We also use the terms *threads* and *processes* interchangeably to refer to the basic units of scheduling in the OS.

For a given execution of a multithreaded program, we first construct a uniprocessor DAG for each processor in the system. Then we add dependence edges between them to model inter-processor communications. Unlike prior work, to simplify the exposition, we consider multithreaded systems with simple in-order processors (i.e., instructions dispatch, execute, and commit all in the order specified by the program). Each node in our model represents an entire instruction (i.e., we do not split it into dispatch, execute, and commit). Within each processor, we maintain the program order dependence of its dynamic instructions. The only difference that a dynamically scheduled processor model introduces is the mapping of events and dependences to nodes and edges; the criticality analysis is the same.

In our model, a program order dependence edge connects an instruction to the instruction that immediately follows it in program order. The program order dependence resolves when the following instruction is issued to a functional unit for execution. Therefore, by the definition of the DAG model, we label a program order dependence edge with the issue time of the following instruction.

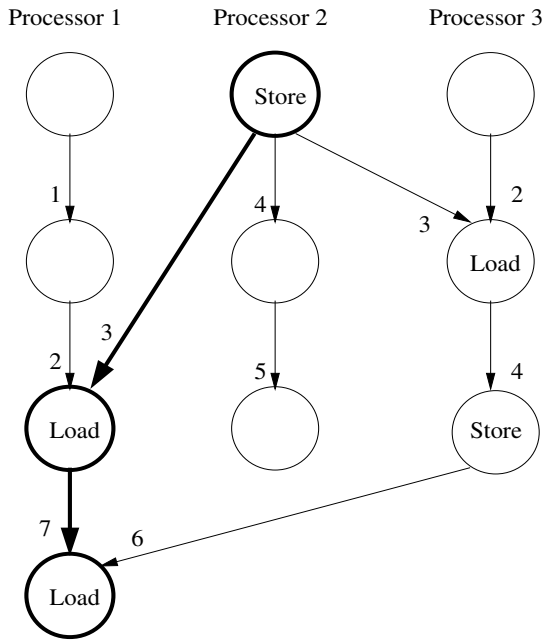


Figure 2-3. A multiprocessor DAG (critical path highlighted).

In a shared memory multithreaded system, processors communicate with each other only via loads and stores to shared memory. Between instructions on different processors, true data dependence governs their ordering for the correctness of the program. Thus the other type of dependence we model, which does not exist in the uniprocessor model, is *read-after-write* (RAW) dependence. A RAW dependence occurs between a store that produces a value and a load that consumes the value in the specific execution that the DAG represents. When the store and load are on the same processor, the RAW edge connecting them models a normal data dependence as in the uniprocessor model. When the store and load are on different processors, the RAW edge models a communication between the two processors in this particular execution. A RAW dependence resolves when the store completes. Therefore, we label a RAW edge with the store's completion time.

Since we model only two types of dependences, a node in our DAG model has at most two incoming edges: a program order edge and a RAW edge. Only load nodes have incoming RAW edges and only store nodes have outgoing RAW edges. Figure 2-3 shows an example DAG for a multithreaded system with its critical path highlighted. Unlabeled

nodes are instructions that are neither loads nor stores. Although not shown in the figure, there can exist RAW edges that connect store and load nodes on the same processor.

2.4 Computing Slack

The global slack of an instruction is the maximum time that the instruction can be delayed without increasing the program's total runtime. In Section 2.4.1, we show formal definitions of local and global slack with respect to nodes and edges in a DAG representing some specific execution. In Section 2.4.2, we present an offline algorithm for computing global slack of instructions based on traces of program execution. While our algorithm computes global slack, it can also determine the critical path of a DAG. In Section 2.4.3, we describe how we apply the algorithm differently to finite and continuous workloads on shared memory multithreaded systems. Finally, in Section 2.4.4, we discuss possible approaches to computing criticality on the fly.

2.4.1 Local and Global Slack

The *endpoint node* of a DAG for a specific execution is the last finished node (instruction) among all the nodes in the DAG. When there are multiple last finished nodes (because they finish at the same time on different processors), we choose an arbitrary node among them to be the endpoint node. To compute global slack in a DAG for a specific program execution, we use the same definitions Fields et al. [15] developed.

Definition 2-1. The *local slack of an edge* $e = (u, v)$, denoted by $L(e)$, is the time that the latency of e can be increased without delaying its target node v.

We compute $L(e)$ as the difference between the arrival time of the last-arriving edge sinking on node v and the arrival time of e. If e is last-arriving, $L(e) = 0$.

Definition 2-2.. The *local slack of a node u*, denoted by $L(u)$, is the maximum time u can be delayed without delaying any of its descendent nodes.

We compute $L(u)$ as the smallest local slack among the outgoing edges of u, i.e., $L(u) = \min_i(L(e_i))$, where e_i is the i^{th} outgoing edge of node u. In case that node u has no outgoing

edge, if u is the endpoint node, $L(u) = 0$; otherwise, $L(u)$ equals the difference between node u's finish time and the finish time of the endpoint node.

Definition 2-3. The *global slack of an edge* $e = (u, v)$, denoted by $G(e)$, is the time that the latency of e can be increased without increasing the length of the critical path.

We compute $G(e)$ as $L(e) + G(v)$, where $G(v)$ is the global slack of node v defined as follows.

Definition 2-4. The *global slack of a node* u, denoted by $G(u)$, is the maximum time that node u can be delayed without increasing the length of the critical path.

We compute $G(u)$ as the smallest global slack among the outgoing edges of u, i.e., $G(u) = \min_i(G(e_i))$, where e_i is the i^{th} outgoing edge of node u. If node u has no outgoing edge, then $G(u) = L(u)$.

Given these definitions, for a non-sink node u, we can derive $G(u)$ from the local slack of u's outgoing edges and the global slack of these edges' target nodes. Let $e_i = (u, v_i)$ be the i^{th} outgoing edge of u, and o_{ij} be the j^{th} outgoing edge of target node v_i. We have the following:

$$G(u) = \min_i(G(e_i)) = \min_i(L(e_i) + G(v_i)). \qquad \textbf{(Equation 2-1)}$$

We illustrate this computation in Figure 2-4, and we note that it lends itself to a recursive algorithm that starts at the end of execution and proceeds backwards in time. In the base case of this recursion, node u is a sink node (i.e., it has no outgoing edge), and $G(u) = L(u)$, which can computed as we showed for Definition 2-2 above.

2.4.2 Offline Algorithm for Computing Global Slack

During the execution of a program, we store information about its instructions so that we can construct a DAG for offline processing. Figure 2-5 shows our offline algorithm for computing the global slack of all nodes in a DAG. Since instructions on the critical path have global slack of zero, we can determine the critical path by backtracing the DAG from

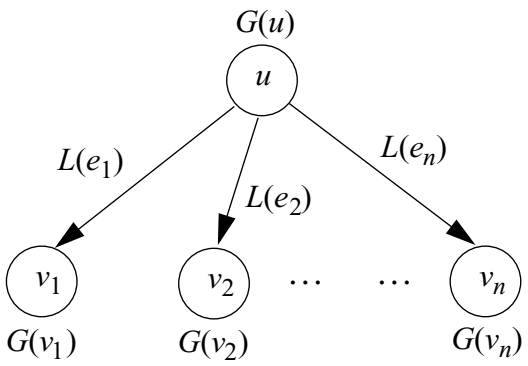


Figure 2-4. Illustration for how to compute global slack. The global slack of node *u* depends on the local slack of each outgoing edge of *u* and the global slack of each successor of *u*.

its endpoint node. On the backtrace, all nodes with global slack of zero constitute the critical path. Our algorithm does not assume any special properties of a DAG. Thus, it is applicable to DAGs for both uniprocessor (single-threaded) and multithreaded systems, as well as both in-order and out-of-order (dynamically scheduled) processors.

2.4.3 Finite vs. Continuous Workloads

A *finite* workload, such as a scientific application, executes a finite number of instructions, so we can construct a DAG for all of its dynamic instructions and the DAG contains a well-defined endpoint. However, a *continuous* workload, such as a web or database server, executes continuously and thus does not have a well-defined endpoint. Nevertheless, given a large interval of execution, we can pick the last finished node within the interval from an arbitrary processor and consider it to be the endpoint. We construct a DAG that includes only the finite number of instructions that directly or transitively lead to the endpoint node via program order and/or RAW dependence edges. If a DAG node has an outgoing dependence edge to a node outside the DAG, we do not include this edge in the DAG. Unlike DAGs for finite workloads, in which non-endpoint nodes can have no outgoing edges, the endpoint node in a continuous workload DAG is the only node that has no outgoing edges. A DAG constructed in this way allows us to apply our algorithm to a finite subset of the infinitely many instructions in a continuous workload, and it enables control policies to apply optimizations to these instructions without increasing their overall runtime. Ran-

```
if u is the endpoint node then
    G(u) = 0
else if u has no outgoing edges then
    G(u) = finish_time(endpoint) - finish_time(u)
else
    G(u) = undefined
end if
Sort all nodes in reverse topological order
for all nodes u in the sorted order do
    if G(u) = undefined then
        min_global_slack = infinity
        for all of u's outgoing edges e_i = (u, v_i) do
            compute L(e_i)
            if L(e_i) + G(v_i) < min_global_slack then
                min_global_slack = L(e_i) + G(v_i)
            end if
        end for
        G(u) = min_global_slack
    end if
end for
```

Figure 2-5. Algorithm for computing global slack of all nodes in a DAG.

domly choosing the endpoint node from all the processors affects the individual instruction slack values computed by the algorithm. Nevertheless, as shown in Section 2.6, the endpoint choice has negligible impact on the overall slack distribution among the instructions. This result suggests that we can design more sophisticated control policies based on aggregated slack values over an interval.

2.4.4 Computing Global Slack On The Fly

Equation 2-1 shows how to compute the global slack of a node. However, this computation requires knowing the entire DAG for a program. This requirement poses difficulty to com-

puting global slack on the fly, because, while a program is executing, only a partial DAG is available.

Although accurately computing global slack on the fly is difficult, several approaches are possible for estimating it. First, we can extend the history-based prediction scheme that previous research proposed for uniprocessors [15, 16]. The challenge is to extend it to multithreaded systems and determine if global slack has good locality in these systems.[3] Second, we can apply speculation techniques to look ahead in the program, and estimate criticality based on portions of the program's entire execution. The challenge is that inaccurate criticality results could lead to control decisions that lengthen the program's execution. Third, instead of computing the absolute criticality of an instruction, it is sufficient for scheduling purposes to know its relative criticality with respect to other instructions. Relative criticality can be estimated without knowledge of the entire execution. By examining the partial DAG corresponding to instructions in the current instruction window, the processor could determine if one instruction is relatively more critical than another.

Estimating criticality on the fly for all instructions is challenging. However, our results in Section 2.6.2 reveal that, when a program has frequent use of spinning synchronization, instructions executed in spin loops (i.e., spinning instructions) often possess large amounts of global slack. Spinning instructions are certainly non-critical, because delaying their execution to some extent does not affect program performance.

In the rest of this chapter, we focus on offline criticality analysis and evaluation to show how important criticality is. In Chapter 3, we focus on the special class of non-critical instructions: spinning instructions, and study how to dynamically detect them to enable efficient resource management.

3. Global slack has good locality if two consecutive dynamic instances of a static instruction have global slack values that are approximately the same.

2.5 Graph Compression

Our algorithm in Section 2.4.2 assumes that we have enough information to construct the DAG for a given program's execution. In our model, a DAG contains nodes representing all dynamic instructions, which can be a large amount of information to store. For example, the sizes of our workloads running with eight processors are all on the order of billions of instructions. Though not infeasible, it is certainly non-scalable and time-consuming to store and offline-process such a large amount of information. To ease this problem, we developed techniques that can compress a DAG to a much smaller one without changing its critical path and slack properties. The compressions exploit the observation that many nodes and edges are not "useful" for computing the critical path of the DAG or the global slack of any node in the DAG, and thus we can safely remove them from the DAG. There are three cases in which we can perform a compression. We describe them as three theorems and show proofs for their correctness. A compression that transforms DAG G to G' is considered *correct* if we can derive from G' the critical path of G and the global slack of any node in G.

Theorem 2-1. Let v_0 and v_k be two nodes on the same processor. Let $p = (v_0, v_1, \ldots, v_{k-1}, v_k)$ be the path connecting them via only program order edges. If all the intermediate nodes on p (i.e., nodes $v_1, \ldots, v_{k-1}$) have neither incoming nor outgoing RAW edges, then we can compress the DAG by removing all the intermediate nodes and connecting v_0 and v_k directly with an edge labeled with the same arrival time as that of the original program order edge sinking on v_k.

Proof. Figure 2-5 shows the situation described by the theorem. Let v_i and v_j be two arbitrary intermediate nodes on the path p, and let e be the program order edge connecting them. Since e is the only outgoing edge from v_i, by Equation 2-1, $G(v_i) = L(e) + G(v_j)$. Since e is the only edge sinking on v_j, by definition, $L(e) = 0$. Therefore, $G(v_i) = G(v_j)$. Let e' be the program order edge connecting nodes v_{k-1} and v_k. Inductively, we have

$$G(v_1) = G(v_2) = \ldots = G(v_{k-1}) = L(e') + G(v_k).$$

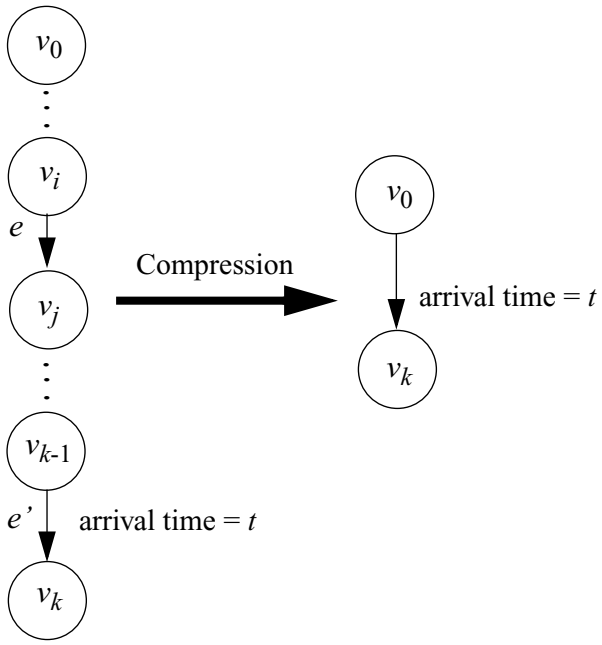


Figure 2-5. Illustration for Theorem 2-1.

Since we compute $L(e')$ by using the arrival times of the edges sinking on v_k, and we compute $G(v_k)$ by using v_k's outgoing edges and descendent nodes, which are all retained in the compressed DAG, we can correctly derive the global slack of nodes v_1, …, v_{k-1} if they are removed.[4]

We can also derive the critical path from the compressed DAG. If the removed nodes are on the critical path of the original DAG, then v_k must be on the critical path and *e'* must be a last-arriving edge to v_k. Since the arrival time of *e'* is retained in the compressed graph, we must find that both v_0 and v_k are on the critical path in the compressed DAG, and from this we can derive that all the removed intermediate nodes must also be on the critical path in the original DAG. Conversely, if the removed nodes are not on the critical path of the original DAG, then we know either v_k is not on the critical path of the original DAG, or, if it is, *e'* must not be a last-arriving edge to v_k. For either case, we must not find in the compressed DAG that both v_0 and v_k appear on the critical path at the same time. Therefore, we can derive that the removed nodes are not on the critical path of the original DAG. ❑

4. Depending on how the slack information is to be used, we may optionally store in v_k the number of nodes that are removed so we can accurately derive how many nodes were present in the original DAG.

Since only load and store nodes have RAW edges, by Theorem 2-1, we can remove all the other nodes without changing the critical path and slack information. We retain, however, all the nodes that do not have outgoing edges since they are part of the base case of our algorithm.[5] Other than these nodes, with the compression, a DAG now contains only load and store nodes and the edges connecting them.

Definition 2-5. Given a DAG G, a RAW edge $e = (u, v)$ is *dominated* if removing it does not change the critical path of G and the global slack of any node in G.

The following compression theorems are based on the observation that many RAW edges are dominated and thus can be removed from the DAG. The following lemma states the conditions for a RAW edge to be dominated.

Lemma 2-1. Given a DAG G, a RAW edge $e = (u, v)$ is dominated if it does not contribute to the computation of *any* of the following values:

(i) the critical path of G,
(ii) the global slack of node u, and
(iii) the local slack of the program order edge into node v.

Proof. Since the RAW edge e does not contribute to the computation of the critical path of G, removing it from G does not change the critical path.

Let $e' = (w, v)$ be the program order edge sinking on v. By definition, the RAW edge e is only useful for computing the global slack of u and w, and e affects $G(w)$ only if it contributes to the computation of $L(e')$. Given the conditions in the lemma, removing e does not change the global slack of any node in the DAG. ❑

Given Lemma 2-1, we can prove two sufficient conditions for a RAW edge to be dominated. The first condition considers the case in which a store has two consuming loads that are performed by the same processor.

Theorem 2-2. A RAW edge $e = (u, v)$ is dominated if there exists another edge $e' = (u, w)$ such that

5. In a continuous workload, the chosen endpoint node is the only node that does not have outgoing edges.

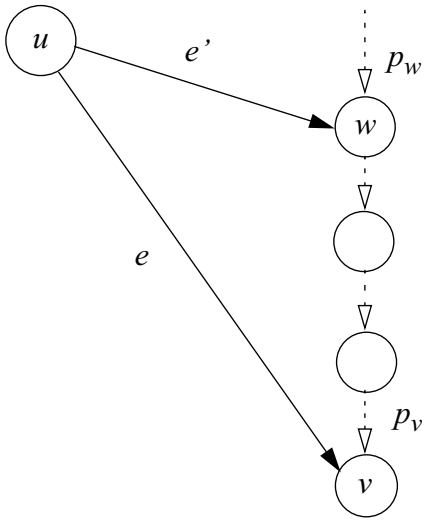


Figure 2-6. Illustration for Theorem 2-2.

(i) nodes w and v are on the same processor;

(ii) node w appears before v in program order;

(iii) the arrival time of e is less than that of the program order edge sinking on v; and

(iv) no node between w and v is the target of a RAW edge whose arrival time is greater than that of the program order edge sinking on that node.

Proof. Figure 2-6 shows the situation described by the theorem. Dashed lines are program order edges and solid lines are RAW edges. Let p_w and p_v denote the program order edges sinking on w and v, respectively.

To show that edge e is dominated, we prove that e does not contribute to the computation of any of the three values listed in Lemma 2-1.

First, by condition (iii), p_v is the last-arriving edge to the node v. Therefore edge e must not be on the critical path, i.e., removing e does not affect the computation of the critical path.

Second, since p_v is the last-arriving edge to v, by definition, the local slack of p_v, $L(p_v)$, is zero. Thus, removing e does not affect the computation of $L(p_v)$.

Third, we prove that e does not contribute to the computation of the global slack of node u. Consider the path from w to v. Let $n_1, \ldots, n_k$ be the intermediate nodes on the path. From

condition (iv), we know that for every intermediate node, its incoming program order edge is the last-arriving edge, which has local slack zero. Therefore, we have $G(w) \leq G(n_1) \leq \ldots \leq G(n_k) \leq G(v)$.

Now consider the global slack of u. We have $G(u) = \min(L(e) + G(v), L(e') + G(w))$. Edges e and e' have the same arrival time at their corresponding target nodes (both edges are labeled with node u's completion time), and because w is earlier than v in program order, the arrival time of p_w is less than that of p_v. Thus, $L(e') < L(e)$. Since we know that $G(w) \leq G(v)$, we have $G(u) = L(e') + G(w)$. Therefore, removing e does not affect the computation of the global slack of u. ❑

The first three conditions in Theorem 2-2 describe a situation that occurs frequently in programs. A value stored by an instruction is read repeatedly by later loads on the same or different processor after the store has already finished. Thus, if the last condition in the theorem is also satisfied, we can remove all such later RAW edges (except the first one) from the DAG.

Theorem 2-2 considers the situation in which we can remove a sequence of RAW edges except the first edge in the sequence. In the following theorem, we consider a situation in which we can remove the RAW edge of the first instruction that reads from memory.

Theorem 2-3. A RAW edge $e = (u, v)$ is dominated if

(i) nodes u and v are on the same processor;
(ii) the arrival time of e is less than that of the program order edge sinking on v; and
(iii) no node between u and v is the target of a RAW edge whose arrival time is greater than that of the program order edge sinking on that node.

Proof. Figure 2-7 shows the situation described by the theorem. Conditions (i) and (ii) ensure that e does not contribute to the computation of the critical path and the local slack of the program order edge sinking on v.

Let e' be the program order edge out of u and let w be the target node of e'. From Condition (iii), we know that e' is the last-arriving edge to w. Thus, we have $L(e') = 0 \leq L(e)$. Similar to Theorem 2-2, Condition (iii) also ensures that $G(w) \leq G(v)$. Therefore, $G(u) =$

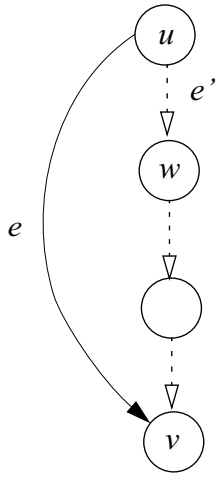


Figure 2-7. Illustration for Theorem 2-3.

$\min(L(e) + G(v), L(e') + G(w)) = L(e') + G(w)$. This proves that e does not contribute to the computation of the global slack of u. ❑

Conditions (i) and (ii) in Theorem 2-3 describe the frequent situation in which a store writes a value and has already finished by the time a later load is issued to read the value.

The three theorems described in this section allow us to compress a DAG significantly. Moreover, we can apply these theorems repeatedly. For example, after removing the RAW edges by Theorem 2-2 and Theorem 2-3, we may further compress the DAG by applying Theorem 2-1. In Section 2.6, we evaluate the effectiveness of the compressions. These theorems were derived for systems with in-order processors. For dynamically scheduled processor models, the DAGs contain more types of nodes and edges and thus require a different set of theorems. However, our methodology for deriving the theorems is still applicable.

2.6 Evaluation

In this section, we describe our simulation methodology and present detailed evaluations for various aspects of program executions under our multiprocessor DAG model.

Table 2-1. Target system parameters.

L1 Cache (I and D)	128 KB, 4-way set associative, 1-cycle hit latency
L2 Cache	4 MB, 4-way set-associative, 16-cycle hit latency
Memory	2 GB, 64 byte blocks, 320-cycle latency
Network	Totally ordered broadcast address network, unordered point-to-point data network, 6.4 GB/sec link bandwidth, 400 cycles uncontended network latency

2.6.1 Methodology

We simulate a multiprocessor target system with the Simics full-system, multiprocessor, functional simulator [38], and we extend Simics with a memory hierarchy simulator to compute execution times. Each node in our system consists of a processor, two levels of cache, some portion of the shared memory, and a network interface.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB. We use Simics/sun4u, which simulates Sun Microsystems' SPARC V9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot unmodified Solaris 8. Simics is a functional simulator only, and it assumes that each instruction takes one cycle to execute (although I/O may take longer), but it provides an interface to support detailed memory hierarchy simulation.

Processor Model. We use Simics to model a shared memory multiprocessor system in which each processor issues one instruction per cycle in-order and generates blocking requests to the cache hierarchy and beyond. The processor operates at 4 GHz; each processor cycle is equivalent to four memory cycles, and each memory cycle spans one nanosecond. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. However, we believe that our critical path analysis applies to any processor model. The only changes required for a dynamically scheduled processor model involve the mapping of events and dependences to the DAG.

Memory Model. We implement a memory hierarchy simulator that supports a MOSI broadcast snooping cache coherence protocol. The simulator captures all state transitions

Table 2-2. Wisconsin Commercial Workload Suite [1] and two SPLASH-2 benchmarks.

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are eight simulated users per processor. We warm up for 10,000 transactions, and we run for 500 transactions.
Java server: SPECjbb2000 is a server-side java benchmark that models a three-tier system with driver threads. We used Sun's HotSpot 1.4.0 Server JVM. The number of threads and warehouses is 1.5 times the number of processors. We warm up for 100,000 transactions, and we run for 50,000.
Static web server: We use Apache 2.0.39 (`www.apache.org`) for SPARC/Solaris 8, configured to use Pthread locks and minimal logging as the web server. We use SURGE to generate web requests. Our experiments use a repository of 20,000 files (totalling about 500 MB) and 10 simulated users per processor. For a system of N processors, we warm up for 100,000 $\times$ N requests and run for 5,000.
Dynamic web server: Slashcode is based on a dynamic web message posting system used by `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com`, and it contains ~3,000 messages. A multithreaded driver simulates browsing and posting behavior for three users per processor. We warm up for 240 transactions, and we run for 50 transactions.
Scientific applications: We use two benchmarks from the SPLASH-2 suite [63]. We use `barnes` with the 64K body input set and `ocean` with the 514x514 input size. For both, we measure from the start of the parallel phase to avoid measuring thread forking.

(including transient states) of our coherence protocol in the cache and memory controllers. Our memory consistency model is sequential consistency and we model the interconnection network and the contention within it. In Table 2-1, we present the design parameters of our target memory system (cycles in the latency values are processor cycles).

Workloads. We evaluate our system with the four continuous commercial applications of the Wisconsin Commercial Workload Suite [1] and two finite scientific applications. These workloads are described briefly in Table 2-2 and in more detail by Alameldeen et al. [1].

Data Collection. Our simulator assigns a sequence number to each dynamic instruction when the processor issues the instruction for execution. The sequence number is a unique, monotonically increasing integer across all instructions on the same processor. A (*processor ID*, *sequence number*) pair thus uniquely identifies an instruction in the system. During

execution, the simulator tracks instruction dependences and stores them in a file. Each line of the file contains information that uniquely identifies an edge in the program's DAG. For each edge, we store the (*processor ID*, *sequence number*) pair, the issue time, and the completion time of both its source and target nodes. We store only RAW edges, since we can derive program order edges by the sequence numbers of the nodes stored in the file. To ensure that we store only necessary information, the simulator dynamically checks whether it can perform a graph compression.

2.6.2 Results

In this section, we present our results on global slack distribution and how the critical path spans across the processors in a system. We then evaluate how different cache coherence protocols and the endpoint choice in a continuous workload affect the slack distribution. Finally we show results on the effectiveness of the graph compression techniques.

Slack Distribution. In Figure 2-8 and Figure 2-9, we plot the probability density function (PDF) of global slack for 2-, 4-, and 8-processor systems for the four commercial workloads in Table 2-2. We plot similar data for the two scientific workloads in Figure 2-10. The x-axis is global slack plus one shown in log scale, so as to provide more resolution for small values and allow the x-axis to represent zero global slack in log scale. The y-axis is the fraction of instructions that have the given global slack value specified by the x-axis. We observe that most instructions have global slack less than 100 ns. There are, however, spikes in the distributions between 100 and 200 ns, which correspond to the latency of inter-processor communication. Instructions in the scientific workloads show larger values of global slack on average than instructions in the commercial workloads: A large fraction of them have global slack greater than 100 ns and some are even greater than 1000 ns. This is because these workloads frequently use spinning synchronization when threads are waiting for locks, barriers, and flags. Instructions executed by spinning threads have non-zero global slack because they can be delayed without affecting overall program performance. Furthermore, the longer a thread spins, the larger its instructions' global slack is. As the number of processors in the system increases, the amount of synchronization contention

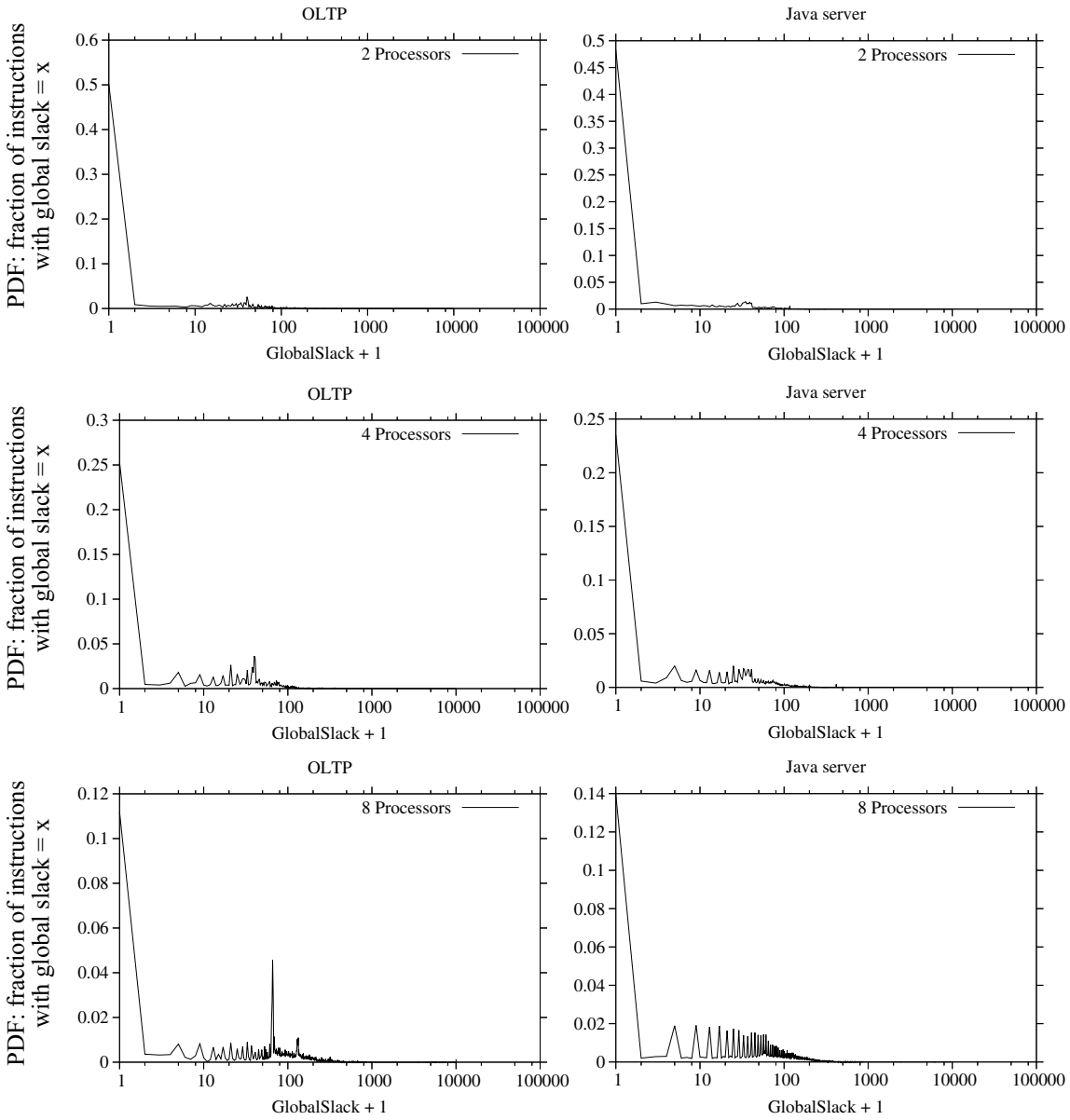


Figure 2-8. Slack distribution for OLTP and Java server.

increases as well, and thus waiting threads often spin for a larger amount of time. Therefore, in Figure 2-10, we see more spikes beyond 100 ns as the number of processors increases from two to eight. The large amounts of spinning in these workloads have motivated our research on automatic detection of spinning and applications based on the detection. We discuss this research in detail in Chapter 3.

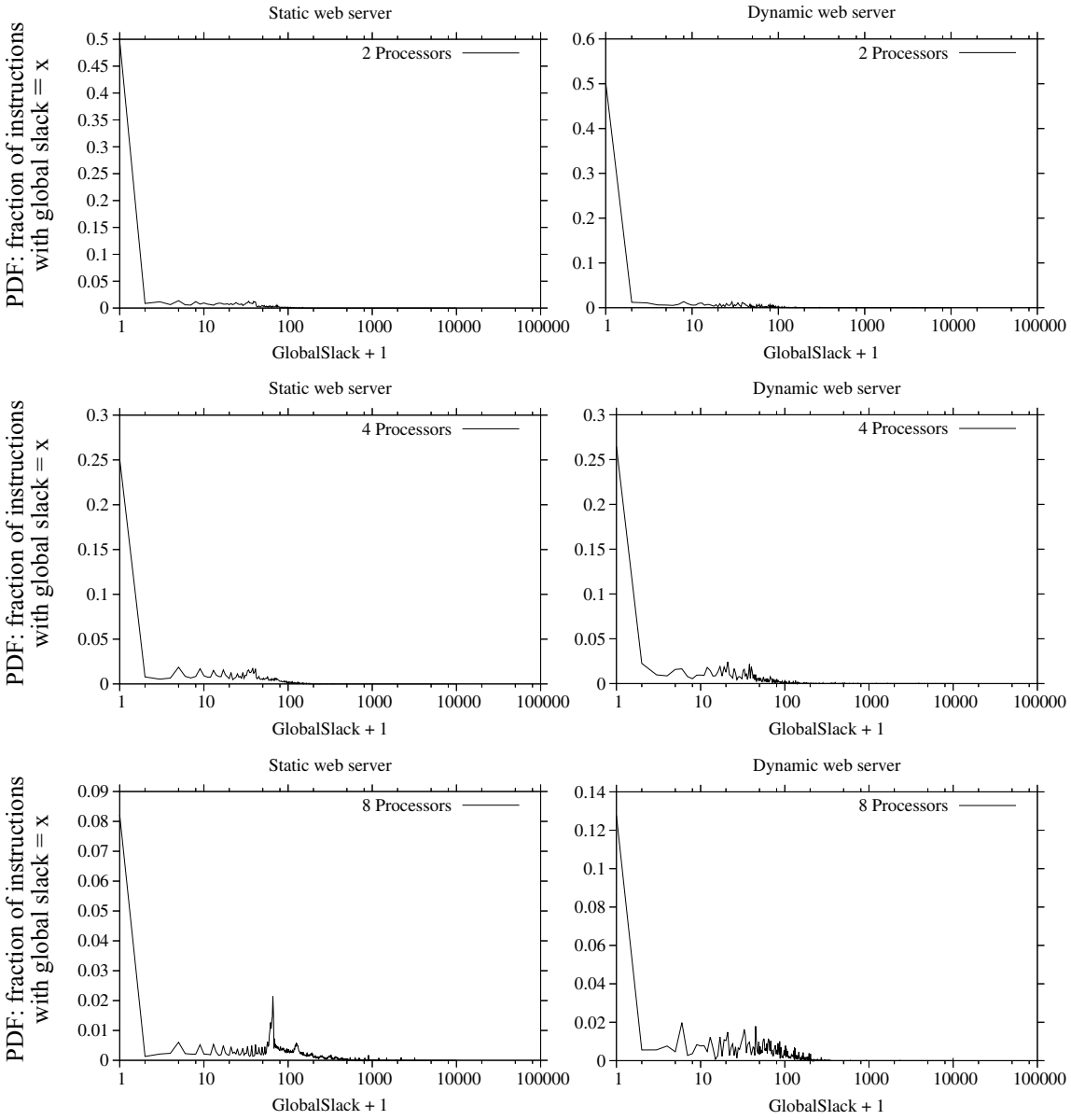


Figure 2-9. Slack distribution for static and dynamic web server.

Critical Path Time Breakdown. The breakdown of the critical path's time on each processor provides insight into the relative criticality of the processors. In Figure 2-11, we show the fraction of the critical path's time spent on each processor in an 8-processor system. We plot our results only for `ocean` and the dynamic web server workload; for the other workloads, the critical path is almost evenly distributed with a maximum 7% variation across all the processors. These critical path time breakdowns closely correspond with

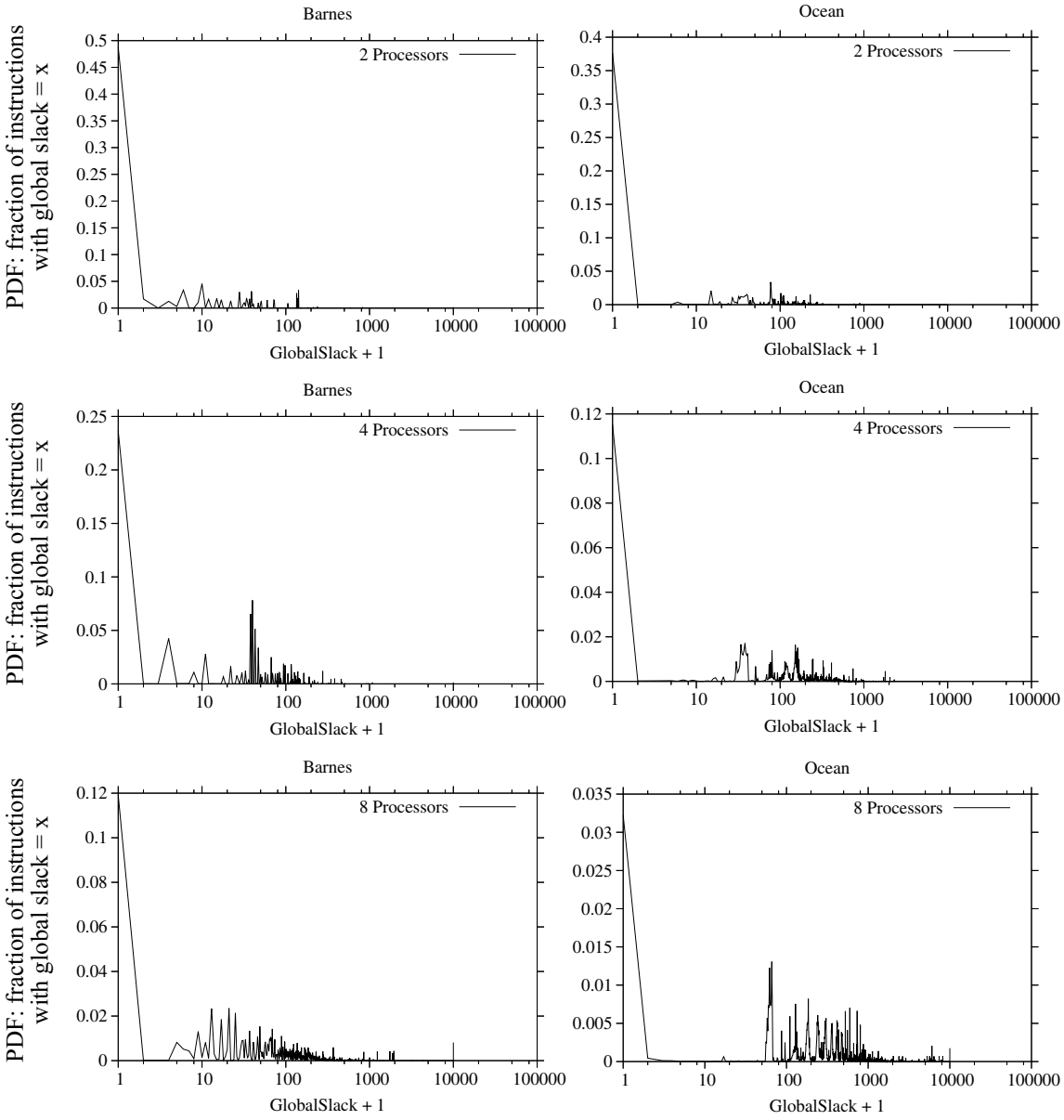


Figure 2-10. Slack distribution for the scientific workloads.

the L2 cache miss rates on the processors, since a processor with a larger L2 cache miss rate incurs more communication to remote memories, thus taking a higher fraction of the critical path's time. In *Ocean*, processor 0 dominates the critical path, which is unsurprising since processor 0 performs the sequential work in this algorithm.

Broadcast vs. Directory Protocols. To study how different cache coherence protocols affect the distribution of global slack, we compare our MOSI broadcast snooping protocol

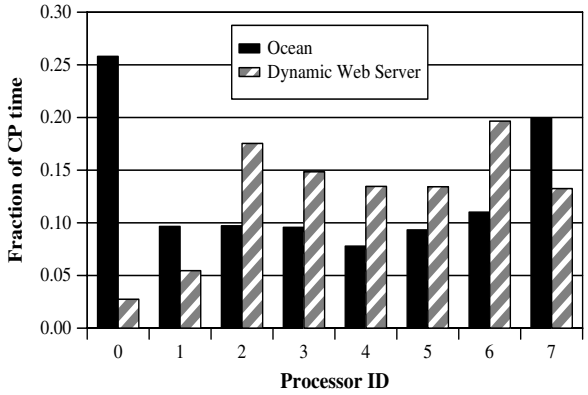


Figure 2-11. Fraction of critical path's time on each processor.

with a directory protocol similar to that used in the AlphaServer GS320 [20]. In Figure 2-12, we plot the cumulative distribution function (CDF) of global slack for all the workloads on an 8-processor system. The x-axis is the global slack value plus one shown in log scale. For any global slack x, the y-axis corresponds to the fraction of DAG nodes (instructions) that have global slack less than or equal to x. From the results, we see that instructions in the directory system possess more slack than those in the broadcast system. Since the systems we model have plentiful network bandwidth, by broadcasting cache block requests to all nodes in the system, the broadcast protocol avoids indirections and achieves better performance than the directory protocol. Our results show that, on average, the critical path of a workload in the directory system is 38% longer in time than that in the broadcast system. In the directory system, instructions typically wait longer for cache misses, thus making them have more slack in their executions.

Choice of Endpoint Processor. In Section 2.4.3, we claimed that, for continuous workloads, we can choose the last finished node from an arbitrary processor as the endpoint node. We now present results that support this claim. To investigate the effects of the endpoint choice, we run our algorithm in Figure 2-5 for each of our continuous workloads (the Wisconsin commercial workloads) P times for a system of P processors, and in each run we construct a DAG with the endpoint node chosen from a different processor. Our results show that, for each workload, all of the P DAGs have almost identical global slack distribution. Specifically, for any given global slack x, the fraction of instructions that have

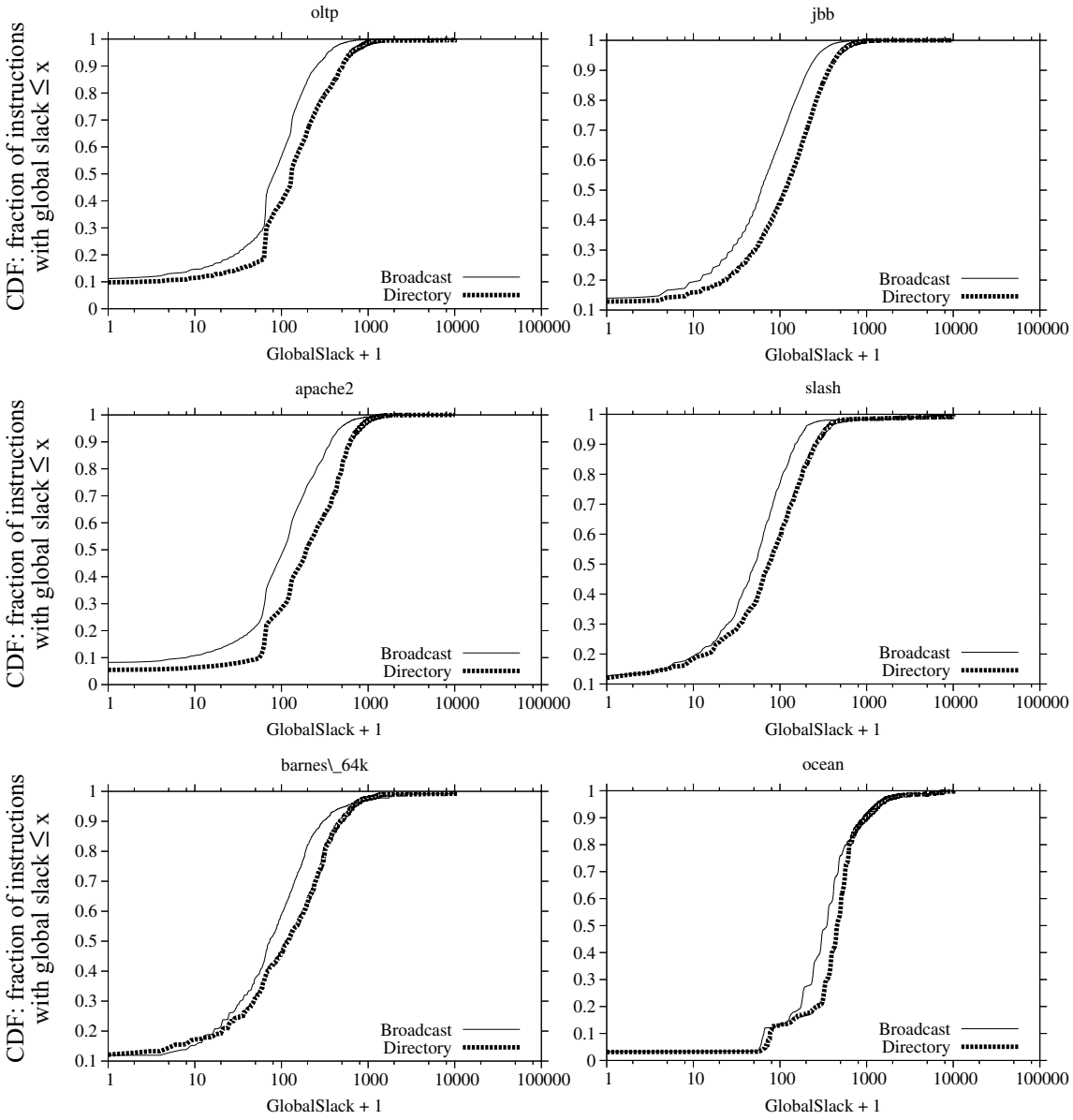


Figure 2-12. Impact of cache coherence protocol.

global slack x differs on average by the order of 10^{-7} in their values among all P DAGs. Moreover, the fraction of instructions that are on the critical path (i.e., with global slack zero) differs on average by 10^{-4} in their values among all P DAGs. These results indicate that the endpoint choice has negligible impact on the global slack distribution for continuous workloads. We expect that similar results hold for large execution intervals (e.g., with billions of instructions as in all of our workloads), because the endpoint nodes tend to have little impact on the global slack of nodes distant from them. While randomly choosing the

endpoint node may affect the computation of individual instruction slacks, our results suggest that we could design control policies for continuous workloads based on aggregated slack values over a large interval.

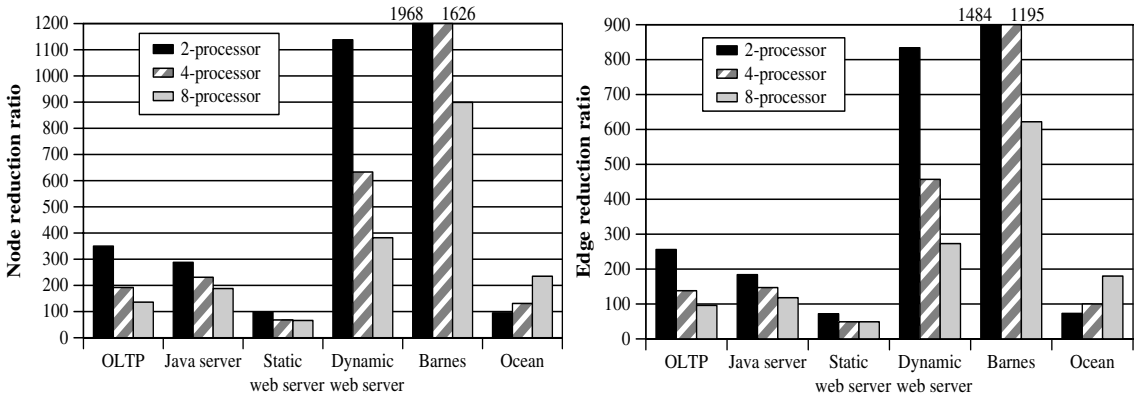
Graph Compression. We measure the effectiveness of Theorem 2-1 by the ratio of the number of nodes in a compressed DAG to the number of nodes in the original DAG. Similarly, we measure the combined effectiveness of Theorem 2-2 and Theorem 2-3 by the ratio of the number of RAW edges in a compressed DAG to the number of RAW edges in the original DAG. Figure 2-13(a) shows our results for the node ratios and Figure 2-13(b) shows the edge ratios for all of our workloads on the 2-, 4-, and 8-processor systems. For all the workloads and systems, we compress a DAG on average by a ratio of 485 in terms of the number of nodes and 352 in terms of the number of RAW edges[6]. The maximum node compression ratio is 1968 and the maximum RAW edge compression ratio is 1484. For all the workloads, except `ocean`, the compression ratios decrease as the number of processors increases. This is because with fewer processors, more RAW edges become local in a processor, which enables the conditions of Theorem 2-3 to occur more frequently. The `ocean` workload exhibits an opposite pattern because the conditions of Theorem 2-2 occur more frequently with more processors, which causes Theorem 2-2 to dominate the compressions.

2.7 Related Work

In this section, we discuss related work on instruction criticality, critical path and slack analysis, and graph compression. All previous work on instruction criticality target uniprocessor systems. This thesis presents the first research that performs instruction criticality analysis for shared memory multithreaded systems.

Srinivasan and Lebeck [54] studied latency tolerance of load instructions in dynamically scheduled processors. Their results show that loads in the same program can have dif-

6. The total edge compression ratios are even higher because, when we remove a node based on Theorem 2-1, we remove a program order edge as well.



(a) Node compression ratios. (b) Edge compression ratios.

Figure 2-13. Graph compression ratios.

ferent levels of criticality. For an 8-way processor they modeled, between 20% and 68% of the loads in their benchmarks need to complete in one cycle, while 67% to 97% can be delayed up to eight cycles without impacting overall performance. In their later paper, Srinivasan et al. [53] proposed a hardware predictor for load criticality. They associated each load with a counter, tracking the number of instructions that are independent of the load and issued within eight cycles following the load's issue. If the counter value is less than 16, the load is classified as critical. To determine the criticality of a load before it is issued, they employed a history-based criticality predictor, similar to a two-level branch predictor.

Fisk and Bahar [17] proposed two hardware designs for measuring the criticality of loads that miss in the cache. The first design checks if a load miss causes the processor's performance to degrade in terms of instruction issue rate or functional unit usage. If so, the load is classified as critical. In the second design, the hardware tracks the number of instructions that depend on a load from the time that the load misses to the time that the miss is resolved and the load commits. If the number of dependent instructions exceeds a certain threshold, the load is classified as critical. For both designs, the hardware places only data of critical loads in the normal cache; non-critical load data are placed in a small structure, called the Non-Critical Buffer (NCB).

Rakvic et al. [47] found that only 25% of loads in the SPEC95 benchmarks are non-vital (non-critical). To dynamically identify non-vital loads, they extended the processor rename register file. A load is non-vital if its dependents are not ready to execute when the load

finishes. Their design focused on only such non-vital loads because they constitute the largest portion of all non-vital loads, and their detection is easy to implement in hardware.

Casmira and Grunwald [8] studied instruction slack for improving processor power efficiency. They computed instruction slack based on partial dependence graphs containing only in-flight instructions (i.e., instructions that reside in the instruction window).

Critical path analysis is a powerful tool conventionally used in operations research for scheduling and managing complex projects [22]. In the research of computer architecture, critical path analysis has been primarily applied to multiprocessors for designing scheduling policies [31, 33] and analyzing performance bottlenecks [24, 25, 65]. Most of these studies model program dynamic events at the procedure level or above.

Fields et al. [16] developed the first instruction-level critical-path model. They used DAGs to model program executions on dynamically scheduled processors. As described in Section 2.3.1, their model takes into account various microarchitectural constraints. To determine on the fly whether a dynamic instruction is critical or not, they designed hardware with two components: a trainer and a critical path table. The trainer works by sampling the criticality of every E-node (representing the event that an instruction becomes ready to execute) in a program's DAG. To take a sample of an dynamic instruction's E-node *n*, the trainer plants a token into the node and propagates the token along all outgoing last-arriving edges. Whenever the token reaches a node and none of its outgoing edges is last arriving, the token dies, indicating that node *n* is not critical. If the token is still alive after the processor has committed some threshold number of instructions, node *n* is classified as critical. The trainer writes this result to the critical path table, indexed by the PC of the instruction to which node *n* belongs. For later instances of the same static instruction, the processor can obtain the criticality prediction by looking up the critical path table.

In their later paper [15], Fields et al. discussed that simply classifying instructions as critical and non-critical is insufficient for managing resources that have multiple "quality-of-service" levels. Thus, they extended their criticality model to characterize instruction slack. To determine if an instruction has *N* cycles of slack, they designed hardware that delays the instruction by *N* - 1 cycles and then observes if the delayed instruction becomes

critical or not, using the above token propagation approach. After the hardware determines an instruction's slack, it writes the slack value to a PC-indexed table, which is then looked up by the processor to predict the slack of future instances of the same static instruction. Our work in this chapter is the first research that extends the uniprocessor model of Fields et al. to shared memory multiprocessors.

Tune et al. [58] developed five criteria, each of which defines a heuristic for estimating whether instructions are on the critical path of a program. In each cycle, if the processor uses the QOLD criterion, it marks the oldest instruction in an instruction queue as critical. With the QOLDDEP criterion, the processor marks each instruction that produces a value consumed by the oldest instruction in the queue as critical. The ALOLD criterion causes the processor to mark the oldest instruction in the active list (a.k.a., reorder buffer) as critical. With the QCONS criterion, the processor marks the instruction that has the most number of dependent instructions in the instruction queue as critical. Finally, if the processor uses the FREED3 criterion, it marks an instruction as critical if the completion of this instruction's execution makes at least three instructions in the instruction queue ready to execute. Seng et al. [49] applied the QOLD criterion to design power optimizations for microprocessors. First, they used instruction criticality to direct the use of asymmetric functional units that differ in speed and power consumption. Second, they split critical and non-critical instructions into different issue queues and issue critical instructions in-order to save power.

Tune et al. [59] introduced *tautness*. The tautness of an instruction is the maximum time that the overall program runtime can be reduced by executing that instruction earlier. Different from slack, which quantifies the latency tolerance of an instruction, tautness quantifies the maximum performance benefits of applying an optimization to an instruction.

Similar to our graph compression, prior research has also explored the removal of unnecessary dependences from execution or task graphs. Beckmann [6] dynamically removed redundant data dependence edges in acyclic task graphs. Netzer [44] dynamically removed shared data dependencies from execution traces for debugging purposes, if the dependences were not necessary for replaying execution.

2.8 Conclusion

Trends in processor architectures are toward hardware-level multithreading by providing execution contexts that can execute multiple threads in parallel. Within each execution context, there are various hardware resources, such as functional units and instruction windows. Resources of the same type (e.g., integer functional units) also often have different speed and power consumption properties. Conventionally, processors treat all instructions as if they have equal effect on program performance, and use simple policies for resource allocations. Prior research [15, 16, 49, 53, 58, 59] advocates criticality-based resource management for uniprocessor systems. This thesis is the first research to model and evaluate instruction criticality for shared memory multithreaded systems.

To quantify instruction criticality, we adopt the approach that Fields et al. [15, 16] used for uniprocessor systems, and extend their uniprocessor criticality model to shared memory multithreaded systems. In our DAG model, each node represents a dynamic instruction and each edge represents a program order or RAW dependence between instructions. We quantify instruction criticality by using global slack, i.e., the maximum time that an instruction can be delayed without increasing program total runtime.

We evaluate instruction criticality using multithreaded commercial and scientific workloads. To enable efficient offline criticality analysis, we propose techniques that can compress a given DAG significantly. Our evaluation shows that instructions in multithreaded workloads possess diverse ranges of global slack values (and thus criticality levels), suggesting large benefits of criticality-based resource management for multithreaded systems. Our slack distribution results in Section 2.6.2 also reveal that, when applications have frequent spinning synchronization, many instructions have significant amounts of global slack, and thus they can be delayed for long periods of time without impacting program total runtime. Therefore, if a processor could detect spinning synchronization dynamically, it could manage resources more efficiently, for example, by running spinning instructions on slower, but more power-efficient functional units. To achieve this goal, in the next chapter, we study how to enable processors to dynamically detect spinning synchronization.

3 Monitoring Thread Synchronizations

In this chapter, we investigate how to enable multithreaded systems to self-monitor the synchronization behavior of application threads. Spinning (a.k.a., busy-waiting) and blocking are two waiting mechanisms that a thread uses when waiting for synchronization events. As we observed in Chapter 2, spinning synchronization presents opportunities for managing hardware resources more efficiently. Thus, in this chapter, we design mechanisms that allow a processor to automatically detect spinning; in the next chapter, we study how to monitor thread blocking synchronization, which enables the system to automatically detect application errors, namely, application deadlocks.

We make the following contributions in this chapter:

1. We demonstrate the prevalence of spinning and show how it misleads existing hardware performance counters.
2. We develop efficient hardware for detecting when a thread is spinning and when a group of threads are spinning.
3. Based on the hardware, we design (a) accurate performance counters, (b) efficient OS scheduling and power management policies, and (c) support for livelock detection.

In Section 3.1, we explain why spinning is a common phenomenon in multithreaded systems. In Section 3.2, we discuss why spin detection is useful and why we detect spinning using hardware instead of software. In Section 3.3, we study the general conditions that enable hardware to detect spinning without knowing application semantics. We then present our spin detector design in Section 3.4. Based on the spin detector, we develop mechanisms that improve system management in Section 3.5. We evaluate our spin detec-

tor and its applications in Section 3.6. In Section 3.7, we discuss related work and finally we conclude in Section 3.8.

3.1 Spinning in Multithreaded Systems

Spinning is a waiting mechanism with which the waiting thread continuously checks for the occurrence of a synchronization event. Blocking is an alternative waiting mechanism with which the OS suspends the waiting thread and schedules another thread to execute. Both spinning and blocking widely exist in multithreaded applications. For fine-grained multithreaded and SMT systems, hardware support, such as full/empty bits [2] and lock boxes [57], can potentially eliminate the use of spinning while still providing fast synchronization. However, to utilize the hardware support, programs often require re-coding and re-compilation, thus placing a burden on programmers and introducing costs that could even outweigh the benefits of the hardware support [48]. Because of the prevalence of spinning in existing code, Intel introduced the PAUSE instruction in Pentium 4 and Xeon to improve performance of spinning synchronization [28]. The POWER5 SMT design allows software to set lower scheduling priorities for spinning threads [29].

Locks, barriers, and flags are the three most prevalent synchronization abstractions and their implementations often consist of spinning. In the area of high-performance computing, OpenMP is the de facto standard for shared-memory parallel programming. OpenMP locks and barriers use spinning as a means for thread synchronization. Figure 3-1(a) shows the SPARC assembly code of a spin loop in `omp_set_lock`, a lock acquire function frequently used by programs in the SPEC OpenMP benchmark suite (SPEC OMP V3.0) [52]. Spinning also occurs in various commercial server applications, such as database servers. Figure 3-1(b) shows an example of flag spinning in the IBM DB2 database system controller process `db2sysc`. Besides user-space applications, spinning is also widely used within operating system kernels. Spin locks and flags are prevalent in most existing operating systems that support SMPs. Figure 3-1(c) shows a spin loop in the Solaris 8 kernel dispatcher.

```
0x2a134:  cmp     %i4, -0x1
0x2a138:  be      0x2a150
0x2a13c:  nop
0x2a140:  ld      [%i5], %i3
0x2a144:  cmp     %i3, -0x1
0x2a148:  bne,a   0x2a134
0x2a14c:  ld      [%i5], %i4
```

(a) Spinning in OpenMP.

```
0xfbf118    cmp       %o0, 0x0
0xfbf11c    bne,a,pt  %icc,<0xfbf118>
0xfbf120    ld        [%o1], %o0
```

(b) Spinning in DB2 controller.

```
0x1002c990  brnz,a,pt  %o0, <0x1002c990>
0x1002c994  ldub       [%i0 + 0xa0], %o0
```

(c) Spinning in Solaris dispatcher.

Figure 3-1. Spin loop examples in SPARC assembly.

In many operating systems, the idle loop is also a spin loop in which the operating system waits for work.

3.2 Motivation for Spin Detection

Our goal is to enable multithreaded systems to self-monitor their runtime behavior, such as performance, availability, and liveness. Existing systems already have certain support for monitoring system behaviors. For example, most systems provide performance counters (e.g., cache misses, instructions) that could encourage a programmer to apply a program transformation; some also monitor temperature for dynamic power adjustments.

Unfortunately, spinning synchronization complicates the development of hardware that provides accurate feedback on the behavior of multithreaded systems. A spinning thread executes a loop of instructions that performs no useful work with respect to its program's intended computation. Consider a thread that is spinning (e.g., `while flag == 1`) and the feedback that existing performance counters provide in this scenario. The counters would record a large value of instructions per cycle, since spinning instructions can execute quickly, and suggest that performance is terrific. The counters would also record a small percentage of cache misses per instruction, suggesting that memory system performance is also excellent. For these reasons, prior research [48] has observed that existing hardware schemes for self-monitoring of multithreaded systems are insufficient and misleading.

To overcome the challenges that spinning poses for multithreaded systems, we design mechanisms to detect the common form of spinning during the execution of a thread. Spinning commonly occurs in the form that the spinning thread executes a loop and, during each iteration of the loop, the thread tests if some condition is true. One approach to detecting such spinning is to instrument the spin loops in the applications or synchronization libraries used by the applications. However, instrumentation can be a significant burden on programmers. First, identifying all spin code in a large code base requires significant work. This is especially true when the code has components from multiple vendors or developers. Second, some spin code, such as spinning on a flag, does not use library routines, and identifying all such spin code can be difficult. Some code, such as the Solaris idle loop, exhibits spinning only on certain execution paths, and thus statically identifying it is difficult. Third, instrumented code requires re-compilation. Besides the burden on programmers, instrumentation also introduces performance overhead during program execution.

To free the burden from programmers and avoid perturbing application performance, we design hardware to dynamically detect if a thread is spinning. We focus on detecting a common form of spinning; Section 3.3 exemplifies a form of spinning that our hardware cannot detect. With hardware spin detection, existing programs require no modification to benefit from the mechanisms we propose in this chapter. For example, the hardware can detect that a thread is spinning, and automatically switch the processor to a low power mode or suspend the spinning thread. This also facilitates easy programming because programmers can use simple spin loops in the code, while the hardware automatically transforms spinning to blocking when necessary.

In Figure 3-2, we compare software and hardware spin detection for two benchmarks, `Radiosity` and `Volrend`, from the SPLASH-2 benchmark suite [63], running on a simulated 8-processor SMP system (see Section 3.6.1 for our simulation methodology). The *total* bar represents the total number of dynamic spinning instructions committed during the benchmark's entire execution. We obtain this value by instrumenting all spin loops in the benchmark's program and the PARMACS library [3], which provides synchronization routines for the SPLASH-2 benchmarks. The *software* bar corresponds to spins detected by a

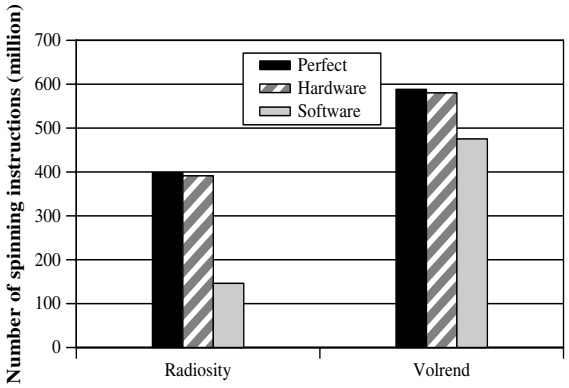


Figure 3-2. Comparison between software and hardware spin detection.

software detection mechanism, which instruments spin loops only in the PARMACS library. The *hardware* bar corresponds to spins detected by our hardware detection mechanism discussed in Section 3.4. As we can see, the hardware approach detects nearly all spins and is much more accurate than the software mechanism. The spins that the software mechanism misses are due to flag spinning specific to the programs (not common library routines); thus instrumenting only shared libraries is not enough for detecting all spins. However, identifying flag spinning in application programs requires thorough understanding of the programs, and instrumenting and re-compiling every program that possesses such spinning can be a tremendous burden on programmers. To avoid these problems, in the next section, we study general conditions that enable hardware to dynamically identify spinning.

3.3 General Conditions for Identifying Spinning

Our goal is to enable hardware to detect the common form of spinning during the execution of a thread. Intuitively, if a thread executes an instruction and later executes it again (e.g., in another iteration of a loop) with the state of the system unchanged, then the thread is spinning between the two executions of that instruction. More precisely, a thread on processor P is spinning between time t_a and time t_b if its execution satisfies the following two conditions.

SPIN CONDITION 1. *The observable state of the thread for the period between t_a and t_b is the same at t_a and t_b.*

For a given period of time in a thread's execution, the *observable state* of the thread includes the architectural registers and memory locations accessed by the thread during the given period. Since the observable state includes the program counter (PC), an important implication of this condition is that the thread executes the same static instruction (PC) at both t_a and t_b. Therefore, all instructions executed between t_a and t_b form a cycle in the control flow graph of the thread's program. We call this cycle a *spin loop* (although it can be more complex than a simple "loop") and each instruction executed in the spin loop a *spinning instruction*. If Spin Condition 1 is satisfied, all the architectural registers and memory locations accessed by the thread between t_a and t_b have the same values at t_a and t_b. Thus, the thread performs no useful work with respect to its computation on processor *P*. This condition, however, does not preclude the observable state from changing between t_a and t_b, as long as it changes back to its initial state by t_b.

SPIN CONDITION 2. *Any change made by the thread to its observable state between t_a and t_b is not observed by any thread outside processor P.*

This condition captures the scenario in which changes made by the thread to its observable state between t_a and t_b may cause changes to the observable state of another thread running on a different processor. This scenario can happen if the observable states of the two threads overlap, e.g., they both access the same memory locations. If a thread satisfies Spin Condition 2, then its execution between t_a and t_b does not affect the computation of any thread outside processor *P*.

The above two conditions provide the basis for spin detection and ensure that the hardware never incorrectly detects spinning that does not exist. These conditions, however, do not capture all forms of spinning. For example, they cannot detect spinning that occurs in a two-phase waiting algorithm [30, 36] in which a thread spins for a while and then blocks. In such a spin loop, a thread typically increments a counter in each iteration until it exceeds some threshold. The execution of the thread violates Spin Condition 1 because the thread's observable state is different at the start and end of each loop iteration. However, from the programmer's point of view, the thread does spin because none of the state changes contributes to the intended computation. Nevertheless, programmers often configure spinning

in a two-phase waiting algorithm to be short so as to optimize the overall synchronization performance. Therefore, missing the detection of such short spinning does not have much impact on the applications of spin detection that we develop in this chapter (see Section 3.5).

Based on these spin conditions, in the next section, we design efficient hardware to dynamically detect spinning. As we discussed earlier, although spin detection could be performed in software—using executable editing, program instrumentation, synchronization library support, and/or OS support—using hardware avoids perturbing application performance and removes this burden from programmers.

3.4 Dynamic Spin Detection

Dynamic spin detection involves checking whether the execution of a thread meets the two spin conditions in Section 3.3. For Spin Condition 1, we observe that any control flow cycle must have an instruction that causes a backward transfer of control flow. We call this instruction a *backward control transfer* (BCT). A BCT can be a backward taken conditional branch, unconditional branch, or jump instruction. Although other instructions such as calls, call returns, traps, and trap returns can also cause backward transfers of control flow, practical spin loops rarely rely only on these instructions to implement control flow cycles. Thus we do not include them as backward control transfers. For Spin Condition 1, we check whether a processor commits a BCT at time t_a and later commits the same BCT (PC) again at time t_b. If so, between t_a and t_b, the processor has executed one iteration of a control flow cycle. To further check whether the observable state of the thread is the same at t_a and t_b, we divide it into observable memory state and observable register state, and we discuss these issues separately in Section 3.4.1 and Section 3.4.2. We also show that, with our approach, if the execution of a thread satisfies Spin Condition 1, then it satisfies Spin Condition 2 as well. We describe how to detect nested spin loops in Section 3.4.3 and illustrate how our spin detection hardware works in Section 3.4.4.

3.4.1 Observable Memory State

In this section, we consider the observable memory state part of Spin Condition 1. Given the execution of a thread between time t_a and time t_b, we assume that any *non-silent store* [35] executed by the thread can cause its observable memory state at t_b to differ from its initial observable memory state at t_a. A non-silent store is a store instruction that writes to a memory address with a value different from the existing value at that address. Our assumption is conservative because the location changed by a non-silent store may later change back to its initial value. However, it simplifies our hardware for checking Spin Condition 1. Any non-silent store committed between t_a and t_b indicates that the observable memory state may differ at t_a and t_b. To detect non-silent stores to cacheable memory locations, we use the ECC store verify approach of Lepak et al. [35]. For stores to non-cacheable memory locations, such as I/O addresses, we conservatively assume they are all non-silent.

We assume an architecture in which processors can share all or part of the memory but not registers, i.e., instructions on one processor cannot write registers on a different processor.[1] Thus, for Spin Condition 2, a thread can only change the observable state of another thread by modifying a shared memory location. Given a thread's execution between t_a and t_b, if all stores are silent, then the thread makes no change to shared memory locations, and thus it satisfies Spin Condition 2. Therefore, by detecting non-silent stores, we can check if a thread satisfies both the observable memory state part of Spin Condition 1 and the entire Spin Condition 2.

3.4.2 Observable Register State

In this section, we describe how to check the observable register state part of Spin Condition 1. Given the execution of a thread between time t_a and time t_b, the observable register state of the thread is the same at t_a and t_b if and only if the entire register state of the processor is the same. The register state of a processor consists of all the processor

1. We treat I/O devices as processors that share I/O space memory with the host processor to which they are connected.

architectural registers, including control registers, but excluding performance counters and registers that are mapped to I/O space locations.[2] Therefore, for Spin Condition 1, we can save the processor register state at t_a and check if it is the same at t_b. This approach is logically simple, but the challenge is to implement it efficiently.

To efficiently check if the observable register state remains the same, we note that the only registers that can change between t_a and t_b are those written by the thread. Thus, checking if the observable register state of a thread remains the same is equivalent to checking if the registers written by the thread are the same. In practice, spin loops are often small and each loop iteration only writes a small number of architectural registers. In all the workloads we study, the maximum number of architectural registers written in a spin loop iteration is only 17. Therefore, by maintaining and comparing only the registers written by the thread, we can greatly reduce hardware costs. The difficulty, however, is that the processor does not know a priori at t_a what registers the thread will write during its execution between t_a and t_b. Nevertheless, our implementation can discover these registers as the execution progresses.

We use a *Register Update Buffer* (RUB) to dynamically track the architectural registers written by a thread in each iteration of a potential spin loop. *An invariant in our algorithm is that the RUB always holds the architectural registers whose current values are no longer the same as when the loop iteration began*. The RUB is empty when the processor starts a potential spin loop iteration, i.e., when it commits a BCT. For each instruction it then commits, the processor checks whether the instruction's architectural destination register (a.k.a., the logical destination register) is already in the RUB. If not, the processor has discovered a new register written by the thread. It then compares the new value of this register (i.e., the value being committed) to its old value (i.e., the value before being overwritten by the commit). If not equal, the processor adds the register number and its *old* value to the RUB. If the instruction's destination register is already in the RUB, then the processor compares the new value of the register to its value currently in the RUB. If equal, it deletes this

2. Since no practical spin loop involves performance counters, we do not include them in processor register state. We do not include memory-mapped registers in processor register state because we treat them as part of the memory state.

register from the RUB; otherwise no action is necessary. With this algorithm, when the processor reaches the end of the iteration, i.e., when it commits a new dynamic instance of the same static backward control transfer, it checks if the RUB is empty. If so, the observable register state of the thread is the same as when the iteration began. We will present full details of the RUB implementation in Section 3.4.4.

3.4.3 Nested Loops

Identifying a control flow cycle (even without trying to detect spinning) involves saving the PC of a BCT and checking if the thread commits the same PC again. At a first glance, it seems sufficient to save only the PC of the most recently committed backward control transfer. This is, however, not true in the presence of nested loops. Figure 3-3 shows the SPARC assembly code for an example nested spin loop in the lock acquire routine of the PARMACS library. When a thread fails to acquire a lock, it spins until successfully performing both the test and test&set.

To identify a nested spin loop, we need to keep multiple PCs, corresponding to the BCT of each level of the nested loop. Since the depth of nesting is typically small in realistic spin loops, we only need to keep the PCs for a handful of the most recent BCTs. To keep these PCs, we add a *Spin Detection Table* (SDT) after the commit stage of the processor pipeline. For each BCT it commits, the processor searches the SDT for a matching PC. A matching PC signifies that the processor has executed a control flow cycle. If no matching PC exists, the processor inserts the PC of this BCT at the top of the SDT, and pushes down all the existing SDT entries, thereby evicting the bottom entry if the SDT is full. To avoid physically moving the SDT entries, we manage the SDT as a circular array and use two registers, `SDT_top` and `SDT_bottom`, to maintain the indices of the top and bottom SDT entries. For all our workloads, we show that a 16-entry SDT provides the same results as an unbounded SDT.

```
            0x12cac   cmp      %l1, 0x0
TTS:        0x12cb0   bl,a     0x12cd4    // test and test&set
            0x12cb4   ld       [%i0], %g2 // load lock status
            ... (never here)
            0x12cd4   cmp      %g2, 0x0   // test lock
BCT 1:      0x12cd8   bne      0x12cb0    // fail test, retry TTS
            0x12cdc   cmp      %l1, 0x0   // delay slot
            0x12ce0   mov      %i0, %o0
            0x12ce4   call     test&set   // test&set lock
            0x12ce8   mov      0xff, %o1
            0x12cec   cmp      %o0, 0xff
BCT 2:      0x12cf0   be       0x12cb0    // fail test&set, retry TTS
            0x12cf4   cmp      %l1, 0x0   // delay slot
            0x12cf8   ret
            0x12cfc   restore
            … (never here) …
test&set:   0x134f8   ret
            0x134fc   ldstub   [%o0], %o0 // atomic load-then-store 0xff
```

Figure 3-3. Nested spin loop in PARMACS's lock acquire routine.

3.4.4 Putting It All Together

In this section, we present a complete view of how our spin detector works. The spin detector has an SDT, and each SDT entry keeps a PC corresponding to a potential control flow cycle. Whenever the processor commits a non-silent store, it clears the entire SDT by setting both `SDT_top` and `SDT_bottom` to invalid. To determine if a control flow cycle is a spin loop, we need to maintain for each SDT entry the observable register state of the running thread. One approach is to associate a RUB with each SDT entry, but then a processor may insert the same register information into multiple RUBs if they all do not already contain the register. To avoid such redundancy, we use a single RUB for all SDT entries, and, for each register in the RUB, we use a bit vector to track SDT entries to which this register

corresponds. A RUB entry contains four fields: a register number (`RegNum`), the value of the register (`RegVal`), a valid (V) bit, and an SDT bit vector. A one in the i^{th} least significant bit of the bit vector indicates that register `RegNum` corresponds to SDT entry *i*. For each BCT the processor commits, if its PC matches the PC of an SDT entry, say *k*, then the processor does a wired-OR over the k^{th} bit of all valid RUB bit vectors. If the result is one, then there exists at least one entry in the RUB that corresponds to this SDT entry. This means that the observable register state of the thread differs and thus the processor was not spinning. If the result is zero, then the observable register state is the same and thus the processor was spinning.

To manage RUB space, we use a free map (implemented as another bit vector) to track free RUB entries, i.e., entries with valid bit zero. Initially all valid bits are zero. Any insertion or deletion consults the free map. When the processor inserts a new entry into the RUB (according to the conditions in Section 3.4.2), it initializes the entry's valid bit to one, and the SDT bit vector to have ones in the bits corresponding to all the valid SDT entries and zeros elsewhere. When the processor inserts an entry into a full RUB, it locates all SDT entries pointed to by this entry's bit vector. For each of these SDT entries, the processor sets a `RUB_overflow` (O) bit in the entry to indicate that the RUB is not large enough to hold the corresponding observable register state. The processor then considers (conservatively) that the observable register state for these SDT entries has changed and will never detect spinning for them. Thus all of the registers maintained in the RUB for these SDT entries are no longer useful. The processor clears the bits pointing to these entries from the bit vector of each RUB entry. If a bit vector becomes all-zero after the clear, the processor adds its RUB entry to the free map. Similarly, when the processor evicts an entry from the SDT, it clears the entry's corresponding bit in the bit vectors of all RUB entries. If a bit vector becomes all-zero, the processor frees the corresponding RUB entry.

Figure 3-4 illustrates the operation of a two-entry SDT and a two-entry RUB for a simple nested spin loop. Initially the SDT and RUB are empty and all registers are zero. The processor starts at instruction I0 and follows steps (a)–(e). Finally, when I3 commits

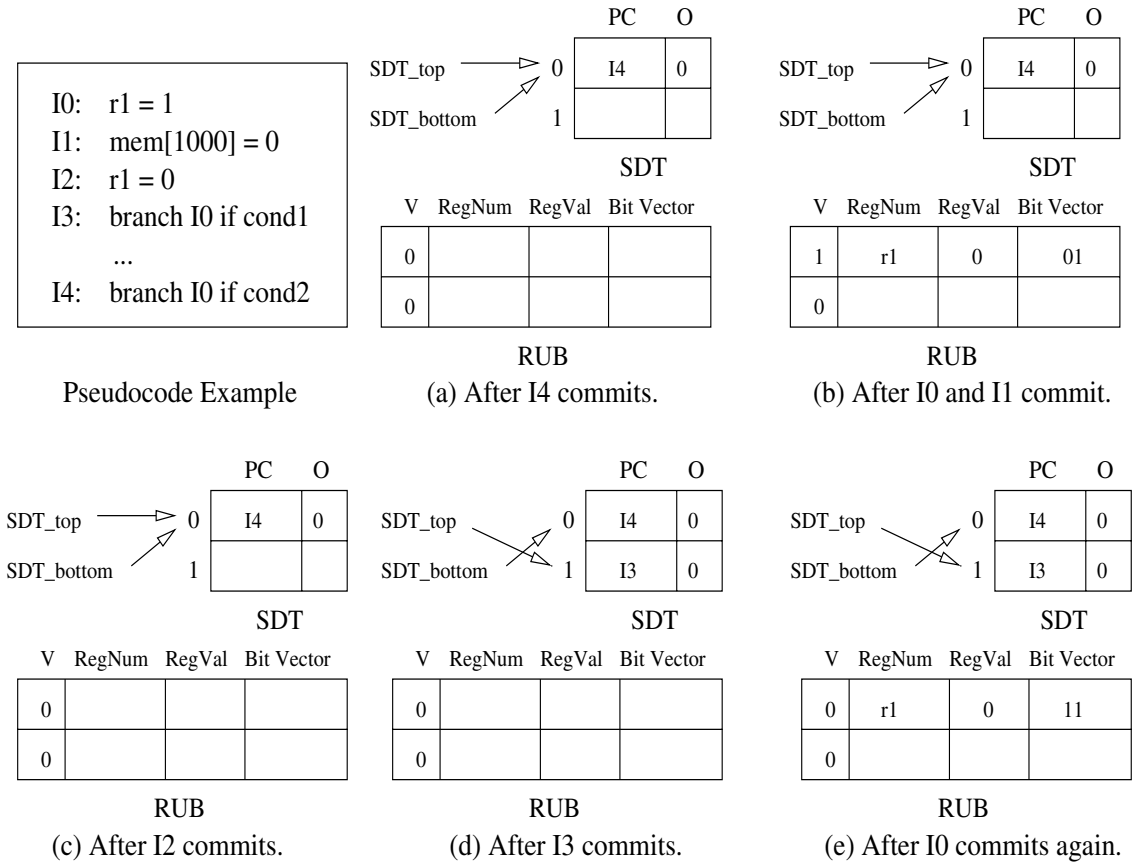


Figure 3-4. Operation of two-entry SDT and two-entry RUB. (a) Assume that I3 is not taken and I4 is taken. Thus I4 is inserted to the SDT after it commits. (b) After I0 commits, r1 and its old value 0 are inserted to the RUB. The bit vector is set to 01 to indicate that the RUB entry corresponds to SDT entry 0. Assume that I1 is a silent store, so it has no effect. (c) I2 changes r1 back to its original value 0, so its RUB entry is deleted. (d) Assume that I3 is taken this time. Set SDT_top = (SDT_top-1+SDT_size) mod SDT_size, and insert I3 at the top of the SDT. (e) I0 commits again. Insert r1 and its old value 0 to the RUB. The bit vector is 11, indicating that the RUB entry corresponds to SDT entries 0 and 1.

again, both the SDT and RUB are the same as in Figure 3-4(d). The processor finds that SDT entry 1 has the same PC as I3 and no entry in the RUB corresponds to it, thus concluding that the thread spun since the last dynamic instance of I3. Our experiments in Section 3.6 show that a 16-entry SDT and a 64-entry RUB (totally less than 1 KB in size) can detect spinning as accurately as if the sizes of these structures were unbounded. The RUB hardware is similar to a processor issue queue in that it performs associative searches

for matching registers. Since the spin detector is small and off the critical path of a processor, it has no impact on processor clock cycle times.

3.5 Improving System Management

In this section, we study how we can apply spin detection to improve management of multithreaded systems. We extend our spin detector to develop accurate performance counters (Section 3.5.1), improve thread scheduling and power management (Section 3.5.2), and support dynamic livelock detection (Section 3.5.3). We experimentally evaluate each of these applications in Section 3.6.

3.5.1 Accurate Hardware Performance Counters

Both hardware and software often rely on performance counters to derive system performance and make dynamic decisions. For example, to save power, the hardware can dynamically resize its issue queue based on online calculation of IPC [18]. IPC is a widely used performance metric for single-threaded microprocessors, as well as multithreaded systems [37, 45, 56, 64]. However, the presence of spinning can lead to arbitrarily inflated IPC numbers that mismatch the actual system performance. For example, a processor spinning on a lock can achieve high IPC while doing no useful work. To accurately measure the performance of multithreaded systems, we propose *useful IPC* (uIPC), which counts only useful (non-spinning) instructions committed per cycle.

We extend our spin detector to provide a new performance counter, called *PerfCtr-SpinInstrs*, which counts the number of spinning instructions committed by the processor. To compute uIPC, we subtract *PerfCtr-SpinInstrs* from the total number of instructions and divide by the number of cycles. To count spinning instructions, we add a counter to each SDT entry. When the processor inserts a new entry into the SDT, it initializes the entry's counter field to zero. When an instruction commits, all SDT entries increment their counters by one. When a processor detects spinning for an SDT entry, it adds that counter value to *PerfCtr-SpinInstrs*. To avoid double-counting of spins because of nested loops, when the processor detects spinning for an SDT entry, it subtracts the counter in the entry

from all entries below it in the SDT. Finally, it resets the entry's counter to zero and pops all entries above it.

3.5.2 Scheduling & Power Management

In this section, we show how dynamic spin detection can help enable more efficient scheduling and power management.

3.5.2.1 Scheduling

We extend our spin detector to implement a *spin-then-yield* synchronization mechanism to improve thread scheduling. When the hardware detects that a thread has spun for *N* iterations (i.e., the processor has committed the same BCT *N* times), where *N* is a tunable parameter, it raises an interrupt to the processor. The OS services this interrupt by invoking a handler routine, which blocks the spinning thread and moves it to the end of the run queue, thus allowing another thread to execute. The spinning thread resumes execution when it returns to the head of the run queue and the OS dispatcher selects it to run again.

The spin-then-yield mechanism is similar to the software-level two-phase waiting algorithm in which the thread spins for a certain number of iterations and then invokes the `yield` system call to relinquish the processor; the difference is that our approach can automatically convert spinning to two-phase waiting, thus simplifying the job of programmers.

3.5.2.2 Power Management

Another usage of our spin detector is to perform dynamic voltage scaling to save power when a processor detects that its current thread is spinning. The processor can switch to a low power mode until it detects that the thread is no longer spinning.

To support detecting when a thread stops spinning, we assume a shared-memory multiprocessor system with an invalidation-based cache coherence protocol; however, our design can be easily adapted to other types of system. A thread can exit a spin loop only if it observes a change to its observable state, and the change is possible only if some other thread or I/O device modifies a cache block that the thread accesses within the spin loop. To track cache blocks that a thread accesses within a spin loop, we use a *load address cache*

(LAC). If a processor detects that its current thread has spun for one iteration and if the BCT is at the top of the SDT, the processor sets a `try_suspend` bit to one, and then starts recording the block addresses of subsequent loads into the LAC. If the processor detects a second iteration of the spin loop, it switches to a low power mode and resets the `try_suspend` bit. The cache controller then probes the LAC for every invalidation it receives. On a match, the spinning thread would potentially stop spinning. Thus the processor switches back to the normal power mode and it clears the LAC. The LAC could overflow before the processor switched to the low power mode. If so, any invalidation will cause the processor to switch back to its normal power mode. The processor also switches back to the normal power node and clears the LAC on every timer interrupt. The operation of the LAC is similar to the load-locked store-conditional primitives in Alpha processors that provide support for atomic instruction sequences [12].

A subtle complication to the above operation is the potential, due to deep pipelines and out-of-order execution, for a load to read a value and have the corresponding cache line invalidated by another processor well before the load reaches the commit stage. Furthermore, there are likely multiple iterations of the spin loop in-flight when any given load commits. A load from an in-flight iteration of the spin loop could read a new value that would cause the processor to stop spinning, but earlier committed iterations of the loop could erroneously trigger the processor to switch to the low power mode. Therefore, to avoid this scenario, the thread must also detect cache invalidations (potential changes to observable state) that occur between when a load issues and when it commits. To achieve this, the processor records the issue time of each load in the reorder buffer entry (or load queue entry depending on the architecture) corresponding to the load. The processor also updates a timestamp when it receives an invalidation from another processor. When a load commits, the processor compares the load issue time and the timestamp, and sets a `recv_invalidate` bit to one if the load issue time is smaller. Upon reaching the BCT for the third time, the processor switches to the low power mode only if `recv_invalidate` is zero. Otherwise, the processor clears all LAC entries and `recv_invalidate` and stays in its normal power mode. If the thread is still spinning, then `try_suspend` remains set; otherwise, the processor clears it. Note that this imple-

mentation is overly conservative in that any invalidation can prevent the processor from switching to the low power mode. A more precise implementation would monitor only the addresses that are observable from within the spin loop. However, our simple implementation is sufficient to serve as a proof of concept.

3.5.3 Architectural Support for Livelock Detection

In this section, we discuss how we extend our spin detector to support livelock detection. From the hardware's perspective, a program is *livelocked* if it executes instructions indefinitely but makes no forward progress in its computation. Our livelock definition encompasses certain situations that software may otherwise consider to be deadlock. For example, when multiple threads of a program wait on each other in a circular fashion, we consider the program livelocked, instead of deadlocked, because it continues to execute instructions.

Conventionally, people prove liveness properties for multithreaded programs using static techniques such as temporal logic [46], model checking [27], and reachability analysis [11]. However, these techniques are impractical for large software systems due to the state space explosion problem. Recently, Engler and Ashcraft [14] proposed RacerX, a static tool that can detect deadlocks in large software systems. However, RacerX focuses on detecting deadlocks involving only lock-like resources. We have proposed Pulse (details in Chapter 4), an operating system mechanism that dynamically detects deadlock. Pulse makes no assumption about resource types, and thus can detect deadlocks that RacerX cannot detect. An assumption of Pulse is that all threads involved in a deadlock are blocked in the OS kernel. After identifying the blocked threads, Pulse speculatively executes them ahead in their programs. For each speculative thread, Pulse records the actions performed by the thread that could unblock another thread (e.g., the release of a lock on which another thread is blocked). Such information allows Pulse to discover the dependences among the blocked threads and detect deadlock if the dependences form a cycle. With our spin detector, we can extend Pulse to detect deadlocks involving both blocked and spinning threads.

In this section, we focus on livelock/deadlock that involves only spinning threads. Pulse relies on monitoring actions that threads perform within the OS kernel to discover thread

dependences. However, most applications perform spinning in the user space, via library calls or application-specific spin routines. Thus, simply extending the speculative execution framework of Pulse requires modifying user applications to inform the kernel about the relevant thread actions. This adds a burden to programmers and introduces performance overhead. To avoid these problems, instead of using speculative execution, we design simple hardware to detect when the threads of a program are simultaneously spinning.

Intuitively, if all threads of a program are simultaneously spinning, then the program is livelocked. Nevertheless, a program could have a subset of its threads livelocked while the rest still moves forward. Thus detecting livelock involves two parts: the OS specifies a group of threads of interest, and the hardware detects if each thread in the group is spinning. Simultaneous spinning of all threads, however, is not sufficient for claiming livelock; the OS also needs to guarantee that no thread outside the group nor any future I/O event could cause a spinning thread to stop spinning. Such guarantees may not be easy to obtain in general, but may be reasonable in some situations. For example, lock-induced livelocks often have well-defined thread groups competing for locks. A thread can stop spinning only if another thread in the same group releases a lock; no thread outside the group or I/O can affect the lock on which the thread spins.

Using the spin detector, our hardware checks if a group of threads are spinning simultaneously. Assume for now that the number of threads in the group (T) is less than or equal to the number of processors (P) in the system, and all T threads run concurrently on different processors. The problem with detecting simultaneous spinning at any *physical* time is the presence of in-flight invalidation requests (for cache coherence) that could cause some threads to stop spinning at a later time. Thus, we use *logical* time, in which no invalidations are in-flight. To implement logical time, we leverage our invalidation-based broadcast snooping cache coherence protocol. For every processor that runs a thread in the group, we initialize its logical time (a 64-bit integer) to zero. Whenever a processor observes a cache invalidation request, it increments its logical time by one. Logical time as such ensures that no in-flight invalidation requests exist when a group of threads is spinning simultaneously.

To support detecting simultaneous spinning, we extend each SDT entry with a time field. When a processor inserts a new entry into its SDT, it sets the entry's time to the processor's current logical time. Whenever the processor detects a spin loop, the corresponding SDT entry's time and the current logical time together form the interval in which the thread was spinning. A processor then sends this interval to the OS (e.g., via an interrupt) or a system service controller (like the one in Sun's Starfire [10]). The OS or the controller keeps for each thread the most recent interval it receives and checks periodically if all of the T threads' intervals intersect. If so, they have been spinning simultaneously.

We now consider the case that T is greater than P. The challenge is that the T threads never all run at the same logical time, and thus their spin intervals may never intersect.[3] To overcome this, we include the SDT and RUB as part of a thread's state that the OS saves and restores at a context switch. If the OS switches a thread out while it is spinning and switches it back in later, based on the restored SDT and RUB, the processor can determine that the thread has been spinning since the last time it was running until now (i.e., the spin interval includes the period during which the thread is not running). Thus, if the T threads are livelocked, eventually the system can detect that they spin simultaneously.

3.6 Evaluation

In this section, we first describe our simulation methodology (Section 3.6.1). Then we present our evaluation results on spin detection (Section 3.6.2), useful IPC (Section 3.6.3), scheduling (Section 3.6.4), and livelock detection (Section 3.6.5).

3.6.1 Methodology

We simulate a symmetric multiprocessor as our target multithreaded system. We simulate this system using Simics/sun4u 1.4.4 [38] with the same memory model as in Section 2.6.1.

3. In a multiprogrammed system, this situation can happen even when $T \leq P$.

Table 3-1. Target processor parameters.

clock frequency	2 GHz
reorder buffer	128 entries
pipeline width	4
pipeline stages	11
direct branch predictor	YAGS, 1024-entry choice PHT, 256-entry direction caches
indirect branch predictor	cascaded, 64-entry leaky BTB filter and second stage table
integer registers	160 (logical) + 64 (rename)
floating-point registers	64 (logical) + 128 (rename)

We extend Simics with TFSim [39] to model the timing of dynamically scheduled processors. Table 3-1 shows the configuration parameters of our target processor.

To evaluate spin detection and its applications, we use SPEC OMP v3.0 [52] and SPLASH-2 [63] benchmarks, and two commercial workloads, Java server and static web server, from the Wisconsin Commercial Workload Suite (see Table 2-2).

For the SPEC OMP benchmarks, we use the medium versions and compile them using Sun Studio 9 with flags suggested by SPEC for Sun systems. We omit three benchmarks: `galgel`, `mgrid`, and `wupwise`, because they take too much time to run on our simulator. Table 3-2(a) shows our settings for the OMP benchmarks. We use the training inputs, as opposed to the reference inputs, such that the simulations finish within a reasonable amount of time. The central part of most OMP benchmarks is a major loop whose body contains code executed by multiple threads in parallel (one iteration of the loop often contains a significant amount of computation). We set the number of OpenMP threads equal to the number of processors in the simulated system. For every benchmark except `art` and `equake`, we warm up the system for one iteration and then run one more iteration for detailed timing simulation. The code of `art` does not have a major loop. Thus we consider its entire program as one "iteration" and run it for a 36×36 windowed image until completion. For `equake`, we run it for two iterations, since it contains relatively little computation in one iteration of its major loop compared to the other benchmarks.

Table 3-2. Configurations of the scientific workloads.

(a) SPEC OMP settings.

Benchmark	Warmup	Simulation
AMMP	1 iteration	1 iteration
Applu	1 iteration	1 iteration
Apsi	1 iteration	1 iteration
Art	none	36×36 image
Equake	1 iteration	2 iterations
Fma3d	1 iteration	1 iteration
Gafort	1 iteration	1 iteration
Swim	1 iteration	1 iteration

(b) SPLASH-2 problem sizes.

Benchmark	Warmup	Simulation
Barnes	512 particles	16K particles
FMM	512 particles	16K particles
Ocean	66×66 grid	258×258 grid
Radiosity	room	room
Raytrace	car	car
Volrend	head-scaleddown4	head
Water-Nsquared	512 molecules	512 molecules
Water-Spatial	512 molecules	512 molecules

Table 3-2(b) shows the problem sizes that we use for the SPLASH-2 benchmarks. For each benchmark, we first run it with the warmup problem size to prime the system, and then we run it again with the simulation problem size to perform detailed timing simulation. To avoid measuring thread forking, our timing simulation starts at the beginning of the parallel phase and continues until the benchmark finishes.

Throughout our evaluation, we use a spin detector with a 16-entry SDT, 8-entry LAC, and 64-entry RUB, totally less than 1 KB in size. In all our experiments, we obtain the same results as if the sizes of these structures were unbounded.

3.6.2 Spin Detection

3.6.2.1 Microbenchmarks

We first evaluate our spin detector with two microbenchmarks to show that it can accurately detect spinning in both lock-intensive applications and the operating system kernel. The first microbenchmark, called `contended lock`, implements *N* threads in an *N*-processor system. All threads vie for a single spin lock, which provides mutual exclusion for a counter that each thread tries to increment until it reaches 100,000. The second microbenchmark, called `null`, models five million cycles of an idle system in which only the operating system runs and no user threads exist. We run both microbenchmarks on a

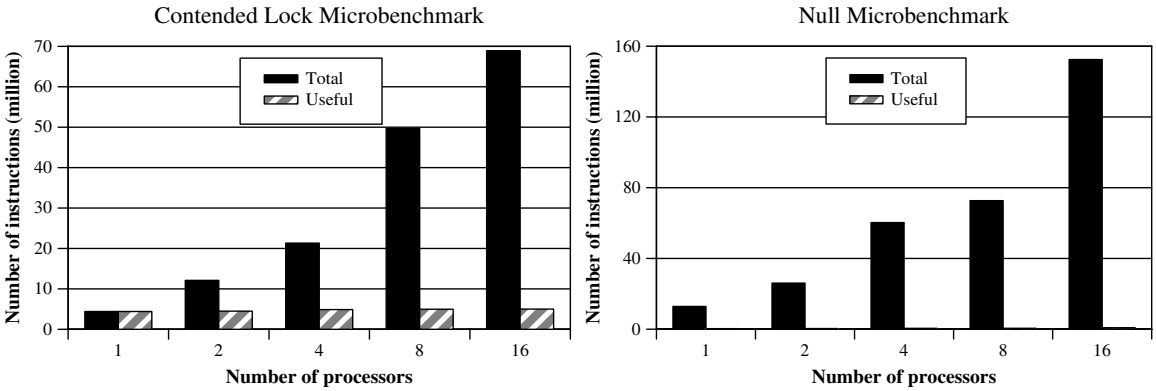


Figure 3-5. Total vs. useful instructions for the microbenchmarks.

system of *N* processors, where *N* equals 1, 2, 4, 8, and 16. In Figure 3-5, we plot the total and useful instructions that the system commits. The results show that we detect significant amounts of spinning. As *N* increases, the total instructions increase, but the useful instructions stay constant. For `contended lock`, this result illustrates that lock contention increases as the number of processors (threads) increases; however, the total amount of useful work that the threads perform (incrementing the counter to 100,000) remains the same. For `null`, spinning comes from the Solaris idle thread, which executes a surprisingly complex nested loop to test for runnable threads. The useful instructions in `null` are so few that their corresponding bars are hardly visible in the figure. These two microbenchmarks demonstrate that our spin detector can accurately detect both simple spinning due to spin locks and complex flag spinning that involves hundreds of instructions in one spin loop iteration.[4]

3.6.2.2 Scientific and Commercial Workloads

In Figure 3-6, we show the total number of committed instructions versus useful instructions for the SPEC OMP benchmarks. We show similar results for the SPLASH-2 benchmarks in Figure 3-7 and for the two commercial workloads in Figure 3-8. The spinning in the SPEC OMP and SPLASH-2 benchmarks is due to spin locks, flags, and barriers. For all these benchmarks, we see that the number of total instructions increases as the number of

4. Recall from Section 3.3 that the two spin conditions ensure that we never incorrectly detect spinning that does not exist.

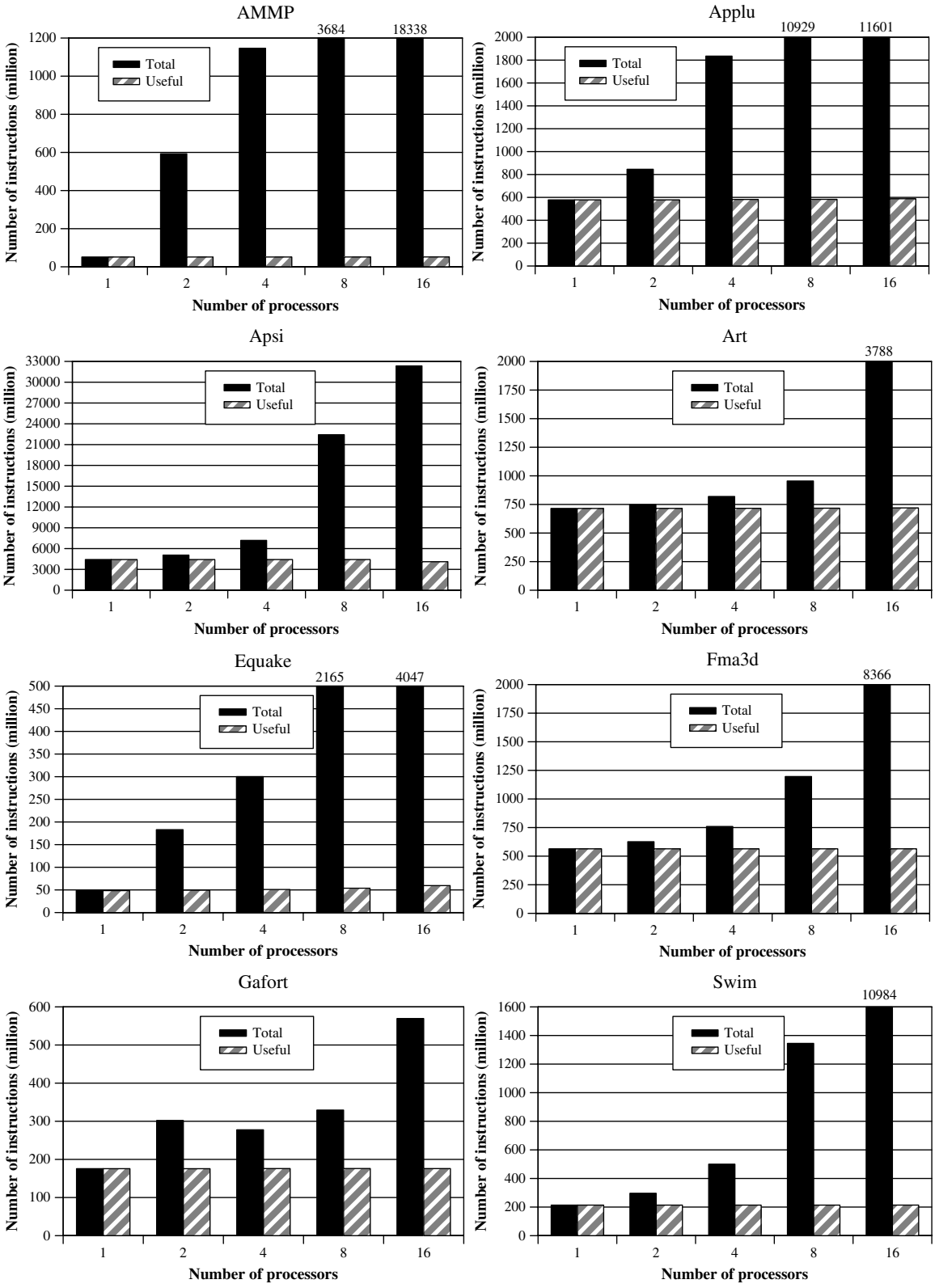


Figure 3-6. Total vs. useful instructions for the SPEC OMP benchmarks.

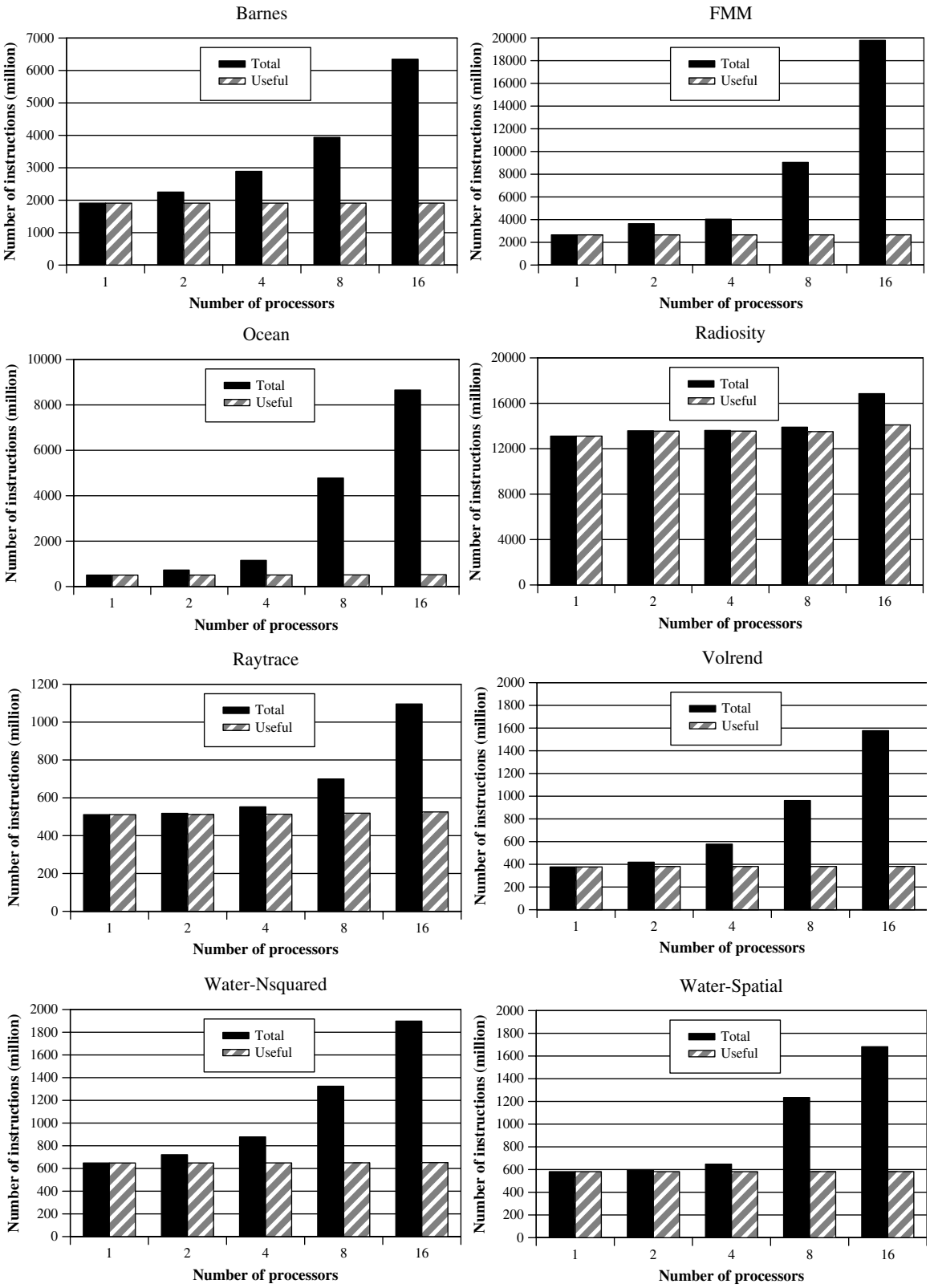


Figure 3-7. Total vs. useful instructions for the SPLASH-2 benchmarks.

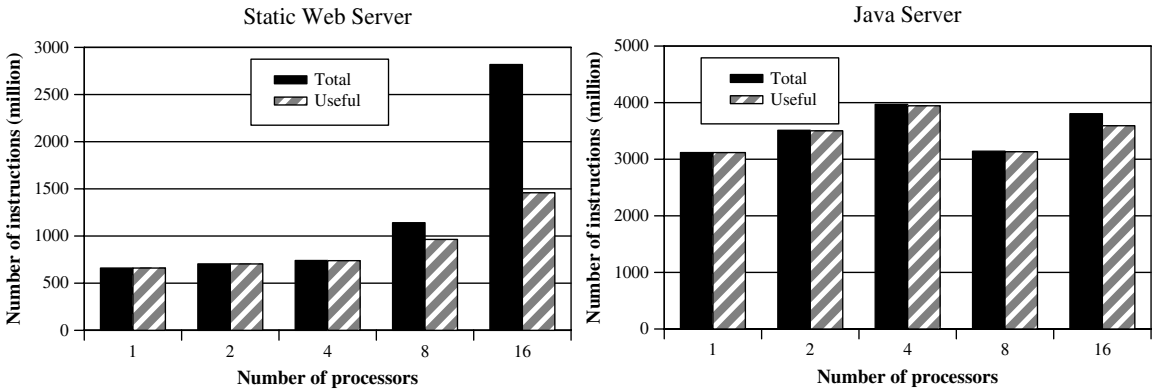


Figure 3-8. Total vs. useful instructions for the commercial workloads.

processors increases, while the number of useful instructions stays almost constant. These results verify that our spin detector accurately detects spinning instructions.

The commercial workloads do not have much spinning at the user level. As we see in Figure 3-8, the number of useful instructions does not stay constant as the number of processors increases for the two commercial workloads. This is because these workloads make significant use of the operating system. The amount of work performed by the operating system kernel changes as the number of processors varies (so does the number of threads in the system), thus causing the number of useful instructions to change. Nevertheless, the differences between the total and useful instructions in Figure 3-8 indicate that our spin detector detects spinning. Further inspection of our data shows that most of this spinning comes from flags and spin locks in the Solaris kernel dispatcher.

3.6.3 Useful IPC

In this section, we evaluate uIPC and verify that it tracks performance more accurately than IPC in both multiprocessor and microarchitecture performance studies. Note that the goal of our evaluation is not to make conclusions about the performance of any system or application; instead, our goal is to show how IPC can be misleading in the presence of spinning and how uIPC better tracks performance.

3.6.3.1 Multiprocessor Studies

In Figure 3-9, we plot performance (reciprocal of runtime), IPC, and uIPC as functions of the number of processors in the system for the SPEC OMP benchmarks. All metrics are normalized to their 1-processor values. IPC and uIPC are sums of these quantities across the processors. Figure 3-10 plots similar data for the SPLASH-2 benchmarks.

Figure 3-9 and Figure 3-10 show that uIPC is linearly correlated with performance, while IPC is not. When performance increases as we increase the number of processors, IPC often shows a drastic increase that is completely disproportionate to the improvement in performance. Thus if we use performance counters that do not distinguish spinning instructions, we would falsely assume that performance improves greatly even though the actual improvement is only moderate. For benchmarks such as `AMMP`, IPC increases even when the actual performance drops as the number of processors increases from four to eight. In contrast, for all the benchmarks, uIPC always accurately tracks performance.

Figure 3-11 plots performance (transactions per second), IPC, and uIPC for the commercial workloads. We see that IPC and uIPC are almost the same from one to eight processors. This is because these workloads use mostly blocking synchronization in the user-level code and do not have much spinning. However, for 16 processors, IPC is higher than uIPC because much more spinning occurs in the Solaris kernel when the processor count increases to 16 (as we have seen in Figure 3-8).

For the static web server workload in Figure 3-11, uIPC is not linear with performance. Moreover, from 8 to 16 processors, uIPC increases while actual performance drops. Our results show that when the number of processors increases from 4 to 8 and 16, this workload executes significantly more useful instructions in the OS kernel. This causes uIPC to increase even when performance decreases. The increased kernel instructions come from both the `httpd` threads of the Apache web server and the `sched` process (PID 0) of the Solaris kernel. As the number of processors increases, the number of `httpd` threads in the system also increases, which causes both `httpd` and `sched` to execute more instructions in the kernel dispatcher.

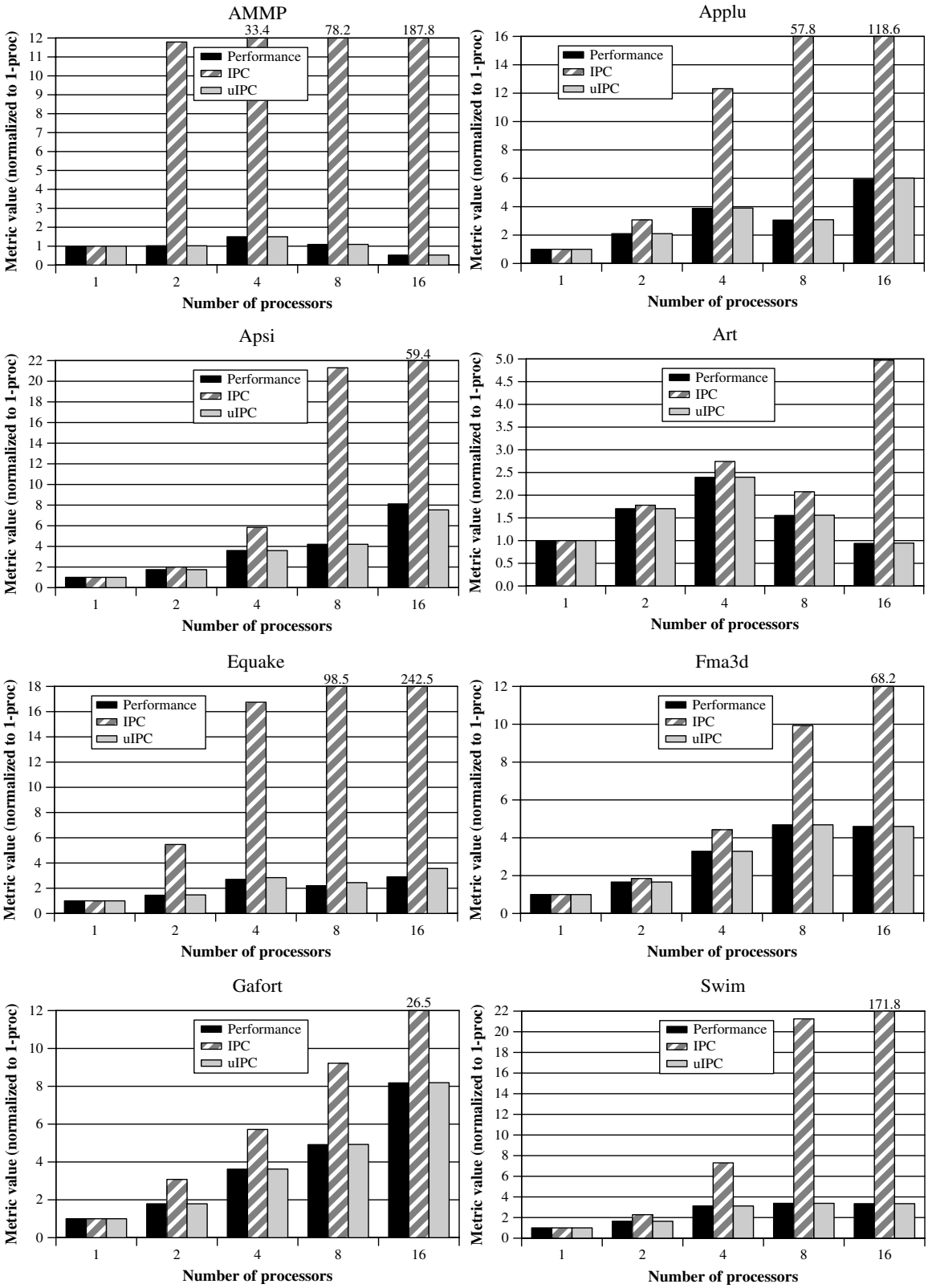


Figure 3-9. Useful IPC results for SPEC OMP benchmarks.

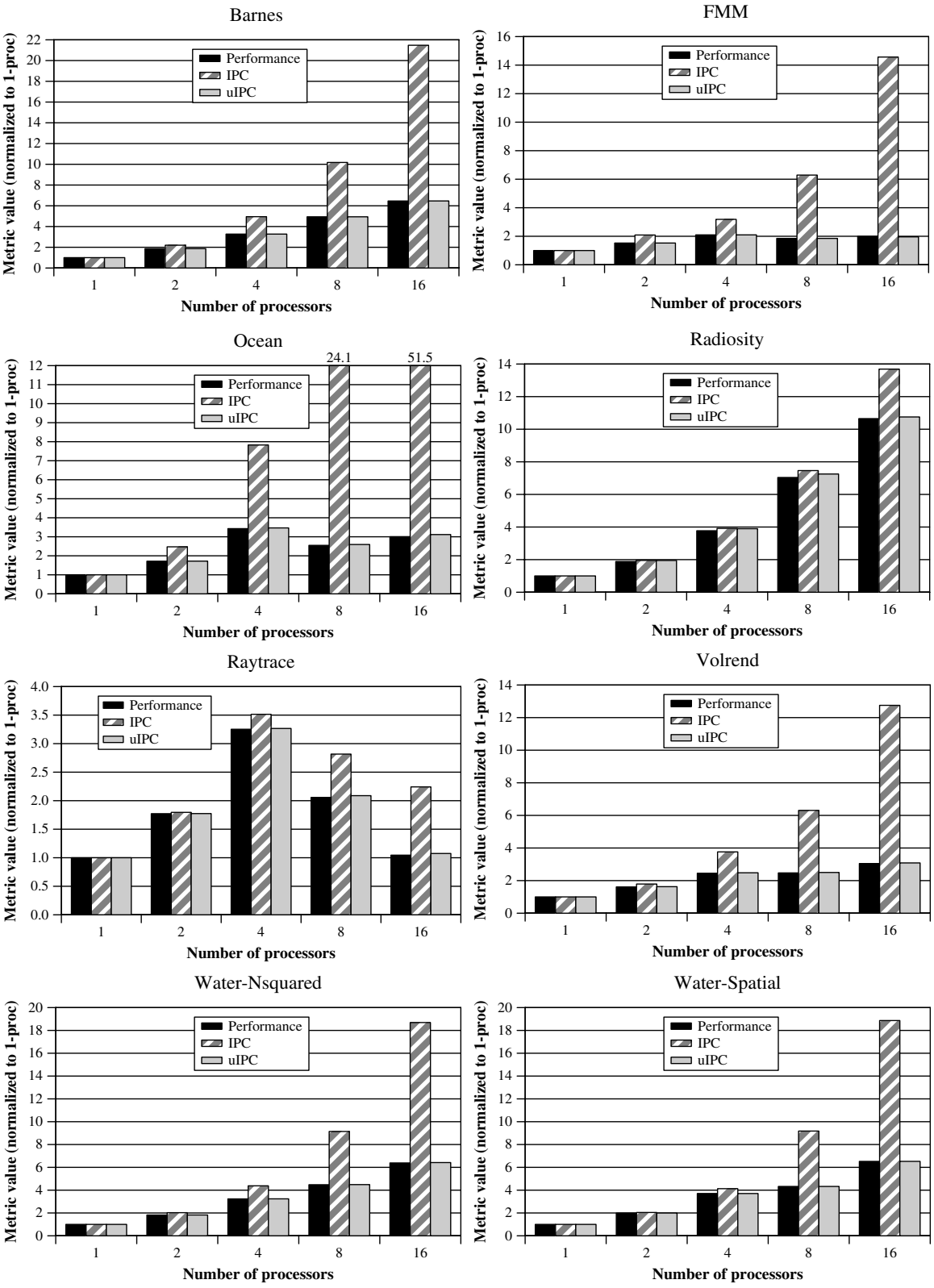


Figure 3-10. Useful IPC results for SPLASH-2 benchmarks.

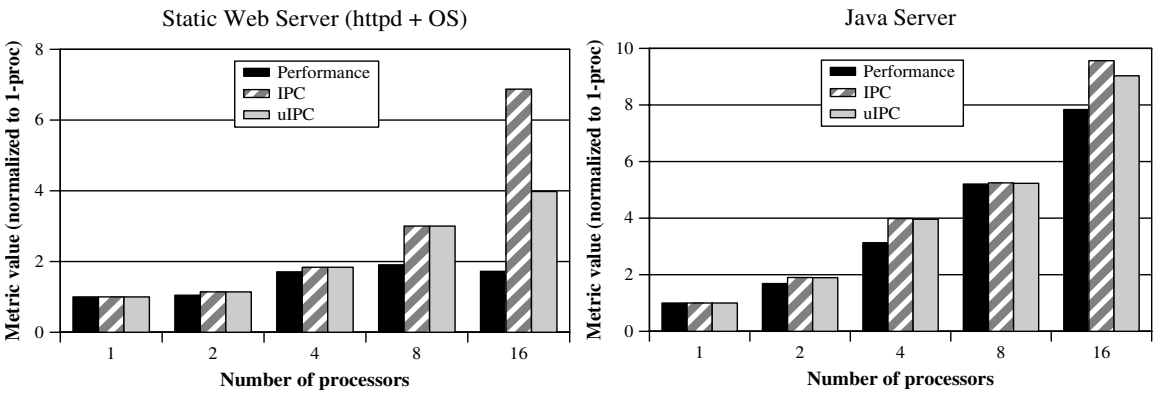
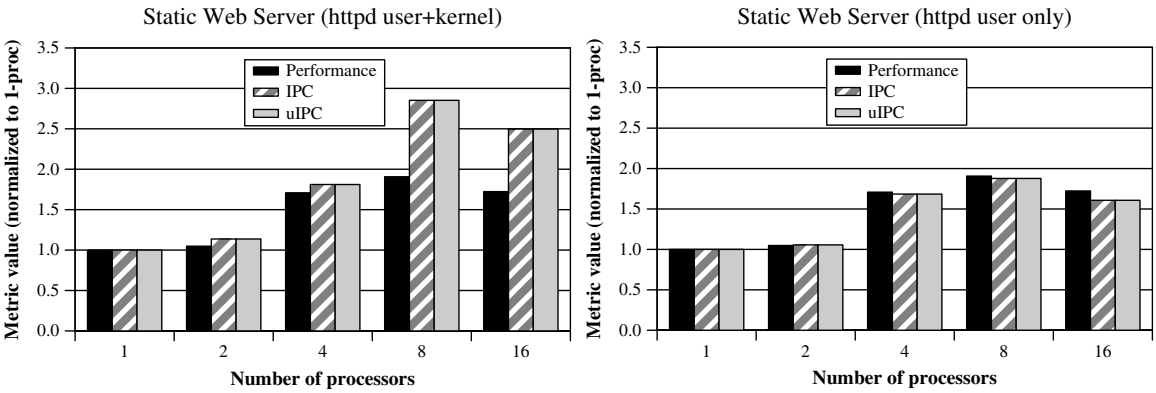


Figure 3-11. Useful IPC results for the commercial workloads.

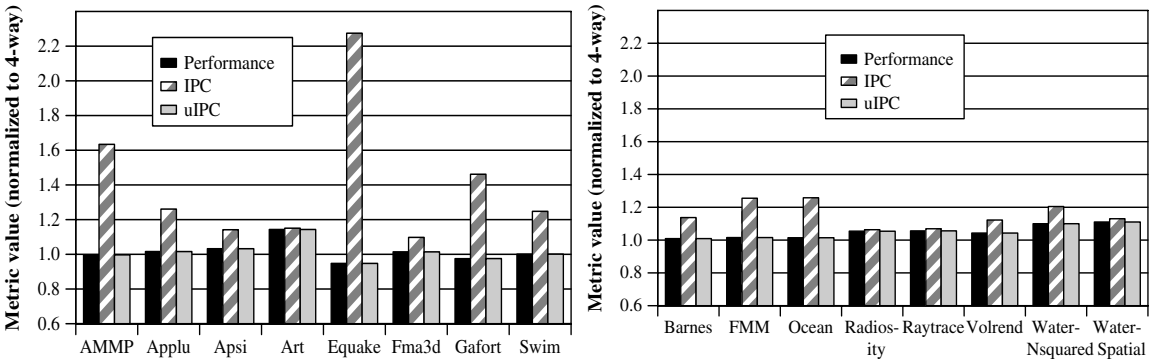
To isolate the performance of `httpd`, in Figure 3-12(a), we re-plot our results by counting only `httpd` instructions. We make three observations from this analysis. First, IPC and uIPC are almost equal in all the systems. This indicates that most spinning instructions in this workload are from the `sched` process. Second, uIPC now increases (or decreases) as performance increases (or decreases). Thus the `sched` process was the major cause for the mismatch between uIPC and performance we saw earlier. Finally, uIPC is, however, still not linearly correlated with performance. We hypothesized that this is due to the greater number of kernel instructions that `httpd` executes in the 8- and 16-processor systems than in the other systems. To verify our hypothesis, in Figure 3-12(b), we re-plot the results for only the user-level instructions in `httpd`. We see that both uIPC and IPC now achieve nearly linear correlation with performance. In practice, performance analysts can do similar analysis as above. Whether to aggregate the performance counters of different threads depends on which aspects of the system (e.g., OS or application) the analysts want to study.

For the Java server workload, uIPC reflects performance directly but not linearly. The non-linear correlation between uIPC and performance is due to the increased number of kernel instructions as the number of processors increases, similar to what we see in the static web server workload.



(a) User and kernel instructions of `httpd`. (b) User-level instructions only of `httpd`.

Figure 3-12. Useful IPC results for the `httpd` process.

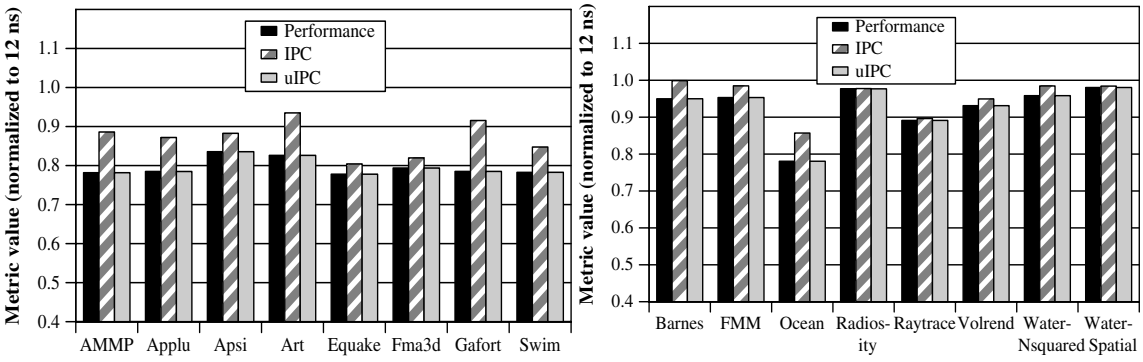


(a) SPEC OMP benchmarks. (b) SPLASH-2 benchmarks.

Figure 3-13. Useful IPC results for 8-way pipeline.

3.6.3.2 Microarchitecture Studies

For a system with a fixed number of processors, we show how a microarchitect can use uIPC. In our first study, for a 2-processor system, we study effects of processor pipeline width (including fetch, execute, and commit width). We set the pipeline width of each processor to four in one experiment and eight in another. In both experiments, the number of functional units of each type is proportional to the width. In Figure 3-13, we plot performance, IPC, and uIPC of the 8-way configuration for the SPEC OMP and SPLASH-2 benchmarks, with each metric value normalized to its corresponding value of the 4-way configuration. In our second study, for the same 2-processor system, we fix each processor's pipeline width at four and then study effects of changing the interconnection network



(a) SPEC OMP benchmarks. (b) SPLASH-2 benchmarks.

Figure 3-14. Useful IPC results for 24 ns network link latency.

link latency from 12 ns to 24 ns. Figure 3-14 shows the results of the 24 ns link latency configuration, normalized to the results of the12 ns configuration.

From these results, we see that uIPC is accurately correlated with performance, while IPC is not. Figure 3-13 shows that, for most benchmarks, performance and uIPC increase slightly as the pipeline width increases from four to eight. However, IPC increases much more because the 8-way processors issue and execute instructions faster than the 4-way, causing the 8-way system to execute a lot more spinning instructions. Two SPEC OMP benchmarks, `equake` and `gafort`, show slight slowdowns as the pipeline width increases from four to eight. These benchmarks have limited instruction-level parallelism, hence increasing the pipeline width beyond four does not improve performance. The slight slowdowns are likely due to subtle timing issues that cause the 8-way pipeline to have a less efficient instruction schedule than the 4-way pipeline. In Figure 3-14, as the network link latency increases from 12 to 24 ns, all metric values decrease, but IPC decreases by a smaller amount than performance and uIPC. This is because when the network link latency increases, threads wait for more time before they successfully obtain a lock. Thus the system executes more spinning instructions, causing IPC to drop at a ratio less than the actual performance and uIPC. These results show that, in contrast to [34], IPC is still not a valid metric for microarchitectural performance studies.

3.6.4 Scheduling

Our spin-then-yield mechanism discussed in Section 3.5.2.1 requires interrupt and scheduling support from the operating system. Since we do not have access to the Solaris source code, we emulate the OS support in our simulator as follows. When the hardware detects that a thread has spun for *N* iterations, we force the thread to jump to a routine, called `force_to_yield`, by setting the program counter of the thread's processor to the start of this routine. Within `force_to_yield`, the thread first saves its current PC, and then invokes the `yield` system call, thus blocking itself. When the `yield` system call returns (i.e., the blocked thread returns to the head of the run queue and the OS dispatcher selects it to run again), the thread jumps back to the saved PC, thus resuming the interrupted spin loop. The spin detection hardware also resets its counter for the spin loop iterations. In our implementation, we hardwire the `force_to_yield` routine to start at virtual address 0x10000 and span only a few bytes. This address range is unused by all Solaris programs and thus it does not affect the code or data of any program.

We evaluate the performance effects of spin-then-yield on a 2-processor system running simultaneously two SPEC OMP benchmarks, `art` and `gafort`. Each benchmark uses two OpenMP threads. We vary the number of iterations *N* that a thread is allowed to spin before yielding the processor from 5,000 to 25,000 with increments of 5,000. For each value of *N*, we run the workload until both benchmarks finish one iteration of their major loop (see Section 3.6.1), and measure performance as the reciprocal of the execution time. In Figure 3-15, for each value of *N*, we plot the system performance normalized to the performance of a system that does not implement spin-then-yield, i.e., *N* equals infinity. We see that spin-then-yield greatly improves performance. Among the different values of *N*, the system obtains the highest performance when $N = 10{,}000$ (16.4% speedup compared to not using spin-then-yield); beyond 10,000, the benefits of spin-then-yield drop gradually.

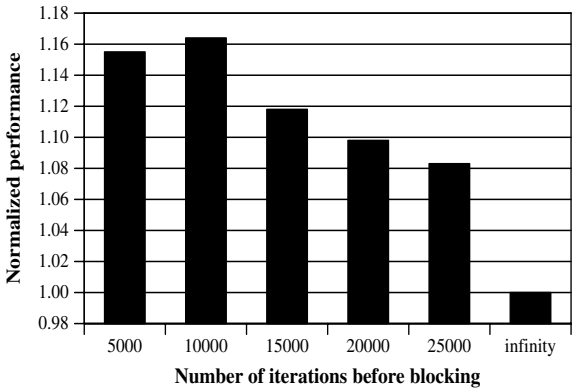


Figure 3-15. Results for spin-then-yield.

3.6.5 Livelock Detection

Since livelock does not occur in our workloads, we create a microbenchmark, called `circular lock`, to evaluate the livelock detection mechanism discussed in Section 3.5.3. The microbenchmark implements *T* threads, each of which holds a lock (Pthreads spin lock) while waiting for a lock held by another thread, and all the locks form a circular dependence chain. Since our livelock detection mechanism requires modifications to the OS, we evaluate it using Simics/x86 1.6.11 running Linux kernel 2.6.3. We add a system call to Linux, which allows a user program to specify a group of threads (PIDs) for livelock monitoring. We modify the kernel's context switch code to save and restore a thread's SDT and RUB.

We run `circular lock` on a simulated 4-processor system with *T* varying from two to sixteen. Our results show that we can detect livelock for all of these cases. In Figure 3-16, we plot the number of spinning threads in the system as a function of logical time for *T* = 4. We see that the number of spinning threads detected by our hardware increases as logical time progresses, and finally the hardware detects livelock when all threads spin simultaneously at logical time 50,495. Moreover, by running `contended lock` (see Section 3.6.2), we verify that our mechanism does not generate false alarms of livelock when it does not exist.

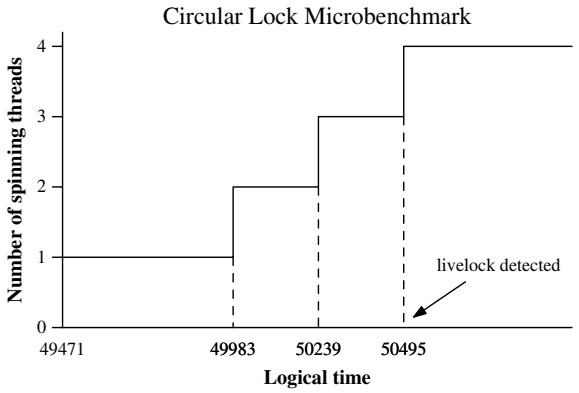


Figure 3-16. Results for livelock detection.

We also evaluate the context switch overhead introduced by saving and restoring the SDT and RUB. Since we do not yet have a detailed timing model for the Simics/x86 simulator, we measure the number of dynamic instructions, instead of time, for each context switch. We measure from the start of the `switch_to` function in the Linux scheduler to the execution of the first instruction of the next process scheduled to run. Over 1,000 context switches, an average Linux context switch executes 98 instructions, while spin detection only increases it to 116 instructions.

3.7 Related Work

We present related work on multithreaded system performance analysis, hardware synchronization support, and livelock detection.

3.7.1 Multithreaded System Performance Analysis

Redstone [48] studied spinning in the OS kernel and showed that studying spinning on SMT is important even though hardware blocking locks [57] may potentially eliminate spinning. Nayfeh et al. [43] showed that synchronization can cause problems for IPC when used to evaluate multiprocessor performance. Nevertheless, they found that IPC tracked performance in their experiments because their workloads did not have much spinning. Similar to us, Yamauchi et al. [64] used effective IPC, defined as the number of useful instructions completed per cycle, to evaluate a single chip multiprocessor. However, they

did not describe how they identified spinning instructions. Recently, Lepak et al. [34] systematically removed sources of non-determinism in multithreaded program executions to redeem IPC as a valid metric for multithreaded systems. However, they did not consider how spinning can negatively affect the accuracy of IPC.

There are also software performance analysis tools that can help programmers analyze spin lock performance. These tools often use code instrumentation [7, 41] or execution traces [62] to identify spins, or simple hardware event counters to estimate costs of spins [51], whereas our hardware spin detector frees programmers from this burden

3.7.2 Hardware Synchronization Support

Tullsen et al. [57] proposed hardware blocking locks to dynamically suspend threads that would otherwise spin. Keckler et al. [32] used register full/empty bits to enable threads waiting on shared data to stall rather than spin. McDowell et al. [40] proposed hardware-lock-then-block for synchronization, which is similar to our spin-then-yield mechanism. All these designs require modifications to the applications as well as special hardware support. In contrast, our mechanisms require no program semantic knowledge and do not need to modify the applications.

Intel IA-32 provides a HALT instruction for suspending a processor's execution and a PAUSE instruction for slowing down the execution of a spin loop to improve overall performance and save power [28]. The IBM POWER5 allows software to set lower scheduling priorities for spinning threads [29]. All these approaches burden programmers since they need to identify spin loops (possibly with the help of a performance analyzer [28]) in existing code, and then modify and recompile the code to utilize the hardware support. In contrast, our approach frees programmers from this burden by adding only modest hardware.

3.7.3 Livelock Detection

Liveness properties of multithreaded programs are conventionally proved statically using techniques such as temporal logic [46], model checking [27], and reachability analysis [11]. Recent work by Engler and Ashcraft [14] described a static tool to detect deadlocks in large

multithreaded systems.[5] Dynamically, indirect approaches are often used to infer potential livelock situations. For example, discarding packets due to queue overflow could imply livelock [42]. Compared to the indirect approaches, our hardware support can provide more accurate information about livelock without the need of knowing program semantics.

3.8 Conclusion

Synchronization is an integral part of multithreaded applications. The behavior of synchronization often directly affects how applications perform. Monitoring thread synchronization behavior enables multithreaded systems to understand application performance, and, based on this information, manage resources more efficiently.

Spinning and blocking are two common waiting mechanisms for synchronization. Spinning enables better performance because it does not involve the context switch overhead associated with blocking. However, spinning threads perform no useful work with respect to applications' computation, and waste resources such as processors and power.

To overcome the challenges that spinning poses, we have proposed efficient hardware to detect the common form of spinning during a thread's execution. We make three uses of this hardware. First, we develop performance counters for useful instructions that processors commit. We show that uIPC correlates with application performance more accurately than the conventional IPC metric. Second, we extend the spin detector to improve OS scheduling and power management. We show that our spin-then-yield mechanism can improve application performance by 16.4%. Third, we apply the spin detector to support dynamic detection of livelock/deadlock in multithreaded programs. Combined with OS support, our hardware successfully detects deadlocks in which all threads are spinning. Nonetheless, such deadlocks constitute only a small portion of the deadlocks that occur in practice—many deadlocks involve threads that are blocked in the OS kernel, waiting for events such as I/O. Thus, in the next chapter, we investigate how to enable multithreaded systems to monitor deadlocks involving blocked threads.

5. We discuss more related work on deadlock detection in Chapter 4.

4 Monitoring Thread Deadlocks

Deadlock can occur wherever multiple processes or threads interact. Monitoring deadlocks in a multithreaded system is essential for maintaining the system's functionality. Timely detection of deadlock and its cause allows the system not only to quickly resolve the error, but to reclaim resources involved in the deadlock. In the previous chapter, we studied hardware and OS support for detecting livelocks/deadlocks involving spinning threads. Since a large fraction of deadlocks in practice involve threads blocked in the OS kernel, in this chapter, we study an OS mechanism (requiring no special hardware) that enables multithreaded systems to detect deadlocks involving blocked threads.

We make the following contributions in this chapter:

1. We design a generic OS mechanism for detecting deadlocks in all user applications.
2. We present a specific Linux implementation of the mechanism.
3. We demonstrate that our mechanism can detect complex deadlocks, which, to the best of our knowledge, no existing tools can detect.

4.1 Introduction

A common problem in multithreaded applications is deadlock. A set of threads (or processes) is deadlocked if each thread is waiting for an event that only another thread in the set can cause. To address this problem, researchers have developed deadlock detection mechanisms. In practice, however, deadlock detection is often not performed in an effective manner. The drawbacks of the various existing approaches include restrictions on the deadlock-prone access patterns that can be handled and lack of information provided on the causes of deadlocks that arise. We classify existing approaches based on whether they are dynamic or static.

One common dynamic deadlock detection approach is to use timeouts. With timeouts, the system assumes a process to be deadlocked after it has been waiting for a shared resource longer than a certain amount of time. This approach is simple, but inaccurate because it cannot differentiate between processes that are deadlocked and processes that simply need a long time to acquire a resource. Even when they detect deadlock correctly, timeouts provide no information to the developer about why the deadlock occurred.

An alternative dynamic approach is based on graph modelling of process interactions. The "textbook" deadlock detection uses the general resource graph model [26], which models the state of a system as a directed graph. The nodes in the graph represent processes and resources, and the edges represent dependences among them. Given a general resource graph, we can detect deadlock by checking if the graph possesses certain properties (e.g., a cycle or a knot). The general resource graph model classifies resources into reusable and consumable. A reusable resource has a fixed number of units; one unit can be assigned to at most one process at a time. A consumable resource has no fixed total number of units; when a unit is assigned to a process, it ceases to exist. Only a process that the system designates as a producer of a consumable resource can produce units of the resource.

In practice, deadlock detection often assumes a simplified resource model: the system contains only reusable resources and there is only a single unit of every resource. This model makes deadlock detection simple to implement, but at the cost of detecting fewer types of deadlock. Under this model, a general resource graph takes a much simpler form as a wait-for-graph (WFG). The nodes in a WFG represent processes and the edges represent dependences between processes—there is an edge from node A to node B if process A is waiting for process B to release a resource. A cycle in a WFG indicates a deadlock. Constructing a WFG requires dynamically tracking the resources. This includes tracking the owner of each resource and the processes that are waiting for the resource at any time.

A common disadvantage of all dynamic techniques is that their analysis only considers control flow paths actually taken. Static deadlock detection (e.g., RacerX [14]) does not have this problem because it performs analysis on all possible control flow paths. However, these methods depend upon programmer specification of lock semantics and availability of

the entire code base. Considering all possible paths also forces static tools to face the issue of filtering out potentially large amounts of false positives.

In this chapter, we propose *Pulse*, an operating system mechanism that dynamically detects deadlock. Pulse is based on the general resource graph model. It uses high-level speculative execution [9, 19] to construct dependency information about processes that are blocked in the OS kernel. Our goal is to detect a wide variety of deadlock situations, including those that can and cannot be detected by existing techniques. However, our intent is not to replace existing techniques, but to increase the types of deadlock that developers can detect. Pulse is complementary to existing techniques; when developers use Pulse and the other tools together, they can obtain the best coverage of deadlocks.

Our implementation of Pulse focuses on detecting deadlocks due to bugs in application programs as opposed to bugs in kernel code. We have implemented Pulse in Linux kernel version 2.6.8.1. Our results show that Pulse can detect deadlock situations in incorrect solutions to the classical dining-philosophers and smokers problems, and a deadlock scenario in the Apache web server version 2.0.49.

Pulse runs as a daemon process and performs deadlock detection in the OS kernel when necessary. Pulse can be in one of three modes: *nap*, *monitor*, and *detection*. Initially, it is in the nap mode (i.e., the Pulse process sleeps in the kernel). Periodically, it awakens and enters the monitor mode. In this mode, Pulse checks if any process in the system has been asleep for a long time (how long is a tunable parameter). If none exists, Pulse returns to the nap mode. Otherwise, the sleeping processes might be deadlocked, and thus Pulse enters the detection mode. In this mode, Pulse identifies the events on which these long- sleeping processes are waiting (e.g., a lock being free or a pipe being non-empty). To discover how these long-sleeping processes depend on each other, Pulse forks each of them to create a *speculative process*. A speculative process first modifies its state such that it will unblock (e.g., by setting the status of the lock on which its parent is blocked to free in its own address space), and then executes ahead in its parent's program. To prevent a speculative process from changing the state of normal processes, Pulse leverages the copy-on-write mechanism in Unix fork and disallows a speculative process to perform I/O writes [19].

During the execution of a speculative process, Pulse records all the events it creates (e.g., releasing a lock or writing to a pipe). After a speculative process terminates, Pulse uses these events to match against the events awaited by the sleeping processes. A match between processes *A* and *B* indicates that if process *A* were not blocked, it would produce an event that unblocks process *B*. Based on this information, Pulse constructs a general resource graph in which the nodes denote the sleeping processes and the events on which they are waiting, and the edges denote the dependences Pulse discovers. Pulse detects potential deadlock by checking if this graph contains cycles. If it detects a cycle, it also prints out the entire graph to help programmers identify the causes of the deadlock.

Pulse differs from existing dynamic techniques in that it discovers dependency information by looking into the future, while existing techniques rely on past information (e.g., which process owns a lock and which is waiting for it) to derive the dependences. Most existing techniques are WFG-based and they commonly restrict themselves to only detect deadlocks involving single-unit reusable resources such as locks. Fundamental in this resource model is the assumption that a busy resource can only be freed by the owner that currently holds the resource. This assumption makes it possible to discover process dependences based on information collected from the past, which includes the identities of the owners and waiters of each resource. However, consumable resources have no notion of ownership, i.e., no process can hold a consumable resource. For example, a semaphore that implements synchronization (instead of mutual exclusion as a lock) has no owners—any process can potentially perform an *up* operation on the semaphore and unblock another waiting process. In contrast, Pulse makes no assumption about the resource model. The ability to look into the future allows Pulse to directly discover what events a process can produce, as opposed to reasoning about it based on ownership information.

The ability to detect deadlocks that involve consumable resources also distinguishes Pulse from existing static approaches, such as RacerX [14]. In Figure 4-1, we use a Venn diagram to illustrate the types of deadlock that WFG-based techniques, static schemes, and Pulse can detect. We draw the set of deadlocks detectable by the static schemes larger than the other sets, because static schemes detect deadlocks along all possible control flow

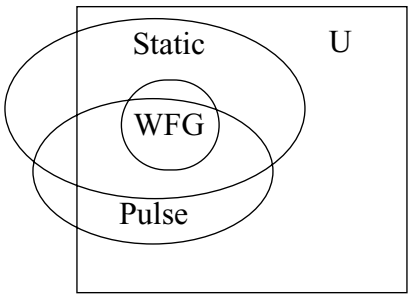


Figure 4-1. Venn diagram for deadlocks. The three circles represent deadlocks detectable by WFG-based techniques, static schemes, and Pulse. U is the universal set of all deadlocks. The regions outside the rectangle represent false positives.

paths, while WFG-based techniques and Pulse only detect deadlocks that occur during execution. There is one particular type of deadlock that Pulse cannot detect yet WFG-based techniques and static schemes can. This happens when Pulse cannot discover all the dependences from running ahead in the program. For example, if a sloppy programmer forgets an unlock operation that can unblock a waiting process, then Pulse will not see this future event and thus will not be able to identify a cycle of dependences.

There are also types of deadlock that Pulse can detect but the other approaches cannot. These include deadlocks involving consumable resources (e.g., RacerX ignores deadlocks with synchronization semaphores) and variable aliasing (e.g., different variables may point to the same lock, which may not be detectable with static analysis). On the other hand, both static detection and Pulse can generate false positives, i.e., detect deadlocks that do not really exist. Static detection can generate more false positives because it cannot completely filter out control flow paths that are never taken in real execution. However, Pulse does create a unique set of false positives that other techniques do not encounter, which we discuss in Section 4.2.5.

Pulse can be viewed as complementary to the existing deadlock detection techniques. If developers use Pulse and the other techniques together, they can obtain the best coverage of deadlocks.

The remainder of this chapter is organized as follows. In Section 4.5, we discuss related work. Section 4.2 describes how Pulse works and its limitations. We discuss how we implement Pulse in Section 4.3. In Section 4.4, we demonstrate that Pulse can detect deadlock

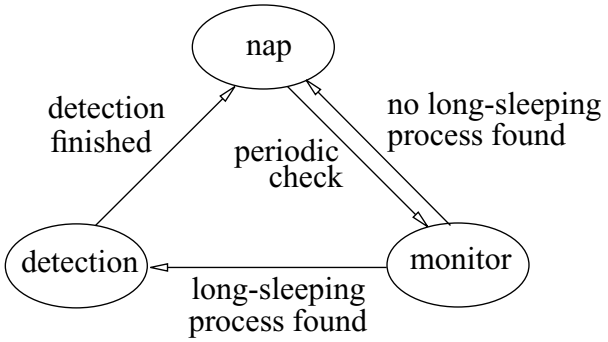


Figure 4-2. Pulse state transition diagram.

situations in the classical dining-philosophers and smokers problems, as well as in the Apache web server. Existing techniques, including WFG-based schemes and RacerX, can only detect deadlock in the dining-philosophers problem, but cannot detect the other two deadlock situations. Finally, we conclude in Section 4.6.

4.2 Deadlock Detection with Pulse

In this section, we present an overview of Pulse (Section 4.2.1) and then describe its design in detail (Section 4.2.2–Section 4.2.4). Finally, we discuss how we can extend Pulse and its current limitations (Section 4.2.5). Section 4.3 presents our Linux implementation of Pulse.

4.2.1 Overview

Pulse exploits the observation that if a set of processes is deadlocked, the processes often sleep within the OS kernel, each waiting for events that can only be produced by another process in the set. Thus, when Pulse sees a set of processes blocked for a long time, it considers that a deadlock might have occurred.

Pulse runs as a daemon process that can be in one of three modes: nap, monitor, and detection. Figure 4-2 is a state diagram that shows how Pulse transitions between these modes. For most of the time, Pulse is in the nap mode in which it sleeps in the OS kernel. Periodically, it awakens and enters the monitor mode to check if any process in the system has been asleep for a threshold amount of time, where the threshold value is a tunable

Process P_1	**Process P_2**
lock(A)	lock(B)
lock(B)	lock(A)
unlock(A)	unlock(B)

Figure 4-3. A circular lock example.

parameter (e.g., five minutes). If no such processes exist, Pulse returns to the nap mode, thus incurring low overhead. Otherwise, Pulse enters the detection mode and performs deadlock detection for these sleeping processes. We show in Section 4.4.4 that a threshold as low as one minute introduces negligible performance overhead. However, if the threshold is too large, Pulse may miss certain deadlocks that can be broken by mechanisms such as timeouts, as we explain in Section 4.2.5.

Pulse uses the general resource graph model [26] to detect deadlock. To illustrate the idea, we use the code in Figure 4-3 as a running example. Suppose that the two processes execute their second `lock` statements simultaneously. Thus process P_1 blocks on lock B and process P_2 blocks on lock A, and they deadlock. When Pulse detects that processes P_1 and P_2 have been asleep longer than the threshold, it enters the detection mode.

In the detection mode, Pulse identifies the events on which the sleeping processes are waiting (Section 4.2.2). In our example, process P_1 is waiting for lock B to be free and process P_2 is waiting for lock A to be free. For each sleeping process, Pulse constructs a *process node* in a general resource graph. For each event it identifies, Pulse constructs an *event node*. We use the term event node instead of resource node [26] to emphasize that a process often waits for an event involving a resource (e.g., a lock being free), as opposed to the resource itself (e.g., the lock). Pulse also constructs a *request edge,* directed from a process node to an event node, if the process is waiting for that event.

To discover dependences between the sleeping processes, Pulse uses speculative execution. For each sleeping process, Pulse forks a speculative process that executes ahead in its parent's program. Speculative execution allows Pulse to discover the events that a blocked process would produce if it were not blocked. In our example, Pulse discovers that

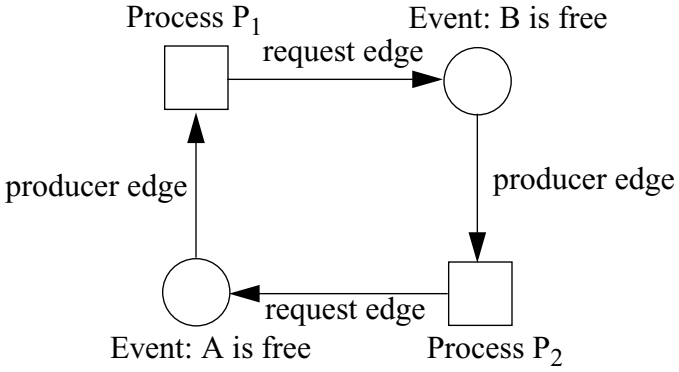


Figure 4-4. Resource graph for the circular lock example.

process P_1 would unlock A and process P_2 would unlock B, if they were not blocked. Thus Pulse constructs a *producer edge*, directed from the event that process P_1 creates, i.e., lock A becoming free, to process P_1. Similarly, Pulse constructs a producer edge from the event node that P_2 creates, i.e., lock B being free, to process P_2.

Figure 4-4 shows the graph that Pulse finally obtains. We use squares to denote process nodes and circles to denote event nodes. Since this graph contains a cycle, Pulse outputs that a potential deadlock exists. It also outputs the entire graph to help developers debug the deadlock.

The general resource graph model allows us to detect whether a potential deadlock exists, and if so, which processes are involved in the deadlock and why they are deadlocked. To construct a general resource graph, our design needs to address the following questions:

1. How do we construct nodes in the graph? In other words, how do we identify the processes and events involved in a potential deadlock? (Section 4.2.2)
2. How do we construct edges in the graph? That is, how do we identify the dependences among the processes and events? (Section 4.2.3)

In the rest of this section, we describe how our design addresses these questions in detail.

4.2.2 Constructing nodes

When Pulse enters the detection mode, it has already identified a set of processes that have been asleep for a long time. These processes are potentially deadlocked; thus, they constitute the process nodes in the general resource graph.

The events on which these processes are blocked constitute the event nodes in the graph. To identify the events for which a sleeping process is waiting, Pulse requires modifications to all blocking system calls. Within these calls, we add new code to record the events for which the caller process is waiting. Pulse characterizes an event by a (*resource*, *condition*) pair. The *resource* field identifies the resource on which the process is blocked (e.g., a lock or a pipe). The *condition* field describes the condition about the resource for which the process is waiting (e.g., the lock being free, or the pipe being non-empty).

4.2.3 Constructing edges

Pulse models all resources as consumable resources. This resource model enables Pulse to detect deadlocks involving more than just single-unit reusable resources. However, it also causes Pulse to generate certain false positives, as we will see in Section 4.2.5.

There are two sets of edges that Pulse needs to construct in a graph: request edges and producer edges. Pulse constructs the request edges at the same time when it constructs the event nodes. As discussed in Section 4.2.2, when Pulse identifies a sleeping process and its awaited events, it can construct a request edge from the process to each event for which it is waiting.

To construct the producer edges, Pulse uses speculative execution. For each sleeping process, Pulse forks a copy of the process, called a speculative process. The speculative process first creates the events awaited by its parent process, if this does not affect any other process. For example, if the parent is blocked on a busy lock, then the speculative process can set the lock variable (often a user-space memory location) to free within its own address space, not affecting any other process. On the other hand, if the parent has been waiting for an I/O, the speculative process cannot create the awaited event because it could affect the

state of other processes. Regardless of whether Pulse creates the events or not, its goal here is to enable the speculative process to unblock and thus run ahead in its program. The next step for the speculative process is to return from the system call that caused its parent to sleep, pretending the call was successful, and then to resume its parent's user-level code after the blocking system call, all within the speculative process's own context (i.e., the parent is unchanged and it continues to sleep).

The execution of a speculative process should be safe, i.e., it should not modify the state of any other process. The Unix fork mechanism implements copy-on-write, which naturally protects a speculative process from modifying the user-space memory state of its parent (and other processes). Some systems support forking a process (or thread) that shares the same address space with the parent. Pulse avoids using fork in such a way and it always invokes fork with copy-on-write enabled. Similar to Fraser and Chang [19], Pulse does not allow a speculative process to write to the file system (thus no I/O writes) or deliver a signal to any other process. In Section 4.3, we discuss in detail the extra measures that our implementation takes to ensure the safety of the kernel state. These safety measures, however, may cause Pulse to produce false positives and/or false negatives of deadlock, as we explain next.

During the execution of a speculative process, Pulse records all the events produced by the process that could potentially unblock other processes. This requires modifying the system calls that counterpart the blocking system calls. For example, a write system call can block when trying to write to a full pipe (i.e., the pipe buffer is full), and can be unblocked only if a process reads data from the pipe. Thus, a pipe read system call counterparts a blocking pipe write system call. We modify all system calls that can unblock a process such that, if called by a speculative process, they record the resources being manipulated and the conditions being produced by the speculative process. As for the pipe example, the modified read system call would record a unique resource identifier for the pipe and some indicator representing that the pipe is being read. For each speculative process, Pulse maintains an *event buffer* that records the events the process produces during its execution. A speculative process adds an event to its event buffer only if the event is not already in the

buffer. If the buffer is full, the process ignores the event (i.e., does not add it to the buffer). As we see in Section 4.4, a buffer of ten events is sufficient for all of our experiments.

A speculative process terminates if one of the following conditions is true:

1. It exits normally.
2. Its event buffer is full.
3. *T* seconds have passed since the creation of the speculative process, where *T* is adjustable by the user.

After all the speculative processes terminate, Pulse matches their produced events against the events awaited by the sleeping (parent) processes. If the speculative version of process *A* produces an event that process *B* is waiting for, then Pulse constructs a producer edge from this event to process *A*. A speculative process could produce events that are not awaited by any sleeping process. We do not include these events in the graph, because this reduces the graph size and does not affect the correctness of deadlock detection.

4.2.4 Putting it all together

The nodes and edges that Pulse constructs collectively form a general resource graph. If the graph contains cycles, Pulse outputs that a potential deadlock exists. To facilitate debugging, Pulse also prints out the entire resource graph. It is also possible to perform symbol table lookups such that the application programmer can map the graph to specific points in the code. Based on all this information and knowledge of the applications, the programmer could easily verify whether a deadlock indeed exists, and, if so, identify its cause.

Pulse requires OS kernel support (e.g., for speculative execution). Compared to user-space solutions, Pulse provides a general solution for all applications, thus freeing programmers from the burden of designing deadlock detection for each individual application. On the other hand, for applications that already have some deadlock detection built-in (e.g., database management systems), they can use Pulse together with their own mechanisms to obtain the best coverage of deadlock.

4.2.5 Discussion

In this section, we first briefly describe our design plans for handling deadlocks involving spinning processes and deadlocks due to kernel bugs, but we leave a full evaluation of these designs as future work. We then discuss in detail the limitations of Pulse.

Kernel deadlocks. Applying Pulse to detect deadlocks due to kernel bugs is difficult, because allowing processes to speculatively execute within the kernel can cause unwanted changes to kernel structures and even crash the system. However, with the help of virtual machine technologies, such as VMware [5] or Xen [61], we could perform speculative execution on different virtual machines, making Pulse possibly applicable to detecting kernel deadlocks.

Limitations of Pulse. Pulse can output false positives, i.e., deadlocks that do not actually exist. There are three reasons why false positives can occur. First, because the execution of a speculative process must be safe, it cannot perform potentially unsafe operations, such as writing to a file. This could cause a speculative process to execute unrealistic program paths, making Pulse obtain incorrect dependences. Such false positives also exist in static detection tools because they often cannot identify unrealistic program paths statically.

Second, as we discussed in Section 4.2.3, Pulse models all resources as consumable resources. Thus it could construct more than one producer edge for resources (e.g., locks) that can be held and freed by only a single owner. The extra edges could cause Pulse to detect cycles that do not actually exist. Such false positives cannot occur in existing static and dynamic detection tools because they model only reusable resources.

Third, for systems with resources more complex than single-unit reusable resources, a cycle in a general resource graph is only a necessary, but not sufficient condition for deadlock [26]. For example, Pulse may detect a cycle when multiple processes block on a set of synchronization semaphores. However, a new process may later enter the system and perform an *up* operation on one of the semaphores, thus breaking the "deadlock". Such false positives are unique for Pulse; all the existing approaches do not consider resources more complex than single-unit reusable resources and thus do not have such false positives.

There are two types of deadlock that Pulse cannot detect (i.e., false negatives). First, some applications employ a timeout mechanism on operations that take too much time. For example, as we will see in Section 4.4, Pulse can detect a deadlock scenario due to cyclic pipe access dependences in the Apache web server. However, Apache can abort the pipe operations after they take longer than five minutes. The timeout effectively breaks the deadlock, although it also silently fails the client's request without providing any information about what has happened. Pulse could miss such deadlocks if its threshold value for entering the monitor mode is too large. However, it can avoid such false negatives by lowering the threshold, possibly at the cost of impacting application performance.

The second type of false negative is due to Pulse's reliance on the future events it discovers. Pulse is able to detect deadlock because it can discover the future events that the sleeping processes could produce if awakened. However, if such events are unavailable, Pulse will not detect the deadlock. There are four scenarios in which the future events can be unavailable.

1. Such events do not exist in the application program. For example, the programmer forgets the unlock statements in the code in Figure 4-3.
2. Speculative processes do not run long enough to discover the events.
3. The event buffers fill up before speculative processes see the relevant events.
4. Speculative processes may execute unrealistic program paths that do not manifest the relevant events.

The first scenario is a fundamental limitation of Pulse. However, the second and third scenarios are not fundamental limitations; they are avoidable by increasing the run time and event buffer sizes of speculative processes. The reason for the fourth scenario is that Pulse does not allow speculative processes to execute potentially unsafe system calls. This scenario may be considered as a fundamental limitation of Pulse, if our implementation chooses to run speculative processes on the same operating system on which the normal processes run (which is what we do in this thesis). However, we may avoid some cases of

this scenario if each speculative process can run on a different operating system, e.g., using VMware [5] or Xen [61].

4.3 Implementation

In this section, we discuss how we implement the design described in Section 4.2. We implement Pulse by modifying Linux kernel version 2.6.8.1. We add a new system call that allows us to invoke Pulse from the user level.

4.3.1 Constructing process nodes

To construct the process nodes, Pulse needs to identify long-sleeping processes. We add a flag, `was_asleep`, to each process's `task_struct`, which is false when the OS creates the process. When Pulse enters the monitor mode, it scans the system for processes that satisfy the following three conditions:

- The process is asleep.
- The process was put to sleep by one of our modified system calls (see Section 4.3.2).
- The process's `was_asleep` flag is true, which indicates that the process was asleep when Pulse checked it last time.

If a process satisfies the first two conditions, but not the last one, Pulse sets the process's `was_asleep` flag to true. The Linux scheduler resets this flag to false when it switches this process to run, i.e., when it awakens the process. For each process that satisfies all the three conditions, Pulse constructs a process node for it.

Some system daemon processes (e.g., automount) sleep in the kernel for a long time. If they satisfy the above conditions, Pulse will construct process nodes for them. Alternatively, we could implement a mechanism to disable the construction of nodes for these processes, if the user of Pulse (e.g., a system administrator) believes that these processes do not deadlock.

4.3.2 Constructing event nodes

To identify the events for which a sleeping process is waiting, we need to modify all system calls that can block. For the purposes of this thesis, we have modified only three blocking system calls: `futex`, `write`, and `poll`. In each call, before putting the caller to sleep, we add code to construct the following two lists and store them in a structure pointed to by the caller process.

- A resource list, ($resource_1$, $resource_2$, …), where $resource_i$ is an integer that uniquely identifies a resource being manipulated by the system call.
- A condition list, (<op_1, val_1>, <op_2, val_2>, …), where the pair <op_i, val_i> encodes the condition about $resource_i$ that can unblock the caller process.

The `futex` and `write` system calls involve only one resource and thus their resource and condition lists consist of only one element. The `poll` system call, however, can operate on multiple file descriptors. Thus it may record more than one resource and condition. We now describe how we modify these three system calls.

Futex. Futex stands for fast user-space mutex. The `futex` system call is the basis of various synchronization primitives in the Native POSIX Thread Library (NPTL), which has been integrated in the recent versions of glibc. For the purposes of demonstrating Pulse, we consider NPTL's mutex and semaphore primitives in glibc 2.3.2. A thread acquiring a busy mutex or semaphore blocks via the `futex` system call. Each NPTL mutex or semaphore corresponds to a user-space memory address, which the caller thread passes to `futex` as an argument. Thus, in `futex` (before the caller is about to sleep), we add code to record this memory address, which uniquely identifies the resource on which the caller thread is blocked. The condition to unblock the caller, however, depends on the context in which the caller thread invokes the `futex` system call, and thus requires modifications to the NPTL library. Note that the applications using the library require no modification.

For the NPTL function that acquires a mutex lock, (`pthread_mutex_lock`), we pass <*equal-to*, 0> as two extra arguments to the `futex` system call. They signify that the

condition to unblock the caller thread is when the mutex value is zero (i.e., the mutex is free).

For the NPTL function that does a semaphore *down* operation (`sem_wait`), we modify it to pass <*greater-than*, 0> as two extra arguments to the `futex` system call. They signify that the condition to unblock the caller is when the semaphore value is greater than zero.

Write. Linux's `write` system call is a generic interface to a wide range of file systems. Our implementation currently considers only writes to pipes, which are implemented in the `pipe_writev` function. For a blocking pipe write, the caller process blocks if the pipe buffer is full. We add code in function `pipe_writev` (before the caller is about to sleep) to record the address of the pipe's inode structure, which uniquely identifies the pipe resource. The condition under which the caller unblocks is when another process reads data from the pipe. We use the pair <*and*, POLLOUT | POLLWRNORM> to encode this condition. The fields POLLOUT and POLLWRNORM are kernel-defined bit masks, denoting that writing now becomes unblocked. As we will see in Section 4.3.3, we also record similar bit masks in system calls that can unblock the write to denote the events produced by those calls. The *and* operator here helps match the blocked process with the process that can potentially unblock it: if the logical *and* of POLLOUT | POLLWRNORM and the bit mask produced by another process is true, then that process can unblock this process.

Poll. The `poll` system call takes a list of file descriptors and events as arguments. If none of the events has occurred for any of the file descriptors, the caller blocks, waiting for one of these events to occur. The events are represented by bit masks, similar to the ones we discussed above. We add code in the `poll` system call (before the caller is about to sleep) to construct a resource list. Each resource is a file descriptor, denoted by the address of its inode structure. We also record a condition list. Each condition is denoted by a <*op*, *val*> pair, where *op* equals *and*, similar to what we do for pipe writes. The *val* field is simply the event bit mask that the caller passed to the `poll` system call.

Based on the information recorded by the modified system calls, Pulse can identify what events a sleeping process is waiting for and create the event nodes for them. The three

modified system calls allow us to demonstrate the ability of Pulse in this thesis. In general, the same approach can be applied to modify other blocking system calls.

4.3.3 Constructing edges

As we discussed in Section 4.2.3, Pulse can construct request edges at the same time when it constructs the process and event nodes. Thus, in this section, we focus on constructing producer edges via speculative execution. We explain our implementation by following the flow of speculative execution in time.

Creating speculative processes. After Pulse identifies the long-sleeping processes, it forks a speculative process for each of them. The Unix fork mechanism allows a speculative process to run in its own address space, not affecting the state of its parent. The difficulty, however, is that the existing fork implementation assumes its caller process to be the same as the one to be forked. What we need, instead, is a function that can fork an arbitrary given process. Our first attempt was to write such a function by modifying Linux's existing `do_fork` function such that it took one more argument: a `task_struct` pointer to the process to be forked. Soon we found that this approach would require us to rewrite many existing kernel functions. First, they all needed to take this one extra argument. Second, they all needed to be modified to operate on data structures of the specified process instead of those of the caller process. To avoid this tedious work, we designed an in-kernel fork mechanism that allows us to use the existing `do_fork` function with only slight modifications.

Figure 4-5 illustrates our in-kernel fork design. To fork process *P*, Pulse first switches *P* in (thus switching itself out), but to a predefined function. Within this function, Process *P* calls `do_fork` to create a copy of itself, *P'*. Finally, it switches the Pulse process back in, which then resumes execution from where it was left off. Similar to ordinary processes, the speculative process *P'* participates in Linux's normal scheduling.

To implement this design, we wrote a simple context switch function, similar to Linux's context switch function. To fork a process *P*, Pulse calls our context switch function, which

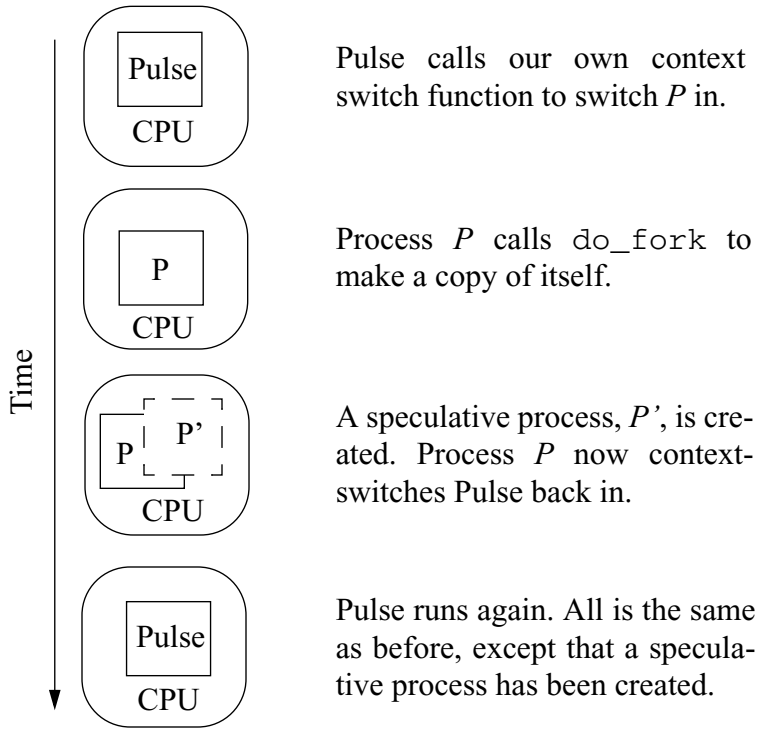


Figure 4-5. Illustration of in-kernel fork.

switches process *P* in by saving the memory and register state of the Pulse process and loading the state of process *P*. With a normal context switch, process *P* would then resume its execution from the point where it was suspended previously. However, in our context switch function, we force it to enter an `in_kernel_fork` function that we added to the kernel. Within this function, process *P* calls `do_fork` to create a speculative process *P'*. It then calls our context switch function to switch the Pulse process in and switch itself out such that it goes back to sleep again.

Certain fields, such as the instruction and stack pointers, in the `task_struct` of process *P* may be changed by the `do_fork` call. Thus, after being switched back in, Pulse restores these fields. Our goal is to ensure that the parent process stays exactly the same before and after the creation of a speculative (child) process. Thus, we modify `do_fork` such that it does not link a speculative process to the children list of its parent. The speculative process does keep a pointer back to its parent, because it needs to know to which sleeping process it corresponds. To allow the speculative process to exit independently of

its parent, we reset its parent to be the init process in the `do_exit` function. In `do_fork`, we also initialize the speculative process such that it does not send any signal to its parent when it exits. As such, the speculative process is completely independent of its parent and does not affect the parent in any way.

Starting speculative processes. After Pulse creates a speculative process, the process participates in normal process scheduling. The normal fork semantics would make the speculative process resume the code in which the parent was suspended. Thus, the speculative process would resume the system call that blocked the parent process previously. However, allowing the speculative process to execute in the kernel freely may cause changes to important kernel structures (e.g., the semaphore protecting a pipe's inode). Such changes should never occur because they could affect the normal execution of other processes, violating the safety property. To solve this problem, we force all speculative processes to execute a common function, `ret_from_spec_fork`, when the OS schedules them to run the first time after creation. Pulse achieves this effect by setting the initial program counter (the *eip* register) value of each speculative process to be the address of this common function when it creates the speculative process.

In `ret_from_spec_fork`, a speculative process first creates the event awaited by its parent as follows. The speculative process checks what events its parent is waiting for by looking up the resource and condition lists maintained for the parent. For example, if the parent is blocked in a `futex` system call, then it must be waiting for the data at a user-space address (*resource*) to become equal to or greater than (*op*) a certain value (*val*). If *op* is *equal-to*, the speculative process writes *val* to the address identified by *resource* within its own address space. If *op* is *greater-than*, it writes *val* + 1 to that address. In fact, for the latter case, any value greater than *val* would be fine since the parent is not waiting for a specific value. However, it is possible that different values may have different meanings. For example, a semaphore's value often represents how many processes are allowed to enter a critical section simultaneously. Thus, our choice of setting the value to *val* + 1 is only heuristic; the parent process may indeed expect a different value at the given address, although it did not explicitly say so when it invoked the `futex` system call.

If the parent process is blocked in a pipe `write` or `poll` system call, the speculative process is not allowed to create the events awaited by the parent. This is because creating the events requires doing I/O on a pipe, which can affect normal processes that access the pipe.

Regardless of whether the events are created or not, the `ret_from_spec_fork` function then forces the speculative process to exit the system call in which its parent is blocked. This is done by jumping to Linux's `syscall_exit` routine and returning the value that represents success for the corresponding system call. Thus, after returning to the user-level code, the application program will have the illusion that the blocking system call has returned successfully. The speculative process then runs ahead in the program.

Recording events. During execution, a speculative process records all of the events that can awaken a sleeping process. Since our implementation considers only three blocking system calls, we only modify the counterpart system calls of these three calls. Similar to recording events for which a sleeping process waits, we record the events that a speculative process produces as resource and condition lists too. For the system calls we modify, these lists happen to both have only one element.

A process blocked in the `futex` system call can unblock only if another process calls `futex` with a FUTEX_WAKE argument. Thus the `futex` system call is the counterpart of itself. We add code in `futex` such that, when called by a speculative process to perform a wakeup, it records this event. We modify the corresponding NPTL library functions, such as mutex unlock (`pthread_mutex_unlock`) and semaphore *up* (`sem_post`), such that they pass the necessary information to allow `futex` to fill in the values for the *resource*, *op*, and *val* fields, which characterize the wakeup event produced by the speculative process. For example, if the speculative process invokes `futex` from `pthread_mutex_unlock`, it records that this process is producing an event, that is, the memory location of the mutex (*resource*) now obtains a new value (*val*). The *op* field is set to be the same as what is used for the counterpart system call (in this case, *equal-to*).

A process blocked in a pipe write system call can be unblocked if another process reads the pipe. Thus we add code in the `read` system call to record the read event. When called

by a speculative process to read a pipe, the `read` system call records *resource* to be the address of the pipe's inode structure, *op* to be the same as what is used for the corresponding blocking pipe write call, i.e., *and*, and *val* to be POLLOUT | POLLWRNORM, which are exactly the bit masks for which the blocking write call is waiting.

For the `poll` system call, we modify two counterpart system calls: `write` and `writev`. For both of them, we record *resource* to be the address of the pipe's inode structure, *op* to be *and*, and *val* to be POLLIN | POLLRDNORM, which are kernel-defined macros representing that the pipe has data available to read.

To ensure safety, all of these system calls return immediately with a success code after they record the produced events. Thus, a speculative process calling these system calls does not really perform the wakeup and read/write operations that these calls normally do. Similar to Fraser and Chang [19], we modify all potentially unsafe system calls such that a speculative process returns immediately when entering these calls. In this way, any state change made by a speculative process is contained within itself, not affecting any other process.

Constructing producer edges. A speculative process terminates according to the conditions in Section 4.2.3. We limit the lifetime of a speculative process to be less than one second. After all speculative processes terminate, Pulse matches their produced events against those awaited by the sleeping processes. If two events have the same *resource* and *op* values, then Pulse applies the operation represented by *op* on the *val* fields of the two events. For example, if *op* is *and*, then Pulse does a logical *and* on the two *val* fields. A result of true indicates that the speculative process produces an event for which the sleeping process is waiting. Thus Pulse constructs a producer edge from the node that represents the event to the node that represents the parent process of the speculative process.

4.3.4 Summary

The major components of our implementation are the in-kernel fork and modification of the blocking system calls and their counterpart calls. The in-kernel fork implementation is the most involved part in our coding; however, our code is small and highly efficient, con-

sisting of only 94 lines of C code, 47 lines of inline assembly, and 7 lines of assembly. Modifying the system calls is easy since it simply involves identifying the corresponding resources and conditions, and recording them in a per-process structure (~160 bytes). For this thesis, we have modified only three blocking system calls and their counterpart calls. The same methodology, however, can be easily applied to modify other system calls.

4.4 Evaluation

We apply Pulse against deadlocked solutions to the classical dining-philosophers and smokers problems and a well-known deadlock scenario in the Apache web server version 2.0.49. This section describes how Pulse works for these different deadlock cases and evaluates its overhead. The evaluation is on an IBM xSeries 445 eServer with eight Intel Xeon 2.8 GHz processors and 32 GB memory. We configure Pulse to transition from nap mode to monitor mode with a default value of every five minutes, unless otherwise mentioned. We set the event buffers of speculative processes to store at most ten events, and every speculative process exists in the system for at most one second. Our results show that Pulse can detect deadlocks caused by incorrect ordering of lock acquisitions, as well as deadlocks involving synchronization semaphores and pipes, which, to the best of our knowledge, no existing tools can detect. Furthermore, Pulse generates no false positives or negatives throughout our experiments.

4.4.1 The dining-philosophers problem

Figure 4-6 shows one incorrect solution to the dining-philosophers problem. We implement the locks as NPTL mutexes. The lock routine is implemented using the `pthread_mutex_lock` function and unlock using `pthread_mutex_unlock`. In Figure 4-6, suppose that all five philosophers take their left forks simultaneously. Then they will all block on their right forks, and there will be a deadlock. We choose this problem as an example of deadlock caused by incorrect use of mutual exclusion locks. All existing dynamic and static tools target this type of deadlock.

```
while (1) {
    think();
    lock(fork[i]);                  // take left fork
    lock(fork[(i + 1) % 5];         // take right fork
    eat();
    unlock(fork[i]);                // put left fork
    unlock(fork[(i + 1) % 5]);      // put right fork
}
```

Figure 4-6. The code of philosopher i.

When Pulse enters the detection mode, it identifies that each philosopher process is waiting for an address (corresponding to its right fork) to contain a value zero (corresponding to the release of the right fork). It then forks a speculative process for each philosopher. The speculative processes first unlock their right forks within their own address spaces, and then execute ahead. During execution, they discover that they produce the unlock events that could unblock their left neighbors. Finally, Pulse constructs the resource graph shown in Figure 4-7. This graph contains a cycle, indicating the existence of a potential deadlock.

4.4.2 The smokers problem

The smokers problem is a classic example of using semaphores for synchronization, instead of mutual exclusion. The static deadlock detection tool RacerX [14] specifically ignores checking deadlocks involving such semaphores. With speculative execution, Pulse is able to detect such deadlocks.

Figure 4-8 shows a solution to the smokers problem that can deadlock. The processes share four binary semaphores: tobacco, paper, matches, and order. The semaphores are implemented using the NPTL library. A P operation is implemented using the `sem_wait` function, and V is implemented using `sem_post`. In Figure 4-8, suppose that the agent releases tobacco and matches, smoker 1 grabs the tobacco, and smoker 3 grabs the matches. Then smoker 1 will block in P(paper), waiting for the address corresponding to paper to have a value greater than zero. Similarly, smoker 2 will block in P(paper), smoker 3 in P(tobacco), and the agent in P(order). Thus all processes will deadlock.

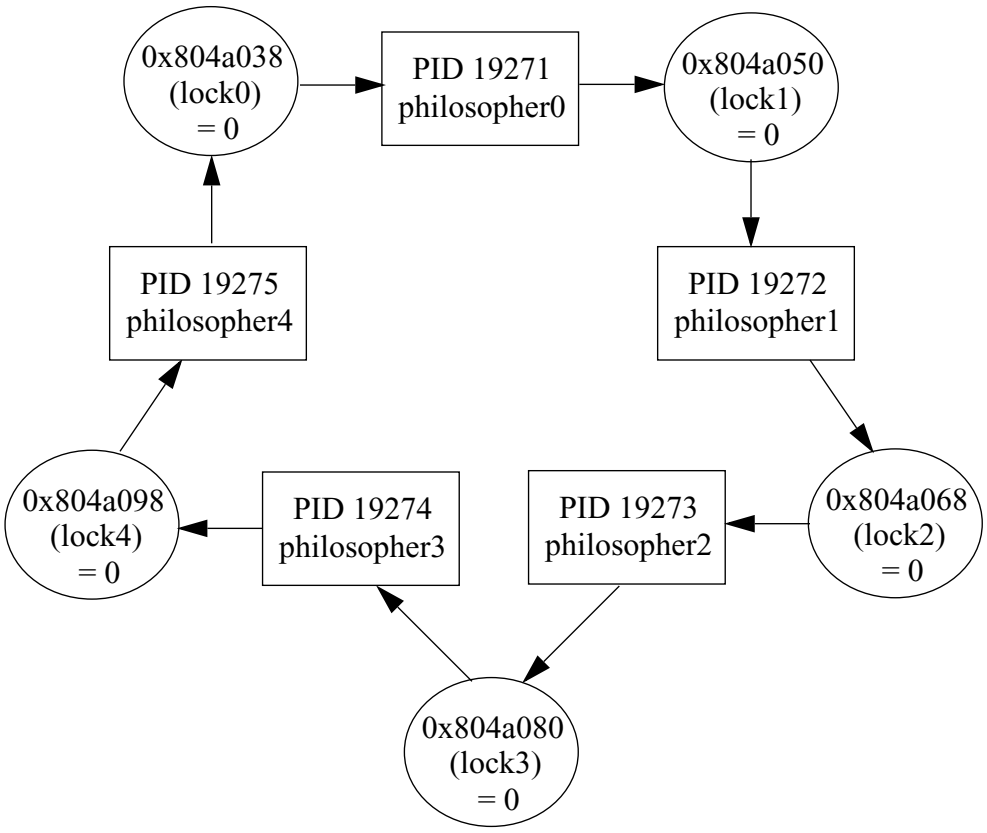


Figure 4-7. Resource graph for the dining-philosophers problem. The hex numbers are virtual addresses corresponding to the locks.

```
smoker 1                smoker 2                smoker 3
while (1) {             while (1) {             while (1) {
  P(tobacco)              P(paper) // block       P(matches)
  P(paper) // block       P(matches)              P(tobacco) // block
  V(order)                V(order)                V(order)
}                       }                       }

        agent
        while (1) {
          P(order) // block
          V(one of tobacco, paper, matches at random)
          V(one of the three at random but not above)
        }
```

Figure 4-8. A deadlocked solution to the smokers problem.

By speculatively unblocking smoker 1, Pulse observes an event corresponding to V(order), which matches the event awaited by the agent. Thus Pulse constructs a producer

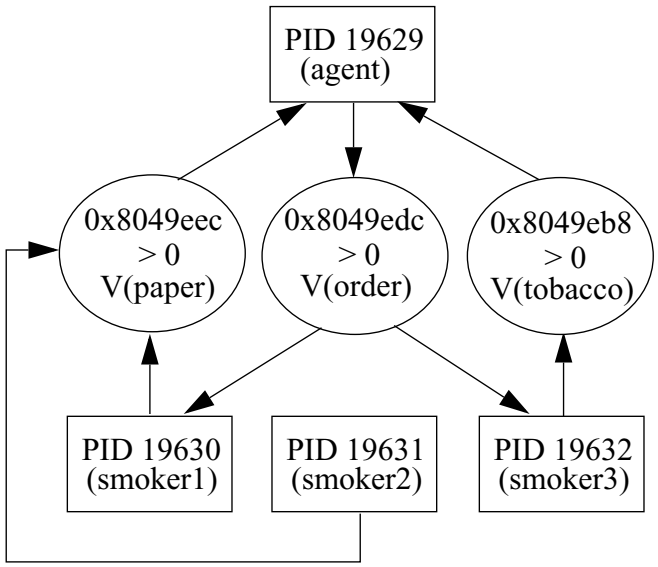


Figure 4-9. Resource graph for the smoker's problem. The hex numbers are virtual addresses corresponding to the semaphores.

edge from an event node representing V(order) to smoker 1's node. Then the speculative process executes P(tobacco) again. Tobacco is not available any more, so this speculative process blocks again. For smoker 2, after Pulse unblocks the speculative process from P(paper), the speculative process blocks immediately in P(matches), and thus it does not produce any event that could unblock another process. So the process node of smoker 2 has no incoming producer edges. Smoker 3 executes similarly to smoker 1.

Suppose that after Pulse speculatively unblocks the agent, the speculative process executes V(tobacco) and V(paper). Then Pulse will discover producer edges from both the event nodes, corresponding to V(tobacco) and V(paper), to the agent. Figure 4-9 shows the final graph we obtain. This graph contains two cycles, agent → V(order) → smoker 1 → V(paper) → agent, and agent → V(order) → smoker 3 → V(tobacco) → agent, indicating the existence of a potential deadlock.

4.4.3 Apache web server deadlock

We have reproduced a well-known deadlock situation in Apache 2.0.49 with the prefork Multi-Processing Module (MPM). A full description of this bug (number 22030) can be found in the Apache bug database (http://nagoya.apache.org/bugzilla/). This bug is widely

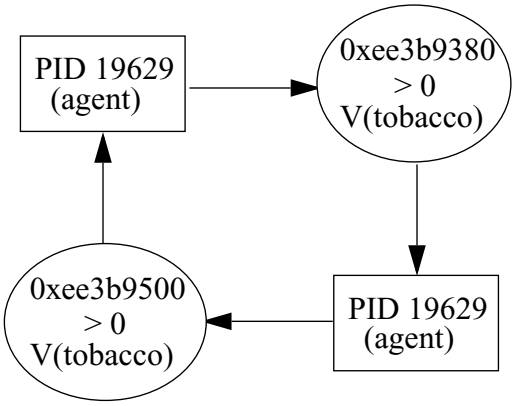


Figure 4-10. Resource graph for the Apache deadlock. The hex numbers are addresses of the corresponding pipe inode structures. We label the events in words although Pulse encodes them abstractly.

referred to as the Apache oversized stderr buffer denial-of-service vulnerability by security companies like Symantec [55]. In the extreme case, this bug can cause an entire web site to stop functioning.

To reproduce the deadlock effect of this bug, we create a Perl CGI script, which first writes 4097 bytes to stderr and then some arbitrary data to stdout. When a client requests this CGI script from a remote browser, a deadlock happens on the server side. This deadlock involves two processes: the CGI script's process and the httpd process that handles the CGI request. The reason for this deadlock is because Apache redirects stderr and stdout of the CGI script to two pipes. The pipe buffer size in Linux is 4096 bytes; a blocking write to a pipe blocks if the write data exceeds 4096 bytes. Thus when the CGI script writes 4097 bytes to stderr, it blocks in the `write` system call, waiting for the httpd process to read data out of the pipe. Meanwhile, the httpd process is blocked in the `poll` system call, waiting for the CGI script to write data to the stdout pipe.

To the best of our knowledge, no existing techniques can detect this deadlock; with Pulse, we successfully detect it. Figure 4-10 shows the resource graph constructed by Pulse. This graph contains a cycle, indicating the existence of deadlock. It is worth noting that Apache has a timeout mechanism that allows it to fail the CGI request after about five minutes, thus breaking the deadlock. However, Apache provides no debugging information

for the deadlock. When multiple sites trigger this deadlock simultaneously, it can effectively become a denial-of-service attack.

4.4.4 Performance Overhead

We evaluate three aspects of Pulse's performance overhead: overhead of the modified system calls, overhead of periodic checking, and overhead of deadlock detection using speculative execution.

System call overhead. We write a microbenchmark for each of the three blocking system calls we modify. Our goal is to measure the overhead that our modified system calls introduce to correct, non-deadlocked applications. Since the additional code in our modified system calls that counterpart the blocking calls is executed only by speculative processes, we do not measure the overhead of these counterpart calls.

Our microbenchmark for the `futex` system call repeatedly invokes `futex` for a total of one million times. We specifically choose the arguments passed to `futex` such that the microbenchmark executes the new code we add, but it does not block (thus allowing the microbenchmark to repeatedly invoke the system call). We run the microbenchmark using the modified and unmodified `futex` call, and compare the average time taken per call. We run similar microbenchmarks for `write` and `poll`. Compared to the unmodified versions, the slowdowns of our modified system calls are: 0.2% for `futex`, 0.9% for `write`, and 1% for `poll`. These slowdowns are small, and most importantly, they occur mostly when a process is about to block.

Periodic checking overhead. We evaluate the performance overhead that Pulse introduces to correct, non-deadlocked applications. This overhead comes from Pulse's periodic activities of transitioning from nap to monitor mode and checking if it needs to enter the detection mode. For applications that do not deadlock and consist of no long-sleeping processes, Pulse does not enter the detection mode—after the periodic checking, it returns back to the nap mode. Our experiments show that, for a system with 100 threads, Pulse takes 0.29 seconds to transition from nap to monitor mode, scan all the threads to determine if it needs to enter the detection mode, and finally transition back to the nap mode. This time stays

almost constant as we create more threads in the system, and only increases to 0.32 seconds when the system has a total of 2000 threads.

To measure how much performance overhead Pulse introduces to normal applications over time, we run Apache Bench from a remote machine to stress test our server running Apache 2.0.49. We set Apache Bench to perform a total of five million HTTP requests and send 1000 simultaneous requests at any time (thus keeping the server busy). The entire test takes about 30 minutes to complete. During this period, without Pulse running, Apache Bench obtains throughput of 2689 requests per second. We then run Pulse and set it to periodically transition from nap to monitor mode every one minute. Apache Bench now obtains throughput of 2684 requests per second, which is almost the same as the throughput obtained without Pulse running. These results show that Pulse has negligible impact on the performance of applications that do not deadlock.

Deadlock detection overhead. We measure the time Pulse takes to detect a deadlock, which is the duration from the time Pulse enters the detection mode to the time it finishes the detection and prints out the results. We obtain the following results: it takes Pulse 2.1 seconds to detect the deadlock in the dining-philosophers problem, 1.7 seconds in the smokers problem, and 1.5 seconds in the Apache web server. We also run these three benchmarks together such that they all deadlock at the same time. We see that Pulse can construct a general resource graph that has three subgraphs, each corresponding to one of the deadlock scenarios, and the entire detection only takes three seconds.

4.5 Related Work

In this section, we discuss related work on dynamic and static deadlock detection and OS-level speculative execution.

Dynamic deadlock detection. Most dynamic schemes detect simple deadlocks that involve only lock-like resources. By tracking every acquire and release of the shared resources, these schemes can discover cyclic dependences among the processes. There is a large body of research on deadlock detection in distributed systems (see a survey by Sin-

ghal [50]). Some software systems also provide dynamic deadlock detection functionalities. For example, the Windows Driver Verifier and Java HotSpot VM both can track the use of locks and detect cyclic lock dependences. Database systems, such as Berkeley DB, MySQL, Oracle, and PostgreSQL, use timeouts and WFGs to detect deadlock. The Linux kernel can perform simple deadlock detection whenever a process invokes the `fcntl` system call. Havelund [23] describes a dynamic deadlock detection mechanism as an extension to NASA's Java PathFinder 2 model checking system [60]. This mechanism records lock operations executed by each Java thread and performs post-mortem analysis to detect potential deadlocks. Different from all these mechanisms, Pulse does not assume only lock-like resources and can detect more general forms of deadlock.

Static deadlock detection. Model checking is a formal verification technique that searches in a program's state space for possible errors, including deadlock. Example model checking systems include Bandera [13], VeriSoft [21], SPIN [27], and Java PathFinder [60]. However, model checking large, complex software systems is still impractical due to the state explosion problem.

Microsoft suggests modelling multithreaded Win32 applications as Petri Nets and using their DLDETECT tool to statically analyze programs for potential deadlock [4]. Sun Solaris provides the Crash Analysis Tool (CAT) that helps users statically analyze system crash dumps to identify simple lock-induced deadlocks. Similarly, Linux supports the Non-Maskable Interrupt (NMI) watchdog, which periodically prints out system information that can help statically identify deadlock.

Recently, Engler et al. [14] proposed RacerX, a static tool for checking data races and deadlocks in large software systems. RacerX annotates source code of the system being checked, constructs the whole-system control flow graph, and searches within the graph for possible deadlocks. An important part of RacerX is to rank the detected deadlocks in terms of how likely they are to occur and filter out inaccurate deadlock warnings. RacerX can only detect deadlocks involving lock-like resources. In contrast, Pulse can detect more complex deadlocks involving consumable resources.

Speculative execution. Chang and Gibson [9] proposed a design for automatically transforming applications to perform speculative execution and issue hints for their future I/O read accesses. Fraser and Chang [19] improved this design by leveraging existing OS features to perform speculative execution. To ensure safety, they sever the ordinary relationships between speculative processes and their parents, and disallow speculative processes to execute potentially unsafe system calls. Pulse employs similar techniques to ensure safety.

4.6 Conclusion

Deadlock is a potential problem for any concurrent system and is often difficult to debug. When a set of processes deadlock, they occupy system resources without doing useful work. Thus, timely detection of deadlock and resolving it is essential for efficient resource management. However, existing deadlock detection techniques are either impractical for large software systems or over-simplified in their assumptions about deadlock-sensitive resources. In this chapter, we propose Pulse, a novel operating system mechanism that dynamically detects deadlock in user applications.

Pulse runs as a system daemon. Periodically, it identifies long-sleeping processes and the events they are waiting for. For each of these processes, Pulse forks a speculative process, which executes ahead in its parent's program. Speculative execution enables Pulse to discover dependences among the sleeping processes. Based on this information, it constructs a general resource graph. If the graph contains cycles, Pulse outputs that a potential deadlock exists. It also prints out the entire graph to help application developers identify causes of the deadlock.

Our evaluation demonstrates that Pulse can detect various types of deadlock, including deadlocks involving consumable resources, which, to the best of our knowledge, no existing tool can detect. We show that Pulse detects deadlock quickly and that it introduces little performance overhead to normal applications that do not deadlock. Application developers can view Pulse as one more tool to add to the deadlock detection toolbox. When they use Pulse and the existing tools together, developers can obtain the best coverage of deadlocks.

5 Conclusion

Multithreaded systems provide hardware execution contexts that allow multiple threads to execute in parallel. These systems often provide infrastructure for important server applications, such as database and web servers. To ensure that applications always receive their desired quality-of-service (e.g., high throughput, low response time), it is important to efficiently manage hardware resources in multithreaded systems.

Existing resource management solutions are far from optimal. The OS and hardware are often unaware of the metrics in which applications are interested for measuring their received quality-of-service (e.g., transactions-per-second for a database application). Consequently, resource management policies are often unaware of the impact that their decisions have on applications. Such lack of knowledge about application behavior can lead to inefficient resource usage in multithreaded systems.

This thesis develops mechanisms that enable multithreaded systems to self-monitor application runtime behavior. To support efficient management of processor resources (e.g., functional units), we study monitoring of thread microexecutions and their correlations with application-level performance metrics. To support efficient management of system resources (e.g., processors), we study monitoring of thread interactions. We focus on monitoring thread synchronization behaviors, which reflect application performance, and thread deadlocks, which reveal errors in applications.

The first contribution of this thesis is a DAG model that quantifies criticality of instructions in shared memory multithreaded systems. By evaluating instruction criticality for commercial and scientific workloads, we found that instructions in the same program often possess a diverse range of criticality values. Thus, processor resource management policies based on instruction criticality can achieve more efficient utilization of resources, such as functional units, than existing simply priority-based policies.

The second contribution of this thesis is a hardware mechanism that automatically detects when a thread is spinning. Based on the proposed hardware, we developed (1) performance counters that accurately reflect application performance, (2) scheduling and power management policies that are more efficient than existing policies, and (3) hardware and software support for detecting thread livelocks/deadlocks.

The third contribution of this thesis is Pulse, an OS mechanism that dynamically detects deadlocks in user applications. Different from existing detection techniques, Pulse uses speculative execution to discover dependences among processes and events. The ability to look into the future allows Pulse to detect complex deadlocks, such as those involving consumable resources, which, to the best of our knowledge, no existing tools can detect.

As we pointed out in Chapter 1, the major obstacle faced by designers when devising resource management policies is the lack of knowledge in the OS and hardware about how applications perform in terms of their specific metrics. To obtain such knowledge, we have taken the approach in which we reason about it by monitoring aspects of application runtime behavior. A direction of future research is to adopt a whole-system approach to redefine the interfaces between the applications, OS, and hardware, such that application-level information can directly communicate down to the lower levels of the system. Designing such interfaces likely requires extending existing programming languages, compilers, operating systems, and/or processor architectures. However, no matter what design choice we make, our goal remains the same: to design systems that provide best quality-of-service to applications with the most efficient usage of hardware resources.

Bibliography

[1] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[3] Ernest Artiaga, Nacho Navarro, Xavier Martorell, and Yolanda Becerra. Implementing PARMACS Macros for Shared Memory Multiprocessor Environments. Technical report, Polytechnic University of Catalunya, Department of Computer Architecture Technical Report UPC-DAC-1997-07, January 1997.

[4] Ruediger R. Asche. Putting DLDETECT to Work. MSDN Library Technical Articles, Microsoft Corporation, January 1994.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 164–177, October 2003.

[6] Carl J. Beckmann. *Hardware and Software for Functional and Fine Grain Parallelism*. PhD thesis, University of Illinois at Urbana-Champaign, April 1994.

[7] Ray Bryant and John Hawkes. Lockmeter: Highly Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, pages 271–282, October 2000.

[8] Jason Casmira and Dirk Grunwald. Dynamic Instruction Scheduling Slack. In *Proceedings of the Kool Chips Workshop, in conjunction with MICRO 33*, December 2000.

[9] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, February 1999.

[10] Alan Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[11] Shing Chi Cheung and Jeff Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.

[12] Compaq. *Alpha 21264 Microprocessor Hardware Reference Manual*. Compaq Computer Corporation, July 1999.

[13] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Puasuareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 1999.

[14] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlock. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 237–252, October 2003.

[15] Brian Fields, Rastislav Bodik, and Mark D. Hill. Slack: Maximizing Performance Under Technological Constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[16] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[17] Brian R. Fisk and R. Iris Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. In *Proceedings of the International Conference on Computer Design*, pages 538–545, October 1999.

[18] Daniele Folegnani and Antonio González. Energy-Effective Issue Logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, July 2001.

[19] Keir Fraser and Fay Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 325–338, June 2003.

[20] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Von Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[21] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of The 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186, January 1997.

[22] H. O. Hartley and A. W. Wortham. A Statistical Theory for PERT Critical Path Analysis. *Management Science*, 12(10):B–469–B–481, June 1966.

[23] Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th SPIN Workshop*, pages 245–264, August 2000.

[24] Jeffrey K. Hollingsworth. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, October 1998.

[25] Jeffrey K. Hollingsworth and Barton P. Miller. Slack: A New Performance Metric for Parallel Programs. Technical Report 1260, Computer Sciences Department, University of Wisconsin–Madison, December 1994.

[26] Richard C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.

[27] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[28] Intel. Using Spin-Loops on Intel Pentium 4 Procesor and Intel Xeon Processor. Order Number: 248674-002, Intel Corporation, May 2001.

[29] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar/Apr 2004.

[30] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 41–55, October 1991.

[31] Hironori Kasahara and Seinosuke Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[32] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 306–317, June 1998.

[33] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[34] Kevin M. Lepak, Harold W. Cain, and Mikko H. Lipasti. Redeeming IPC as a Performance Metric for Multithreaded Programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[35] Kevin M. Lepak and Mikko H. Lipasti. Silent Stores for Free. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 22–31, December 2000.

[36] Ben-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[37] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[38] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[39] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[40] Luke K. McDowell, Susan J. Eggers, and Steven D. Gribble. Improving Server Software Support for Simultaneous Multithreaded Processors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 37–48, June 2003.

[41] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffery K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.

[42] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[43] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[44] Robert H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 1–11, 1993.

[45] Kunle Olukotun, Basem A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[46] Susan Owicki and Leslie Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[47] Ryan Rakvic, Bryan Black, Deepak Limaye, and John P. Shen. Non-vital Loads. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 165–174, January 2002.

[48] Joshua M. Redstone. *An Analysis of Software Interface Issues for SMT Processors*. PhD thesis, University of Washington, December 2002.

[49] John S. Seng, Eric S. Tune, and Dean M. Tullsen. Reducing Power with Dynamic Critical Path Information. In *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 114–123, December 2001.

[50] Mukesh Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22(11):37–48, November 1989.

[51] Yan Solihin, Vinh Lam, and Josep Torrellas. Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. In *Proceedings of Supercomputing '99*, November 1999.

[52] SPEC. SPEC OpenMP Benchmark Suite V3.0. http://www.spec.org/omp, December 2003.

[53] Srikanth T. Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, July 2001.

[54] Srikanth T. Srinivasan and Alvin R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 148–159, November 1998.

[55] Symantec. *Symantec NetRecon™ 3.6 Security Update 6 Release Notes*. Symantec Corporation, August 2003.

[56] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[57] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 54–58, January 1999.

[58] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 185–196, January 2001.

[59] Eric S. Tune, Dean M. Tullsen, and Brad Calder. Quantifying Instruction Criticality. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.

[60] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Journal of Automated Software Engineering*, 10(2):203–232, April 2003.

[61] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.

[62] Robert W. Wisniewski and Bryan Rosenburg. Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems. In *Proceedings of SC2003*, November 2003. To appear.

[63] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[64] Tadaaki Yamauchi, Lance Hammond, Kunle Olukotun, and Kazutami Arimoto. A Single Chip Multiprocessor Integrated with High Density DRAM. *IEICE Transactions on Electronics*, E82-C(8):1567–1577, August 1999.

[65] Cui-Qing Yang and Barton P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *Proceedings of the Seventh Conference on Distributed Memory Computer Systems*, pages 366–373, June 1988.

Biography

Tong Li was born on February 10, 1976 in Xi'an, China. He received his B.E. from Northwestern Polytechnic University in China in 1995 and M.S. from the University of Kentucky in 1999, both in Computer Science. He was a recipient of the James B. Duke Graduate Fellowship in 1999. His research interests include computer microarchitecture, multiprocessor architecture, and operating systems. His Ph.D. thesis focuses on designing hardware and operating system support for efficient resource management in multithreaded systems.