USING COMPLETE MACHINE SIMULATION TO UNDERSTAND COMPUTER SYSTEM BEHAVIOR

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Stephen Alan Herrod

February 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Mendel Rosenblum, Principal Advisor

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Anoop Gupta

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Kunle Olukotun

Approved for the University Committee on Graduate Studies:

Abstract

This dissertation describes complete machine simulation, a novel approach to understanding the behavior of modern computer systems. Complete machine simulation differs from existing simulation tools by modeling *all* of the hardware found in a typical computer system. This allows it to help investigators better understand the behavior of machines running commercial operating systems as well as any application designed for these operating systems. These include database management systems, web servers, and other operating system-intensive applications that are important to computer system research. In contrast, most existing simulation tools model only limited portions of a computer's hardware and cannot support the accurate investigation of these workloads. In addition to extensive workload support, the complete machine simulation approach permits significant hardware modeling flexibility and provides detailed information regarding the behavior of this hardware. This combination of features has widespread applicability, providing benefits to such research domains as hardware design, operating system development, and application performance tuning.

Although machine simulation is a well-established technique, it has traditionally been limited to less ambitious use. Complete machine simulation extends the applicability of traditional machine simulation techniques by addressing two difficult challenges. The first challenge is to achieve the speed needed to investigate complex, long-running workloads. To address this challenge, complete machine simulation allows an investigator to dynamically adjust the characteristics of its hardware simulation. There is an inherent trade-off between the level of detail that a hardware simulator models and the speed at which it runs, and complete machine simulation provides users with explicit control over this trade-off. An investigator can select a high-speed, low-detail simulation setting to quickly pass through uninteresting portions of a workload's execution. Once the workload has reached a more interesting execution state, an investigator can switch to slower, more detailed simulation to obtain behavioral information.

The second challenge is to efficiently organize low-level hardware simulation data into information that is more meaningful to an investigation. Complete machine simulation addresses this challenge by providing mechanisms that allow a user to easily incorporate higher-level workload knowledge into the data management process. These mechanisms are efficient and further improve simulation speed by customizing all data collection and reporting to the specific needs of an investigation.

To realize the benefits of complete machine simulation and to demonstrate effective solutions to its challenges, this dissertation describes the SimOS complete machine simulator. The initial version of SimOS models uniprocessor and multiprocessor computer systems in enough detail to run Silicon Graphics's IRIX operating system as well as the large class of applications designed for this platform. Furthermore, recent versions of SimOS support additional architectures and operating systems. Our early experiences with SimOS have been extremely positive. In use for several years, SimOS has enabled several studies not possible with existing tools and has demonstrated the effectiveness of the complete machine simulation approach.

Acknowledgments

I would like to thank several people who have made my time at Stanford both an enjoyable and rewarding experience. First, I would like to thank my advisor, Mendel Rosenblum, for his guidance throughout my graduate career. He has been a vocal supporter of SimOS since its humble beginnings and has made significant contributions throughout its lifetime. Furthermore, he has motivated me throughout the Ph.D. process and has taught me how to perform modern computer systems research. I would also like to thank Mendel, Anoop Gupta and Kunle Olukotun for their participation on my reading committee. Their comments have greatly improved the quality of this dissertation.

The students in the Hive and Disco operating system groups have been the heaviest users of SimOS, and have all contributed to its success. A few students have made especially significant contributions and deserve special mention. Emmett Witchel is largely responsible for the Embra CPU simulator. He was an early contributor to the SimOS effort, and his legacy continues long after his departure for colder lands. Ed Bugnion and Scott Devine also joined the SimOS effort early in its development. They have contributed heavily to its design both through substantial coding and through vociferous participation in extensive arguments over its features and direction.

I would also like to thank the National Science Foundation and the Intel Foundation for the graduate research fellowships that supported me at Stanford, ARPA for funding this project in contract DABT63-94-C-0054, and Silicon Graphics for the machines and IRIX source code with which SimOS was developed.

On a more personal note, the entire FLASH group has helped make my Stanford years among the most enjoyable of my life. Specific thanks to Robert Bosch, Ed Bugnion, John Chapin, Scott Devine, Kinshuk Govil, Beth Seamans, Dan Teodosiu, Ben Verghese, Ben Werther, and the entire softball team for many years of Tressider lunches, quasi-competitive sports outings, and ice-cold frappuccinos. Special appreciation goes to my officemates, John Heinlein and Robert Bosch, for putting up with many hours of stories, jokes, complaints, and gossip.

Thanks also goes to the folks at Rains #14H for improving life away from the computer. Paul Lemoine, Randall Eike, Ulrich Thonemann, Steve Fleischer, Michael Youssefmir, and Karl Pfleger have all helped delay my graduation date through coffee, frisbee, DOOM, rice goop, tennis, golf, C&C, and many nights out on the town. Roommates come and go, but these friends have remained long after the boxes were packed.

I also thank my parents, Ted and Andrea, and the rest of my family for unwavering encouragement while I spent time far away from home. Without all of their support, "vacationing" from Texas would have been much more difficult.

Finally, I thank my wife Flavia for being right by my side throughout the long graduate school process. She is both an inspiration and an ever-faithful companion with whom I will remain long after "See-mose" is gone.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

This dissertation describes complete machine simulation, a novel approach to helping researchers understand the behavior of modern computer systems. Complete machine simulation differs from existing simulation approaches in that it models all of the hardware typically found in a computer system. As a result, it can boot, run, and investigate the behavior of machines running a fully functional operating system as well as any application designed to run on this operating system. This includes database management systems, web servers, and other operating system-intensive applications that are important to computer system researchers. In contrast, most existing simulation tools model only limited portions of a computer's hardware and cannot enable the same breadth and quality of investigation available through the use of complete machine simulation.

This dissertation argues that complete machine simulation approach is an effective tool for computer system investigations. In support of this argument, the work described in this dissertation makes three primary contributions:

- **Demonstration of the significant benefits that complete machine simulation provides to many types of computer systems research.**

  Complete machine simulation offers several benefits to computer systems research including extensive workload support, accurate and flexible machine modeling, and comprehensive data collection capabilities. Our experiences with complete machine simulation have shown these benefits to be quite valuable, allowing us to perform studies not possible with existing tools and techniques.

- **Demonstration that adjustable levels of simulation speed and detail help complete machine simulation provide results as quickly as possible.**

  The biggest challenge facing complete machine simulation's acceptance is its performance, and this work demonstrates how adjustable simulation speed and detail characteristics address this challenge. Specifically, this work recognizes the importance of three specific simulation execution modes and the ability to dynamically switch between them during the course of a workload's execution.

- **Specification and implementation of mechanisms for addressing complete machine simulation's data management challenges.**

  Another challenge for complete machine simulation is efficient conversion of hardware-level data into higher level behavioral information, and this work demonstrates how supporting investigation-specific data management addresses this challenge. Specifically, this work introduces efficient and flexible mechanisms that allow an investigator to customize all simulation data classification and reporting to meet the specific needs of their study.

1.1 The challenge: Understanding computer system behavior

The complexity of modern computer systems presents a challenge to researchers interested in understanding their behavior. In the continual pursuit of higher performance, hardware implementations are growing increasingly complex and the effects of individual architectural features are difficult to ascertain. The close interaction of complex applications and operating systems with this hardware further complicates efforts to understand system behavior.

Effective computer system behavioral information is required in several different investigative domains and takes different forms accordingly. For example, computer architects need to understand the effects of potential architectural features on the behavior of important workloads. Detailed hardware performance information regarding these features is critical for selecting an appropriate architectural design. Similarly, software engineers are continually striving to improve the performance of applications. Understanding the run-time behavior of an application, including its interaction with its supporting hardware and operating system, helps focus the software engineering effort towards modifications most likely to yield significant improvement. Finally, operating system designers must continuously provide additional services and improved performance in their product. Information concerning the operating system's interaction with the hardware platform and with its supported applications is essential to both of these endeavors. The challenge facing all computer system researchers is thus to obtain the information that best helps them understand increasingly complex computer system behavior, and the development and use of complete machine simulation is directly motivated by this challenge.

1.2 Techniques for investigating computer system behavior

Driven by the need for information regarding a computer system's behavior, researchers and designers have utilized several different approaches. These techniques and tools, each of which has strengths and weaknesses, fit into the broad categories of analytic modeling, hardware prototyping, and software simulation.

Analytic modeling

Analytic models are mathematical approximations of the behavior of complex systems. These models are often used for analyzing higher-level system issues. For example, an analytic model might be used to understand a disk drive's queuing behavior as it satisfies read requests arriving at probabilistic intervals. When an analytic model can be devised for a particular investigation, their results can be valuable. However, they have limited applicability to many computer system investigations. Analytic models typically break down when looking at detailed hardware and software interactions, requiring too many simplifying approximations to provide the accurate behavioral information required by computer architects and software engineers.

Hardware prototyping

Hardware prototyping is a more commonly used technique for understanding computer system behavior, especially in architectural design investigations. Hardware prototyping consists of designing and building a hardware component and including mechanisms for self-observation. The hardware prototype is integrated into an existing computer system and exercised by driving the system with applications. The applications act as input data for the prototype's observing mechanisms, and the resulting data describes the prototype's behavior.

Hardware prototypes are good at collecting very low-level, focused behavioral information and can collect this data at very high speeds, but there are also several limitations to their use. First, building hardware prototypes is both time-consuming and expensive, and many investigations can not afford the required time or financial commitment. More importantly, hardware prototypes have restricted flexibility: it is difficult to significantly reconfigure hardware, and thus a prototype is restricted to architectural investigations that fall within its own limited domain.

Software simulation

Because it addresses many of the deficiencies of analytic modeling and hardware prototyping, software simulation is the most popular method of testing, evaluating, and designing modern computer systems. Software simulation involves modeling some of the functionality and behavior of a computer system completely in software and then driving this model with appropriate input data. For brevity's sake, software simulation is hereafter referred to simply as simulation.

Unlike analytic models, simulation can model complex hardware and software interactions very accurately. Unlike hardware prototyping, simulation is extremely flexible and capable of modeling a wide domain architectural designs and configurations. Even in cases where it is possible to build a hardware prototype of a given design, a comprehensive simulation of the same hardware is both less expensive and less time-consuming to develop. In addition to flexibly modeling the behavior of hardware, simulation models also include code to collect information regarding the hardware's

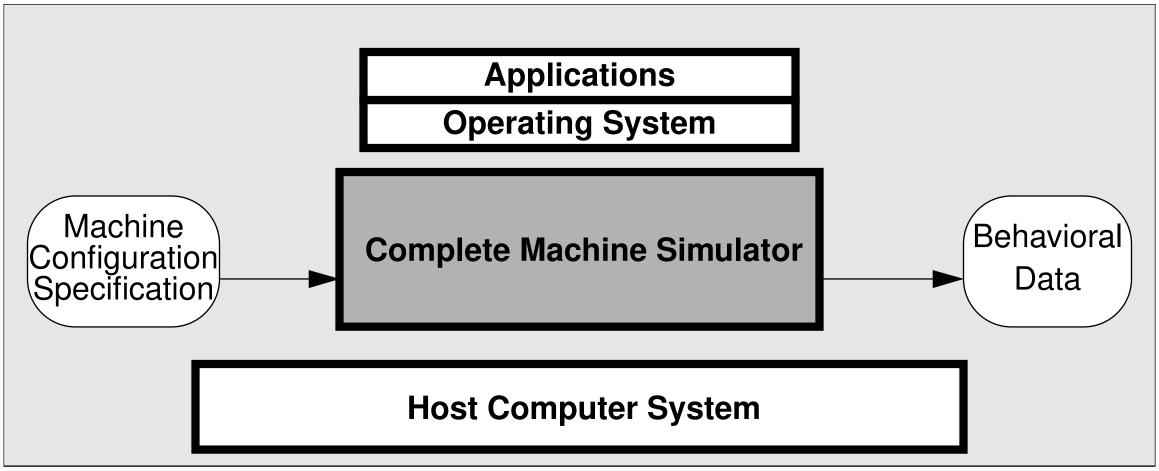

Figure 1.1. Complete machine simulation

activity. A simulation model has tremendous visibility into its own behavior and can potentially provide accurate behavioral information.

Simulation's flexibility and visibility make it an attractive tool, and it is used in almost every stage of investigating a computer system; from the evaluation of research ideas, to the verification of the hardware design, to performance tuning once the system has been built. However, simulation does have its own weaknesses, the primary of which is speed. The exact flexibility and visibility that make simulation attractive can also make it several orders of magnitude slower than a hardware implementation. If simulation cannot provide behavioral information in a timely manner, its usefulness to researchers and designers is significantly decreased. To address this problem, existing simulation tools typically model only limited portions of a computer's hardware. This limited modeling approach simplifies the simulator implementation effort and improves the speed at which simulators provide their data, but also affects the quality and applicability of the simulator's results.

1.3 Complete machine simulation

As illustrated in Figure 1.1, a complete machine simulator is simply a program that runs on top of existing host computer systems and models all of the hardware found in a modern computer system. The use of this program is relatively straightforward. First, the user specifies the characteristics of the exact "target" machine that the complete machine simulator should model. These characteristics include processor architecture model, processor clock speed, cache configuration, disk seek time, memory system bus bandwidth, and numerous other parameters. The resulting software-based machine can boot and run an operating system and provides the illusion of being just another computer system. For example, an investigator can log into the simulated computer system and then run any applications that have been installed on the simulated computer's file system. While a complete machine simulator provides the outward appearance of being a normal computer system, internally it collects detailed hardware behavioral data only available through the use of simulation. This data is subsequently used to better understand some aspect of the computer system's behavior.

1.3.1 Benefits

Complete machine simulation differs from existing simulation tools in that it models *all* of the hardware found in a typical computer system. This feature provides several benefits for computer system investigators:

- **Extensive workload support**

  By modeling all of the hardware found in an existing computer system, a complete machine simulator can support the full execution of an operating system designed for that computer system. Furthermore, by supporting the execution of an existing operating system, the complete machine simulator can support the investigation of virtually any application designed for that operating system. This includes traditional performance benchmark applications such as those in SPEC, but also more complex, operating system-intensive applications such as database management systems and web servers. The latter applications are important to many types of computer system research, yet are poorly supported by existing tools.

- **Accurate and flexible machine modeling**

  By modeling all of the hardware found in a computer system, complete machine simulation also provides accurate machine modeling capabilities. One of the strengths of any software simulation tool is the ability to model the behavior of present and future hardware designs with significant detail. Complete machine simulation fully exploits this flexibility and allows an investigator to customize every aspect of the simulated machine's behavior to match that of a specific target computer system. The ability to accurately model both present and future hardware designs enables complete machine simulation's use in a variety of architectural investigations.

- **Comprehensive data collection**

  Software simulation provides tremendous visibility into the behavior of the hardware it models. By modeling all of a computer's hardware, complete machine simulation extends this visibility to every aspect of a computer system's execution. As a result, complete machine simulation can provide detailed information regarding every aspect of the modeled computer system's execution behavior. This includes hardware activity as well as the behavior of the software that this hardware supports. Each hardware model can be augmented with code to collect detailed statistics regarding their behavior during workload execution. Furthermore, this information can be obtained non-intrusively. The act of observing the simulated computer's behavior in no way affects its execution, allowing both detailed and accurate statistics collection.

Because it encompasses more functionality than existing simulation tools, the complete machine simulation approach is applicable to a broad class of computer system investigations. Computer architects can evaluate the impact of new hardware designs on the performance of complex workloads by modifying the configuration of the simulated hardware components. Operating system programmers can develop their software in an environment that provides the same interface as the targeted computer platform, while taking advantage of the visibility offered by a simulation environment. Programmers can collect detailed information that describes the dynamic execution behavior of complex applications and use this information to improve their performance. Furthermore, the ability to use a single tool across all of these domains is a benefit in its own right, reducing the implementation, training, and deployment costs associated with the use of multiple investigative tools.

1.3.2 Important features

The idea of simulating all of the hardware in a computer is not new. For example, computer system designers have long used machine simulation techniques to aid in architectural validation. However, three important features help complete machine simulation extend the applicability of machine simulation techniques to more ambitious use:

- **Functional and flexible hardware simulation**

  One of the most important goals of complete machine simulation is support for the investigation of important workloads executing on configurable computer hardware. Complete machine simulation satisfies this goal by supporting both functional and flexible hardware simulation. Specifically, a complete machine simulator implements the exact hardware interfaces found on existing machines, allowing it to execute unmodified operating system and application binaries. While providing this hardware functionality, complete machine simulation gives a user substantial control over the specific implementation details of the simulated hardware. As a result, an investigator can examine a wide variety of workloads as they execute on very specific architectural configurations.

- **Dynamically adjustable simulation speed and detail**

  Complete machine simulation can provide extremely detailed information regarding the behavior of a target computer system, but the benefits of this information are greatly mitigated if it takes an excessively long amount of time to obtain. To address this challenge, a complete machine simulation implementation can allow an investigator to dynamically adjust the characteristics of its hardware simulation. There is an inherent trade-off between the level of detail that a hardware simulator provides and the speed at which it provides this detail, and an effective complete machine simulator provides users with explicit control over this trade-off. An investigator can select a high-speed, low-detail simulation setting to quickly pass through uninteresting portions of a workload's execution. Once the workload has reached a more interesting execution state, an investigator can switch to slower, more detailed simulation to obtain accurate and detailed behavioral results. This feature allows users to select the exact level of detail required at each stage of an investigation, maximizing the speed at which this data is obtained.

- **Efficient management of low-level simulation data**

  A complete machine simulator has excellent potential for providing an investigator with detailed computer system behavioral data because it "sees" all of the hardware activity that occurs during a workload's execution. This includes the execution of instructions, MMU exceptions, cache misses, CPU pipeline stalls, and many other facets of machine behavior. Hardware simulators can be easily augmented with code to measure this activity, but there are two challenges associated with this approach. First, hardware data is often at too low of a level to be meaningful to many investigations. Additionally, this data is generated at a very high rate and efforts to monitor and organize it can significantly slow down a simulation. To address these challenges, a complete machine simulator can provide flexible mechanisms for organizing complete machine simulation's hardware-level data into more meaningful software-level information. Furthermore, these mechanisms can be highly efficient, minimizing the overhead of their data manipulation.

1.4 SimOS

To demonstrate the capabilities and benefits of complete machine simulation, this dissertation introduces SimOS. The SimOS project started in 1992 with the goal of building a tool capable of studying the execution behavior of modern workloads. Recognizing the limitations of our existing tools, we designed SimOS to be a complete machine simulator. The initial version of SimOS modeled entire uniprocessor and multiprocessor computer systems that are capable of booting and running IRIX, an implementation of SVR4 Unix developed by Silicon Graphics. By running the IRIX operating system, a SimOS-modeled computer system is binary-compatible with computer systems shipped by Silicon Graphics, and can support the execution of almost any program designed for that platform. More recent versions of SimOS model DEC Alpha-based machines in enough detail to boot and run DEC Unix, and efforts to support additional architectures and operating systems are well underway.

To provide functional and flexible hardware simulation, SimOS takes a modular approach to machine simulation. SimOS includes well-defined interfaces for the development and inclusion of multiple CPU, memory system, and I/O device models. Each model provides the functionality required to execute operating systems and applications, but provides this functionality while modeling widely varying hardware implementation details.

To provide adjustable simulation speed and detail, SimOS utilizes a combination of compatible hardware simulation models. These models use high-speed machine emulation techniques as well as more traditional hardware simulation techniques to support three important modes of simulator usage.

To provide flexible and efficient data management, SimOS incorporates a Tcl scripting language interpreter customized to support hardware data classification. SimOS users create investigation-specific Tcl scripts that interact closely with the hardware simulation models to control all data recording and classification.

SimOS has been heavily used for several years and has enabled several studies previously inaccessible to computer system researchers. In addition to describing several of these studies, this dissertation will introduce several limitations that we have encountered with the complete machine simulation approach and suggest possible research directions for addressing them.

1.5 Operating system-intensive workloads

As mentioned above, one of the most important benefits of complete machine simulation is its support for the investigation of *operating system-intensive* workloads. This term is used frequently in this dissertation and deserves further description. Operating system-intensive workloads consist of applications that spend a substantial portion of their execution time in operating system code. This execution behavior may be due to explicit use of operating system services or to secondary effects such as multiprogrammed process scheduling or interrupt processing.

Two examples of operating system-intensive applications that are referred to in this dissertation are database management systems and web servers. Database management systems are among the most heavily used applications on modern computer systems, and their execution consists of substantial operating system activity. Database management systems typically consists of multiple processes, and the scheduling of these processes is the job of the operating system. Additionally, database management systems often have significant file system activity, accessing the database tables as well as writing to a commit log file to indicate the completion of each transaction. The operating system is responsible for managing much of this activity, scheduling disk requests and responding to interrupts indicating disk access completion. Finally, database management systems often execute in a client-server environment where remote applications make requests of the database management system via a network connection. As a result, significant execution time can occur in the operating system's networking code, receiving and replying to these requests. The impact of this heavy usage of operating system services can be quite significant. For example, an investigation into the behavior of a typical database transaction processing workload found that it spends close to 40% of its execution time running operating system code [Rosenblum95].

Another important application whose execution consists of significant operating system activity is a web server. Web servers rely quite heavily on an operating system's networking services for accepting and responding to HyperText Transport Protocol (HTTP) requests. Web servers also include significant file system activity accessing requested web pages and saving the requests to an access log. Furthermore, many web servers are consist of multiple processes to allow concurrent handling of HTTP requests. The scheduling of these processes of course has significant impact on the behavior of the server. Together, these characteristics make the performance of a web server extremely dependent on the behavior of the operating system. For example, a simple investigation of the popular Zeus web server [Zeus97] indicates that it spends more than 70% of its total execution time in operating system code when running the Webstone benchmark [Trent95].

In these and other examples, workload behavior is largely determined by the operating system activity that occurs during its execution. As such, tools that can not include operating system effects are often unable to provide the information needed to properly understand the behavior of this important class of workload.

1.6 Additional terminology

Discussion of a simulator that runs on one computer system and models a completely different computer system can be confusing. To help reduce this confusion, this section provides an early introduction to the terminology used throughout this dissertation. When discussing the use of complete machine simulation, the *host computer system* is the hardware and operating system supporting the execution of the simulation tool. In discussing the complete machine simulator itself, the *target machine* is a collection of simulated hardware components configured to model a particular computer implementation. The operating system running on top of this target machine is the *target operating system*. These two components combine to form the *target computer system*.

In discussing specific investigations that utilize complete machine simulation, an *application* is a single instance of a user-level program. The term *workload* refers to one or more applications and includes all of the operating system activity that occurs during their execution. A single execution of the simulator is referred to as an *experiment*, and an *investigation* consists of one or more related experiments.

1.7 Organization of this dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 provides further motivation for the use of complete machine simulation by describing how it benefits several fields of computer systems research.

- Chapter 3 describes the need for functional and flexible hardware simulation and how SimOS's modular simulation approach satisfies this need.

- Chapter 4 describes the challenge of quickly obtaining simulation results and how adjustable levels of simulation speed and detail can address this challenge.

- Chapter 5 describes the data management challenges facing complete machine simulation and the SimOS mechanisms for efficiently organizing hardware-level data into information customized to the needs of an investigation.

- Chapter 6 discusses our experiences with SimOS. In addition to describing how several different investigations benefited from the use of SimOS, this chapter describes limitations that we have encountered with SimOS and the complete machine simulation approach.

- Chapters 7 and 8 conclude the dissertation with a survey of related work and a summary of this research's contributions.

Chapter 2
Motivation

The previous chapter provides an overview of complete machine simulation and briefly introduces three of its benefits: extensive workload support, flexible and accurate architectural modeling, and comprehensive data collection. This chapter provides more concrete motivation for the development and use of complete machine simulation by describing how these benefits apply to modern computer systems research. The first part of this chapter describes the original needs that sparked the development of a new investigative approach. The SimOS implementation of complete machine simulation addressed these original needs and proved advantageous to other types of research as well. The second part of this chapter further motivates the development and use of complete machine simulation by describing its applicability to three diverse fields of computer systems research: hardware design, operating system development, and application performance tuning. SimOS provides unique benefits to each field, enabling several studies not possible with existing tools and techniques. Furthermore, the fact that a single tool can address such a wide variety of research needs is a benefit in its own right. Computer researchers typically utilize several different tools, each designed to address specific investigative needs. By addressing several needs with a single tool, complete machine simulation can reduce a research group's tool implementation, training, and deployment costs

2.1 History

The development of the SimOS complete machine simulator began in 1992 to fill a void in our existing tool set. One of our research group's original goals was a thorough investigation of the cache and memory system behavior of modern workloads. At that time, the best available tool for this task was Tango Lite [Goldschmidt93]. Tango Lite is an execution-driven simulation environment designed to model the behavior of parallel programs. Tango Lite obtains memory system behavioral information by rewriting an application's assembly code, instrumenting load and store instructions with calls to a configurable memory system simulator. The instrumented application is compiled and executed on an existing computer system. At run-time, the instrumented application passes memory reference addresses to the user-defined memory system, allowing the determination of cache hit rates, memory sharing patterns, and other aspects of the application's memory system behavior.

While Tango Lite is an effective tool for investigating scientific applications such as those found in SPLASH [Woo95], several factors limit its usefulness for studying other important workloads. First, Tango Lite requires access to an application's source code to make it utilize a special macro package [Boyle87] and to instrument its memory references. However, many applications, especially those in the commercial sector, are only available in binary form. Even if Tango Lite could instrument application binaries, it would still suffer from its design as a *user-level* simulator. User-level simulators can investigate the behavior of an application itself, but ignore all operating system activity that normally occurs during the application's execution. For example, user-level simulators implement "perfect" system calls where kernel functionality is provided without actually running operating system code. Similarly, operating system activity such as virtual address translations, exceptions, and device interrupts is either omitted or somehow "faked" by the simulator.

Furthermore, user-level simulators typically model only a single application process at a time and can not include the effects of operating system process scheduling. The omission of operating system activity is often acceptable in the investigation of scientific applications as they typically make very few system calls and their execution requires minimal operating system activity. However, applications such as database management systems and web servers have significant operating system activity, and the inability to include this activity compromises the accuracy and applicability of user-level simulation tools.

These limitations directly motivated the development of the SimOS complete machine simulator. The initial goals were to support application investigations without access to their source code, to observe all of the application's operating system activity, and to support the investigation of multiprogrammed and other operating system-intensive workloads. Simulating all of the hardware found in a computer system seemed to be a feasible approach as it could certainly model highly configurable memory systems and could also support the execution of a complete operating system. Supporting the execution of an operating system would in turn allow it to support the execution of any type of workload designed to run on this operating system. Furthermore, software simulation would allow us to observe and measure all of the workload's user-level and operating system-level activity. During the development of this complete machine simulator, we began to recognize the benefits that such an approach could provide to other types of computer systems research as well. To fully explore these benefits, the development of complete machine simulation became a research project in its own right, with active exploration of its capabilities continuing to this day.

2.2 Benefits for computer systems research

While complete machine simulation was initially developed to investigate the memory system behavior of modern workloads, it has proven to be advantageous to other domains of research as well. This section provides further motivation for the development and use of complete machine simulation by describing how its benefits apply to three diverse fields of computer system research: hardware design, operating system development, and application performance tuning. To better convey the significance of these benefits, this section also describes how complete machine simulation improves upon the most common investigative techniques used in each field, enabling studies that existing investigative tools and techniques have difficulty supporting.

2.2.1 Hardware design

One important domain of computer systems research is hardware design. The goal of this type of research is to design and implement the highest performance computer hardware within a set of constraints. These constraints include monetary cost, time deadlines, chip or board space, power consumption, and many other factors. Hardware design thus requires a continual trade-off between an implementation's cost and performance to develop the best possible product given a set of constraints. To evaluate the performance side of this trade-off, designers attempt to predict the behavior of their proposed hardware in support of important applications. For example, researchers typically model the behavior of a specific hardware design in software and then drive this model with data from existing workloads. The behavior of the modeled hardware design is taken to be representative of the behavior of an actual hardware implementation and helps predict its effectiveness. Furthermore, when a hardware design does not provide the expected or desired level of performance, simulation can provide data that helps determine why.

Complete machine simulation is a particularly effective tool for hardware design, providing significant benefits to the performance evaluation effort. First, complete machine simulation's extensive workload support allows a researcher to evaluate a hardware design under a variety of applications. This includes traditional benchmark applications such as those in SPEC [SPEC97] as well as more complex, operating system-intensive applications such as database management systems and web servers. Furthermore, the substantial operating system activity that occurs during the latter class of applications is included in the simulation, providing additional input data for the evaluation. As a result, hardware designs can be evaluated in the context of the exact workloads that the final hardware implementation will be required to support, ultimately leading to a higher performance product.

Second, complete machine simulation's flexible machine modeling capability enables its use in the evaluation of almost any hardware design. Complete machine simulation models all of the hardware found in a computer system and allows a designer to customize the behavior of any of this simulated hardware to match a proposed design. For example, a designer can easily incorporate a new processor pipeline, cache configuration, memory system design, or disk model into the simulated machine and evaluate its performance. Furthermore, an investigator can evaluate the proposed designs in the context of an entire computer system. Computer hardware components are never used in isolation, and the effects of hardware's behavior propagates throughout the system. For example, in a real computer system, a new processor cache implementation would cause different load and store instruction activity, affecting the behavior of the processor pipeline and the rest of the memory system. Additionally, the effects of new hardware would normally propagate up to the application and operating system, ultimately changing the behavior of the workload. While traditional simulation tools evaluate a hardware component in isolation, complete machine simulation evaluates the hardware component as it interacts with the rest of the complete computer system. As a result, complete machine simulation can provide more accurate performance predictions than traditional tools.

Architects have traditionally employed trace-driven simulation to evaluate proposed architectural designs. Trace-driven simulation consists of two phases, trace collection and trace processing. In the trace collection phase, researchers use software or hardware techniques to monitor the behavior of a computer system and collect a "trace" of workload activity. Trace collection involves running the workload of interest on a system modified to record events such as instruction execution or memory references. The trace can be collected either by using software instrumentation as in ATOM [Eustace95], Epoxie [Borg89], FastCache [Lebeck95], and Paradyn [Miller95], or by using a hardware monitor such as in BACH [Grimsrud93], DASH [Torrellas92, Chapin95a], and MONSTER [Nagle92]. The trace is typically saved to non-volatile storage and provides input to some type of hardware simulator. The results of this trace processing phase are taken to approximate the behavior of the modeled hardware in its execution of the traced workload.

The widespread use of trace-driven simulation in the evaluation of hardware designs attests to its speed, flexibility, and ease of implementation. However, there are several limitations to its effectiveness. First, most software-based trace collection tools are unable to capture the activity that occurs during operating system execution. Because operating system activity is omitted from most traces, hardware design evaluation is often based entirely on the user-level portion of applications. Additionally, trace-driven simulation of isolated hardware components omits the important interactions that normally occur between a hardware component and the rest of the computer system. As mentioned above, hardware components are never used in isolation, and the effects of hardware's behavior normally propagates throughout the system. Because trace-based simulation separates the collection of hardware events from the modeling of new hardware component behavior, these interactions do not occur, and the predicted real-life behavior of new hardware designs is compromised.

Computer architects also employ user-level simulation tools such as Tango Lite to evaluate their hardware designs. These simulators generate data regarding an application's user-level execution behavior on the modeled hardware and can thus provide some performance predications. However, the hardware design evaluation effort again suffers due to the omission of operating system activity. Furthermore, user-level simulators are unable to support many operating system-intensive applications or multiprogrammed workloads. As a result, important applications are again absent from the evaluation of a hardware design.

In both cases, the difficulty of obtaining useful data regarding operating-system intensive workloads has led to a heavy reliance on benchmark applications with minimal operating system activity such as the applications that comprise the SPEC benchmarks. Designing hardware to effectively support the execution of these benchmarks provides an important marketing story, but does not necessarily translate into performance gains for more commonly used workloads. Database management systems and web servers are extremely important applications, yet the simulation tools used to evaluate hardware designs are typically unable to capture a significant portion of their execution behavior. As a result, many hardware design decisions are made without complete information regarding significant execution activity that the hardware is required to support.

2.2.2 Operating system development

Another important field of computer systems research is operating system development. In this domain, operating system programmers are required to provide improved functionality and port their code to new platforms while simultaneously minimizing the performance overhead of the operating system's execution. In a commercial environment, this development process is further complicated by very strict time constraints. Operating system development typically involves a repeated cycle of modifying the operating system source code and then running the resulting kernel on existing hardware to evaluate its correctness, functionality, and performance.

Complete machine simulation provides several benefits to the operating system development process. First and foremost, complete machine simulation provides a platform for operating system development long before the targeted hardware is present. The task of porting an operating system to new machines is often delayed by the lack of an actual hardware platform for code testing and tuning. Complete machine simulation's flexible and accurate machine modeling capabilities allows an investigator to model a non-existent machine and provide the operating system with the exact same hardware interfaces that will be found on the completed machine. The ability to enable operating system porting efforts to proceed concurrently with hardware design and implementation can dramatically speed up the overall time to completion of a new computer platform.

Second, complete machine simulation provides better operating system debugging support than hardware. For example, simulation can provide completely repeatable workload execution. This deterministic execution is particularly beneficial to operating system debugging where bugs are often difficult to reproduce. Additionally, complete machine simulation is easily extended to interact with and improve upon existing debugging tools. As described in Chapter 6, complete machine simulation allows a developer to apply normal debugging techniques such as breakpoints and single-stepping to all operating system code, including exception handlers and other timing-critical sections of code whose execution is typically difficult to examine.

Finally, complete machine simulation provides substantially more visibility into operating system behavior than possible with real hardware. Coupled with flexible data collection and reporting capabilities, this feature helps developers to understand and improve the performance of their operating system code. Complete machine simulation can provide detailed information regarding an operating system's performance as it supports a wide variety of important workloads. This performance information can include simple profiling information such as heavily executed procedures, but also more detailed behavioral statistics. For example, it can include the cache misses that occur during the operating system's execution and attribute these misses to the responsible procedures or data structures. This detailed information helps focus performance tuning efforts on the most troublesome areas of the operating system.

Operating system development has long been hindered by the lack of tools capable of aiding the porting effort or providing detailed behavioral information. The few simulation tools that are capable of investigating operating system behavior suffer from several limitations. Advances in software instrumentation techniques have enabled the trace-driven simulation of some operating systems [Chen93] [Perl97]. However, software instrumentation results in an operating system that is both larger and longer running than in its original form. As described in [Chen93], the resulting time- and memory-dilation affects the accuracy of the trace and thus of any derived performance data.

Hardware monitoring mechanisms can also collect traces of operating system activity. While less intrusive than software instrumentation, the monitors only provide information about the limited types of hardware activity that they observe. For example, in [Chapin95a], the hardware-based trace collection mechanism only captures memory reference activity occurring on the system bus, limiting the trace's use to investigations of an operating system's memory system performance. As a result of the limited applicability of simulation tools to operating system development, kernel programmers often resort to modifying the operating system code to observe its own behavior. Operating systems are often littered with code to count the invocations of particular procedures or to measure the performance of locks and semaphores. The information collected by this inserted code can help indicate some simple operating system performance problems, but does not provide more comprehensive hardware-level performance data such as cache or processor pipeline behavior. Furthermore, the code modification technique only applies to examining operating system performance on existing machines, and does not necessarily help indicate where performance problems will arise on future platforms.

2.2.3 Application performance tuning

A third important type of computer systems research is application performance tuning. In this domain, programmers attempt to discover and eliminate performance problems with the sole goal of speeding up an application's execution. To determine and implement the most effective code modifications, a programmer requires detailed information regarding the application's behavior.

The complete machine simulation approach also provides benefits to this type of research. Complete machine simulation's extensive workload support allows its use in the investigation of almost any application. This includes complex, multi-process applications such as CAD tools, database management systems, and web servers. Additionally, complete machine simulation's comprehensive data reporting provides detailed information regarding every aspect of the application's behavior. For example, it can report a variety of hardware-related performance problems that occur during the application's execution such as mispredicted branches, cache misses, and processor pipeline stalls. These problems can be responsible for substantial application performance loss, and knowledge of their occurrence is essential to eliminating them. For example, if a programmer discovers that a specific data structure is experiencing significant cache misses, they can often restructure it to improve its cache locality and improve the application's performance. Complete machine simulation also reports operating system-related performance problems such as excessive page faults, expensive system call invocations, and poor process scheduling. These problems can be responsible for a substantial portion of an application's execution time and knowledge of their occurrence can often help reduce their impact.

One of the most common techniques for obtaining application performance information is the use of profiling tools. Profiling tools use a variety of techniques to determine where an application spends most of its execution time. These profiling tool typically use a computer system's hardware timer to periodically sample the CPU's program counter. This information provides a statistical indication of the most heavily executed sections of application code. This basic information can help focus an application tuner's attention on the most important application procedures or loops. However, profiling tools only indicate where an application spends most of its execution time, and does not indicate the specific source of performance problems. Many recent CPU's incorporate counters that track various processor events as they occur during an application's execution. These counters are integrated directly onto the CPU, and track events such as mispredicted branches or data cache misses at high speeds and with minimal application perturbation. This additional information improves upon traditional program counter-based profiling, incorporating basic memory system and processor pipeline behavioral data into the reported data.

For even more detailed performance information, investigators sometimes employ user-level simulators. As mentioned above, these tools provide information about an application's behavior, but omit the operating system activity that would normally occur during its execution. From an application's standpoint, the execution environment modeled in user-level simulators is a significant improvement over actual computer systems. For example, system calls occur instantaneously, having no impact on the application's execution time. Additionally, the application is always actively "scheduled" and is fully resident in memory. As a result, the application does not face any of the performance problems associated with execution in multiprogrammed systems. Furthermore, operating system code is never executed, allowing the application to avoid competition for the processor's limited instruction and data cache space. Unfortunately, this execution environment is not representative of existing computer systems, and these performance "benefits" impact the quality of data that user-level simulators can provide. As a result, the behavior of many important applications tends to be less well understood, hindering efforts to improve their performance.

2.3 Summary

The complete machine simulation approach was originally adopted to satisfy very specific investigative needs, but its flexible hardware modeling capability and ability to observe a computer's operating system behavior has proven useful to other fields of computer systems research as well. This chapter has described several specific benefits that complete machine simulation provides for the fields of hardware design, operating system development, and application performance tuning. Chapter 6 revisits complete machine simulation's benefits and describes several specific investigations that they have enabled.

Chapter 3

Functional and Flexible Hardware Simulation

The previous chapters introduce complete machine simulation and the benefits that it provides to several fields of computer systems research. This is the first of three chapters that describe the implementation features that allow complete machine simulation to provide these benefits.

The most important goal of complete machine simulation is to support the investigation of a large class of workloads as they execute on highly configurable computer hardware. This chapter describes how complete machine simulation's functional and flexible hardware simulation approach helps achieve this goal. The first part of this chapter describes the two primary components of complete machine simulation's hardware simulation approach; providing software-visible functionality and supporting configurable implementation details. The second part of this chapter describes how SimOS satisfies these hardware simulation requirements. SimOS takes a modular approach to simulating a computer, allowing the development and inclusion of multiple CPU, memory system, and I/O device simulators. Each of these simulated components provides the basic functionality required to execute operating systems and applications, but provides this functionality while modeling widely varying hardware implementation details.

3.1 Hardware simulation tasks

Complete machine simulation differs from most simulation approaches in that it models all of the hardware typically found in a computer system. More specifically, complete machine simulation models computer hardware in enough detail to support the execution and investigation of operating systems and application programs. This section describes the two primary components of this simulation task; providing software-visible hardware functionality and supporting configurable hardware implementation details.

3.1.1 Providing software-visible functionality

One goal of complete machine simulation is to support the investigation of *unmodified* operating systems and application programs. This goal is quite important as it allows the investigation of a much wider range of workload than is possible with many existing tools. For example, most operating systems and commercial software packages are shipped in a binary format without any publicly available source code. As a result, simulation tools that recompile or instrument an application's source code are unable to examine their behavior. As described in the previous chapter, supporting unmodified workload binaries also allows complete machine simulation to avoid the time- and space-dilation effects that accompany instrumentation-based tools.

To satisfy this goal, a complete machine simulation must be compatible with the hardware that the workload normally runs on. Specifically, a complete machine simulator must export the same hardware interfaces that are normally visible to the workload binaries and provide the hardware functionality expected by interactions with this interface. The rest of this section describes these expected interfaces and the corresponding hardware functionality that a complete machine simulator must provide.

CPU functionality

Operating systems and applications expect significant functionality from a computer's CPU, the most fundamental of which is the proper execution of instructions. This includes normal user-level instructions as well as those that are only accessible in "privileged" processor modes. Additionally, operating systems expect a memory management unit

(MMU) that relocates every virtual address to a location in physical memory or generates an exception if the reference is not permitted (e.g. a page fault). The operating system also expects the CPU to inform it of other exceptional events such as arithmetic overflow, the use of privileged instructions in user mode, or the occurrence of external device interrupts. Furthermore, multiprocessor workloads expect all of this functionality to be replicated, allowing the parallel execution of several independent instruction streams.

Memory system functionality

Operating systems and applications also expect their underlying hardware to provide some type of memory system that coordinates communication between the CPU, main memory, and other devices. For example, the memory system must read and write the contents of main memory in response to certain CPU instructions. Additionally, the memory system must provide an I/O address space to enable communication between the CPU and I/O devices. The I/O address space allows software, typically operating system device drivers, to access the registers that control and query I/O devices. The memory system must also transfer data between devices and main memory in response to device register accesses, and the specific method of data transfer often depends on the device itself. For example, programmed I/O devices expect the memory system to move data to or from the device a single byte at a time. Other devices support direct memory access (DMA), where the CPU informs the I/O device of a location in main memory and an amount of data to transfer. The device transfers data directly to or from memory, interrupting the CPU to indicate its completion. Regardless of the implementation, the memory system must manage this data transfer to ensure that main memory always reflects the proper machine state.

I/O device functionality

In addition to coordinating communication with I/O devices, a complete machine simulator must provide the functionality of the I/O devices themselves. These devices include at least a timer that interrupts the CPU at regular intervals, a storage device which contains the operating system and application files, and a console for interaction with the user. Some workloads may expect additional I/O device functionality, such as that of a networking card or graphics chip. Whether writing data to a SCSI disk, receiving typed commands from the system console, or communicating with another computer over network, workloads require significant utility from a computer's I/O devices, and complete machine simulation must provide the expected functionality.

3.1.2 Supporting configurable implementation details

In addition to supporting the functional execution of a workload, a complete machine simulator must also enable a detailed investigation of the workload's execution behavior. Specifically, it should provide information about a workload's behavior as it executes on a specific computer configuration. To provide this information, a complete machine simulator must model specific hardware implementation details while providing hardware functionality. Furthermore, a complete machine simulation must support highly configurable implementation details to provide this information across a wide range of potential computer configurations.

The distinction between hardware functionality and hardware implementation details deserves discussion. Hardware functionality relates to a basic architectural specification and is not specific to any single machine implementation. In contrast, hardware implementation details do not affect the functional execution of a workload, but determine how the hardware behaves and how quickly it completes this execution. For example, CPU functionality consists of applying the an instruction's behavior to its registers and to main memory according to a well-defined architectural specification. In contrast, CPU implementation details determine how long it takes to execute these instructions and includes effects such as the processor's pipeline, clock rate, and branch prediction scheme.

As another example, memory system functionality consists of coordinating communication between the CPU, main memory, and I/O devices, but implementation details determine the latency of this communication. In the case of memory references, latency includes the effects of caches, bus speed, arbitration protocols, queuing delays, and even DRAM characteristics. These implementation details do not provide any functionality that is required for workload execution, but play a determining role in the workload's execution behavior. As a final example, disk drive functionality simply consists of reading or writing data as requested. In contrast, disk drive implementation details assign latencies to this activity and require a simulator to model factors such as SCSI controller delays, disk head seek time, and disk block transfer time. In each case, hardware implementation details are not necessary for the functional execution of an operating system and application programs, but are essential to understanding a workload's behavior on a specific computer configuration.

3.2 SimOS hardware simulation

This section describes the SimOS approach to satisfying complete machine simulation's hardware simulation requirements. The first part of this section describes the hardware functionality that SimOS provides and some of the more interesting aspects of its implementation. The second part of this section describes SimOS's modular approach to hardware simulation and how this modularity supports highly configurable hardware implementation details.

3.2.1 Providing software-visible functionality

SimOS models the hardware of modern computer systems in enough detail to boot and run IRIX, the Silicon Graphics, Inc. implementation of Unix System V Release 4. In fact, a SimOS-modeled machine is binary-compatible with actual machines shipped by Silicon Graphics, allowing it to execute the wide assortment of commercial applications designed for this platform. To support the execution of IRIX and its applications, SimOS provides the same hardware functionality that is visible to software on real Silicon Graphics machines. The rest of this section describes SimOS's hardware functionality and the more interesting aspects of its implementation.

CPU functionality

SimOS supports execution of the MIPS-IV instruction set, including arithmetic, floating point, and privileged "co-processor 0" instructions. SimOS also provides the virtual address to physical address translations that occur in a processor's memory management unit (MMU). For the MIPS architecture this means implementing the associative lookup of the translation look-aside buffer (TLB), including raising the relevant exceptions if the translation fails. SimOS also takes exceptions on events such as arithmetic overflow, the use of privileged instructions in user mode, or the occurrence of external processor interrupts. SimOS can also simulate multiple CPU's simultaneously to enable the execution of multiprocessor workloads.

Memory system functionality

As illustrated in Figure 3.1, SimOS manages the communication between CPU's, main memory, and I/O devices similarly to a normal memory controller. Communication between the CPU and main memory is straightforward. SimOS maintains the contents of the target machine's memory by allocating memory in its own address space. This allocated memory is sized according to the amount of main memory "installed" on the target machine, and simulating the functionality of loads and stores to main memory simply involves reading and writing this allocated memory. Communication between CPU's and I/O devices is slightly more complicated. The Silicon Graphics/IRIX platform utilizes memory-mapped I/O where device registers are mapped into a portion of the physical address space. This allows IRIX device drivers to use normal CPU read and write instructions to access device registers. SimOS uses a hash table called a *device registry* to communicate device register accesses to the appropriate I/O device simulator routine. SimOS provides the expected I/O device functionality by mapping an appropriate device simulator routine at every location in this I/O address space that IRIX device drivers utilize. In response to these device driver requests, the simulated I/O devices provide varied functionality, interrupting the processor as appropriate. As on a real machine, I/O

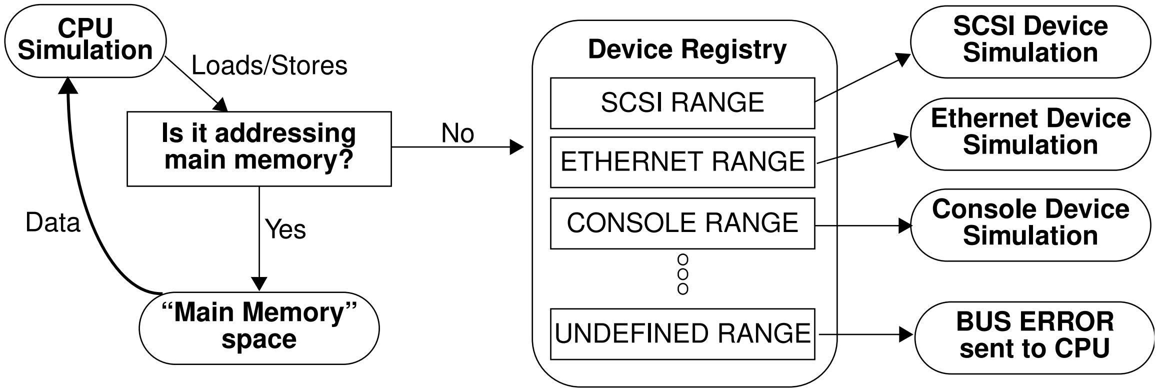

Figure 3.1. SimOS memory system functionality

address space references that do not correspond to "installed" I/O devices result in bus error exceptions.

I/O device functionality

In addition to coordinating communication between the CPU, main memory, and I/O devices, SimOS provides the functionality of the I/O devices themselves. The implementation of this device functionality is particularly interesting as it often requires SimOS to act as a gateway between the simulated machine and the non-simulated world. This interaction takes many forms, but in each case SimOS exploits the functionality of its host platform to provide the expected functionality. For example, SimOS provides the functionality of a console device by communicating with a real terminal on the host machine. SimOS outputs console writes to the terminal and translates typing at the terminal into console input. This allows a user to interact with the simulated machine just as if it were a normal hardware-based machine.

As a second example, SimOS provides ethernet functionality by multiplexing the simulated machine's network activity with that of the host machine. Ethernet packets sent from the SimOS-modeled machine are forwarded through the host machine's ethernet, and return packets are routed back to the simulated machine. Furthermore, we have allocated a range of IP addresses for SimOS-modeled machines, allowing them to act as nodes on the internet and enabling a wide range of network communication possibilities. For example, we have configured SimOS-modeled machines as NFS clients and servers to ease the transfer of large files between SimOS and hardware-based machines. Similarly, SimOS-modeled machines can communicate over the X-windows protocol to display their program output on a remote machine's display. We have even configured a SimOS-modeled machine as a web server that provides the SimOS user-guide web pages to the internet community.

As a final example, SimOS uses the host machine's filesystem to provide the functionality of a hard disk, maintaining the disk's contents in a file on the host machine. Reads and writes to the simulated disk become reads and writes of this file, and DMA transfers simply require copying data from the file into the portion of the simulator's address space representing the target machine's main memory. To create a new disk that can be mounted by a SimOS-modeled machine, a user simply creates the appropriate disk image in a file on the host machine.

SimOS's implementation of disk functionality is particularly interesting because of its support for sharing filesystem images among several users. The disk images required to boot an operating system and execute complex applications can occupy several gigabytes of space on the host machine's filesystem, making it desirable to share them among investigations. However, sharing disk images can be troublesome because workloads typically modify files during their execution. If a workload's modifications were saved to the host machine's filesystem, it could affect any future investigations that share the disk image. Furthermore, sharing the disk image file among multiple concurrent simulations can result in file consistency problems.

To avoid these problems, SimOS supports a *copy-on-write* disk simulation mode. Illustrated in Figure 3.2, the copy-on-write mode allows multiple instances of SimOS to run concurrently by maintaining all target machine disk modifications in memory rather than to the shared disk image file. Subsequent target machine disk reads obtain the most recent filesystem data by checking the list of modified disk blocks first, only accessing the original disk image for blocks that have not yet been written. In this example, the first instance of SimOS returns "ccc" in response to the read of disk block 3 while the second instance returns "ghi". The copy-on-write mechanism allows any number of investigations to share a single disk image file, easing the process of workload creation and avoiding excessive host filesystem space requirements.

3.2.2 Supporting configurable implementation details

In addition to providing the hardware functionality required to execute operating system and application software, SimOS supports significant flexibility in the modeling of hardware implementation details. Illustrated in Figure 3.3, SimOS takes a modular approach to machine simulation, encouraging the development and inclusion of multiple CPU, memory system, and I/O device implementations. While each simulated hardware component must provide the functionality expected by the operating system and

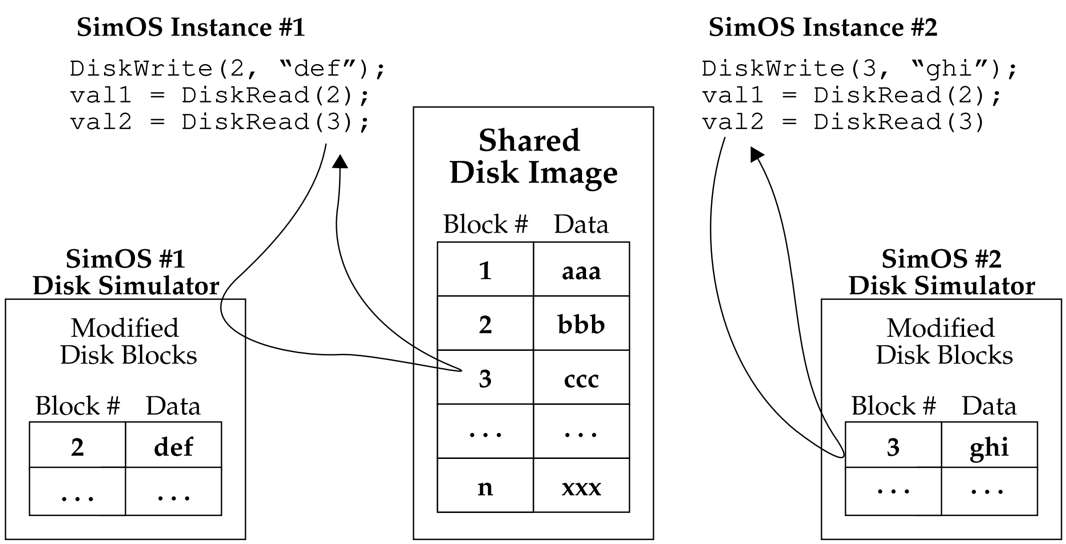

Figure 3.2. SimOS's copy-on-write disk simulation mode

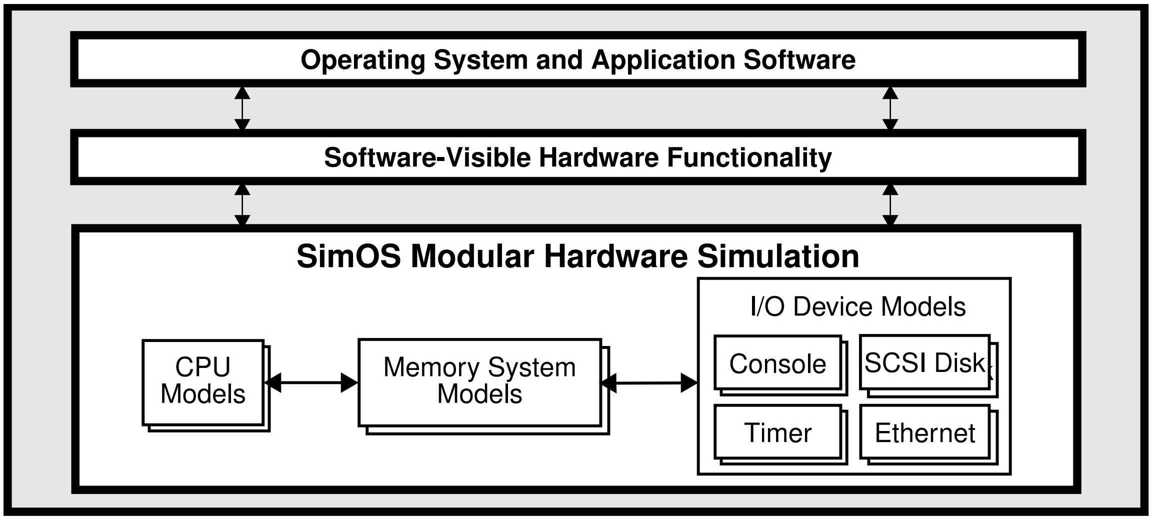

Figure 3.3. Modular hardware simulation

application software, it can provide this functionality while modeling radically different implementation details. To simulate a very specific architectural implementation, an investigator can either configure existing hardware component models or, when implementation detail changes are more significant, create and include completely new models. The next chapter describes several specific examples of hardware component models exhibiting a range of implementation details.

To support the development and inclusion of modular hardware simulators, SimOS provides well-defined interfaces between each major hardware component, and these interfaces are similar to those found in existing computer systems. For example, the interface between a CPU and memory system model is similar to that of modern system buses. The CPU model submits load and store requests to the memory system, along with the physical address that should be accessed. The memory system model determines the latency of this request, eventually returning the requested data. The memory system has tremendous flexibility in determining this memory request latency. For example, a simple memory system model could use a constant latency for each request while a more detailed model could determine the latency by simulating multiple levels of caches, bus arbitration protocols, and other implementation details.

As described above, memory system interface with I/O device models through the device registry, where accesses to device registers are forwarded to specific I/O device simulators.

As a result, IRIX device drivers determine the exact interface for developing new I/O device models. Each I/O device model must provide the expected functional response to the device driver's device register accesses, but it can provide this functionality with varying implementation details. Additionally, we have written new device drivers for IRIX that provide significant I/O device modeling flexibility. For example, we have written a fairly generic disk device driver for IRIX that communicates through the use of SCSI commands and disk status queries. Each disk model implementation must provide the appropriate response to each command and query, but has significant flexibility in determining the latency and behavior of the disk in providing these responses. We have also written fairly generic device drivers for communication with ethernet and console hardware, allowing SimOS to flexibly model implementation details for these I/O devices as well. In addition to providing increased flexibility, creating SimOS-customized device drivers eased the I/O device modeling effort by allowing us to avoid supporting many of the very esoteric hardware details built into shipping device drivers and to concentrate on modeling more basic device functionality.

The ability to develop and incorporate multiple implementations of each computer hardware component provides significant machine modeling flexibility and allows an investigator to create an entire simulated machine simply by selecting from a list of existing component models. If the existing models do not fulfill the particular needs of the investigation, the investigator can develop additional components and easily incorporate them into the SimOS infrastructure. SimOS's modular hardware simulation approach provides other important benefits as well. As described in the next chapter, the ability to model the same hardware functionality with different amounts of implementation detail is essential to high-speed simulation.

3.3 Summary

The most important goal of complete machine simulation is to support the investigation of a large class of workloads as they execute on highly configurable computer hardware. To achieve this goal, complete machine simulation must provide all of the hardware functionality that is visible to workloads and support highly configurable hardware implementation details. The SimOS implementation of complete machine simulation demonstrates how a modular hardware simulation approach satisfies these requirements. SimOS provides well-defined interfaces for the development and inclusion of multiple hardware component models. Each model provides the functionality required to execute operating system and application code, but can provide this functionality while modeling the hardware implementation details of interest to a particular investigation.

Chapter 4

Dynamically Adjustable Simulation Speed and Detail

Another important characteristic of complete machine simulation is its ability to provide appropriately detailed information regarding a computer system's hardware and software behavior. However, the benefits of this information are significantly reduced if it takes an excessively long time to obtain. This chapter describes how complete machine simulation can efficiently provide simulation results. The first part of this chapter describes the challenge of quickly obtaining simulation data and how the ability to dynamically adjust the level of simulation speed and detail addresses this challenge. The second part of this chapter describes how complete machine simulation can implement dynamically adjustable simulation speed and detail. The SimOS implementation of complete machine simulation provides three general simulator execution modes exhibiting specific speed-detail trade-offs, and allows a user to switch between them during the course of a workload's execution. The final part of this chapter describes SimOS's use of high-speed machine emulation technology and more traditional simulation techniques to implement each simulator execution mode.

4.1 The performance challenge

Every type of computer system investigation benefits from obtaining behavioral data as quickly as possible. For example, hardware design requires examining several potential configurations and determining which implementation features provide the most benefit. The faster that a simulator can provide the required performance data, the larger the design space that an investigator can evaluate within a given time frame. As a result, faster simulation tools can help lead to better hardware designs. Similarly, fast turnaround time is an essential part of application performance tuning. When attempting to improve the performance of an application, a programmer typically makes one or more algorithmic changes and then obtains data regarding the performance impact of these changes. This process is repeated as many times as possible within the available time period. The more times that this feedback cycle can be iterated, the better that the end application performance will be.

The use of complete machine simulation would appear to be at odds with the goal of obtaining data as quickly as possible. Detailed simulation of a computer is inherently slow as it attempts to accurately model the behavior of hardware components completely in software. As an example, an RTL model of a CPU provides cycle-accurate performance detail, but is only capable of simulating the execution of a few hundred cycles every second. As a result, modeling the execution of a single instruction on a particular target machine can require many millions of instructions on the host machine. The resulting *simulation slowdown* causes significant delays in obtaining the desired behavioral information for an entire workload. Even significantly less detailed hardware models impose restrictive slowdowns. For example, the Mipsy CPU simulator described below models few processor details, yet still executes a machine's instructions more than 200 times slower than a hardware implementation would.

This information delay is worsened by the fact that complete machine simulation is designed to investigate workloads that are typically long-running even on a non-simulated computer. For example, Chapter 6 describes an investigation of a Sybase database server running a transaction processing workload. This workload requires execution of over 20 billion instructions just to boot the operating system and initialize the database server and client programs. Investigation of even the most basic workloads often requires a clean boot of the operating system, a process that takes IRIX over a half billion instructions. The combination of longer running workloads and slower "hardware" can result in prohibitively long simulation times. For example, just the preparation of the database workload mentioned above would require several days of execution time on the Mipsy CPU simulator, and more detailed simulation would take even longer. Such a time commitment is a significant hindrance to any computer system investigation. The performance challenge facing complete machine simulation is thus to support both the fast and detailed investigation of long-running workloads.

4.2 The solution: Dynamically adjustable simulation speed and detail

Most performance investigations do not require extremely accurate and detailed information across the entire execution of a workload, and this provides an opportunity to obtaining simulation results quickly. Preparing a complex workload for investigation usually requires simulating large amounts of "uninteresting" sections of execution such as booting the operating system, reading applications and their data in from a disk, and initializing the workload for investigation. In these cases, only the proper functionality of the simulated platform is required, and detailed timing information within these sections is typically ignored. Once the workload has reached a more interesting section of execution, detailed behavioral information becomes desirable. This trait of *localized interest in detail* is characteristic of most investigations and provides an excellent opportunity for simulation speed gains.

In all simulation tools, there is an inherent compromise between the amount of detail that is modeled and the speed at which the simulator executes. This *speed-detail trade-off*, illustrated in Figure 4.1, is particularly relevant for complete machine simulation. At one end of the spectrum is an extremely detailed but extremely slow hardware simulation model. For example, RTL and other gate-level simulators model a hardware implementation with a great deal of accuracy, but provide hardware functionality at a very slow speed. At the other end of the spectrum is a simulation model that provides only the

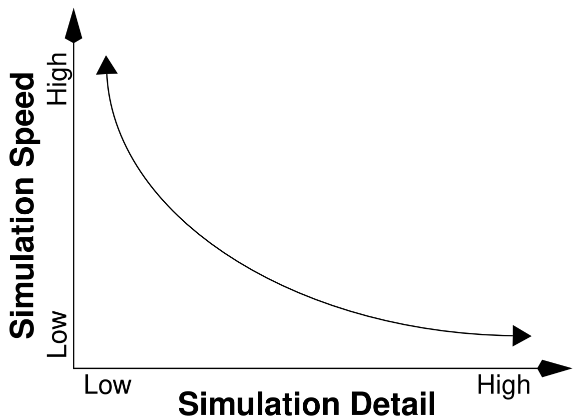

Figure 4.1. The simulation speed-detail trade-off

hardware detail required to support workload execution. This class of simulator can quickly support the required functionality, but is not faithful to any specific hardware implementation. Between these extremes lie a number of intermediate simulator models, each providing varying levels of speed and detail.

Localized interest in detail and the ability to provide hardware functionality at different levels of simulation speed and detail are the foundation of providing effective behavioral information as quickly as possible. Our approach is to allow an investigator to change the speed and detail characteristics of the complete machine simulator during the course of workload execution. For example, the simulated machine could be run in a high-speed, low-detail mode to quickly execute through the operating system boot and other uninteresting portions of a workload. When the workload reaches a more interesting section, the investigator can switch the simulator into a slower but more detailed mode for collecting data. This dynamic adjustment capability allows complete machine simulation users to select the exact level of detail required at each stage of an investigation, maximizing the speed at which useful data is obtained. The end result of this customization is effective computer system behavioral information that is obtained as quickly as possible.

4.3 Implementation

Completely adjustable control over complete machine simulation's speed and detail characteristics is an attractive concept, but difficult to implement. To provide similar utility, complete machine simulation uses its modular hardware interfaces to create multiple machine implementations, each occupying specific points on the speed-detail curve. This section describes the specific points on the speed-detail curve that we have found to be most valuable and how complete machine simulation allows an investigator to switch between them during the course of workload execution.

4.3.1 Simulator execution modes

As described in the previous chapter, a complete machine simulator combines several individual hardware component models to form a simulated computer. Furthermore, each hardware component can have multiple implementations each making different speed-detail trade-offs. These components determine the simulated machine's final speed and detail characteristics. A number of hardware component model combinations are possible, but only certain combinations are practical. For example, connecting an extremely detailed CPU model to a very simple and low-detail memory system model would reduce the overall accuracy of behavioral information. Furthermore, any gains in speed to be achieved through the use of the faster memory system model will likely be overshadowed by a significantly slower CPU model. Consequently, certain groupings of hardware models exhibiting similar speed and detail characteristics are more effective than others. We call these groupings *simulator execution modes* and have found three general modes to be particularly useful: *positioning mode*, *rough characterization mode*, *and accurate mode*.

Positioning mode

There are several times during the course of a workload's execution when only the functionality of a system is required. For example, preparing a complex workload for investigation usually requires simulating large amounts of uninteresting execution such as booting the operating system, reading data from a disk, and initializing the workload.

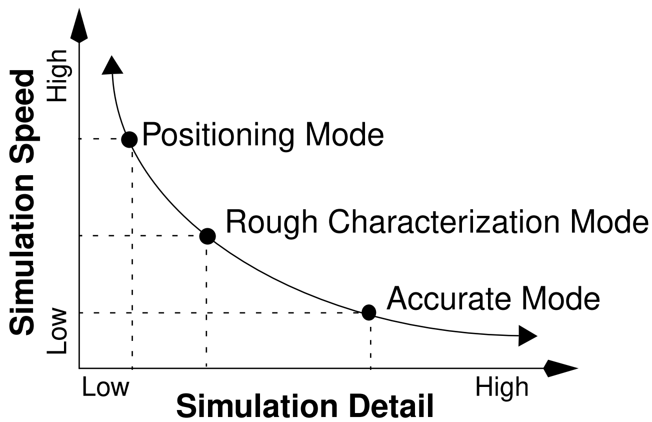

Figure 4.2. Simulator execution modes

Furthermore, issues such as memory fragmentation and file system buffer caches can have a large effect on the workload's execution. Many of these effects are not present in a freshly booted operating systems; they only appear after prolonged use of the system. Realistic studies require executing past these "cold start" effects and into a more representative steady state for detailed investigation. We group these requirements into the category of workload positioning, and provide a *positioning mode* to address them. In positioning mode, complete machine simulation provides the very basic functionality of the target platform as quickly as possible. The only requirement is to correctly execute the workload, and an investigator has little interest in the simulated system's detail or faithfulness to any existing implementation,

Rough characterization mode

The speed of positioning mode is essential for the setup and initialization of complex workloads, but the lack of hardware implementation details makes it unsuitable for obtaining any useful behavioral information. To gain more insight into a workload's behavior, we provide a *rough characterization mode* that maintains high simulation speed yet provides timing estimates that approximate the behavior of the machine. This mode is particularly useful for performing a high-level characterization of workloads to determine first-order bottlenecks. For example, it provides enough detail to determine if a workload is paging, I/O bound on a disk, or suffering large amounts of memory system stall.

Additionally, rough characterization mode is fast enough that it can provide this information over relatively long periods of workload execution time.

Accurate mode

One of the most important features of complete machine simulation is the ability to model a computer configuration with significant accuracy and provide very detailed information regarding its behavior. This information is essential to almost every type of computer system investigation, and complete machine simulation provides an *accurate mode* to obtain it. However, the detailed behavioral information provided by the accurate mode results in speeds that are far too slow to execute most workloads in their entirety.

4.3.2 Dynamic simulator execution mode selection

The existence of different simulation execution modes provides little utility if they can not be effectively utilized. As mentioned in Section 4.2, they key to quickly obtaining simulation data is the ability to *dynamically* adjust the speed-detail characteristics of the simulator. To provide this ability, complete machine simulation must include some means of switching between simulation modes during workload execution. The key to providing this capability is transferable hardware state.

Different models of a hardware component can provide a variety of statistics and implementation details, but all must provide the same basic hardware functionality. To provide this functionality, there is a common hardware state that each model must maintain. For example, all CPU models must keep track of the current value of their registers, regardless of the clock speed or pipeline that they simulate. Similarly, models of main memory and disks must always maintain their correct contents so that reads receive the proper data. In addition to maintaining this core hardware state, we require each hardware model to support the exchange of this core state with other models of the same hardware component. As a result, an investigator can switch between simulator execution modes at any point during a workload's execution. For example, in a study involving a database transaction processing workload, an investigator could use the positioning mode to quickly boot up the operating system and warm up the contents of the file cache. Upon completion of this workload positioning, simulated time could be temporarily suspended while the core state of the CPU, memory, and I/O device models is transferred into the rough characterization mode hardware models. Once this transfer is complete, execution of the transaction processing workload resumes, only with more detailed modeling of system behavior. If even more detailed information regarding the workload's execution is desired at a later point, complete machine simulation can again transfer hardware state, this time from the rough characterization mode hardware models into those of the accurate mode.

Together these dynamically selectable simulator execution modes provide workload execution control similar to that found on a VCR. Positioning mode is similar to fast-forward, allowing a user to quickly pass over sections of a workload's execution that they find less interesting. Rough characterization mode is similar to the normal play mode of a VCR, presenting a decent view of the workload's execution without an extraordinary time commitment. Finally, the most accurate modes are best compared to slow-motion playback. They provide a very detailed view of computer system behavior, but at often prohibitively slow speeds.

4.4 SimOS's simulator execution modes

SimOS provides clear interfaces between each of the major hardware components in the simulated computer system, allowing it to incorporate different models of each hardware component. While each of the models provide the basic hardware functionality required by IRIX and its applications, they differ significantly in the type and amount of detail that they provide. As a result, they also differ in the type of statistics that they can collect and the speed at which they support a workload's execution. This section describes how SimOS models these hardware components to provide each simulator execution mode.

4.4.1 Positioning mode

The goal of positioning mode is to provide the hardware functionality required to run workloads and to provide this functionality as fast as possible. As such, SimOS's positioning mode models very few timing and implementation details and can provide an investigator with only minimal workload behavioral information. As described in the

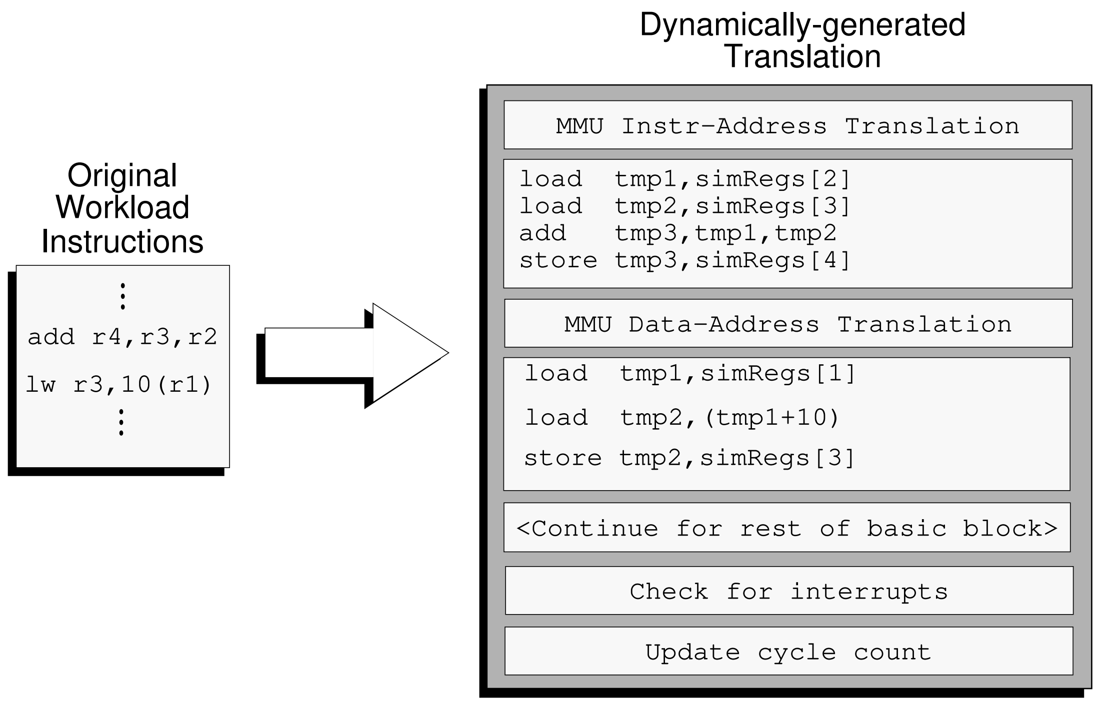

Figure 4.3. Embra's dynamic binary translation

previous chapter, supporting operating system and application program execution requires a complete machine simulator to implement the required hardware interfaces. In providing the expected functionality, a few components play a determining role in the overall speed of the simulated machine. The CPU, cache, and memory system account for the bulk of simulation costs, and SimOS includes the Embra hardware simulator [Witchel96] to minimize this cost. Illustrated in Figure 4.3, Embra uses the dynamic binary translation approach pioneered by the Shade system [Cmelik94]. Dynamic binary translators convert blocks of instructions into code sequences that implement the effects of the original instructions on the simulated machine state. The translated code is then executed directly on the host machine. Using translation caching and other optimizations, Embra can execute uniprocessor workloads with a slowdown of less than a factor of ten, orders of magnitude faster than conventional simulation techniques.

Embra extends the techniques of Shade to support the functionality required in complete machine simulation. These extensions include modeling the effects of the memory-management unit (MMU), privileged instructions, and the trap architecture of the

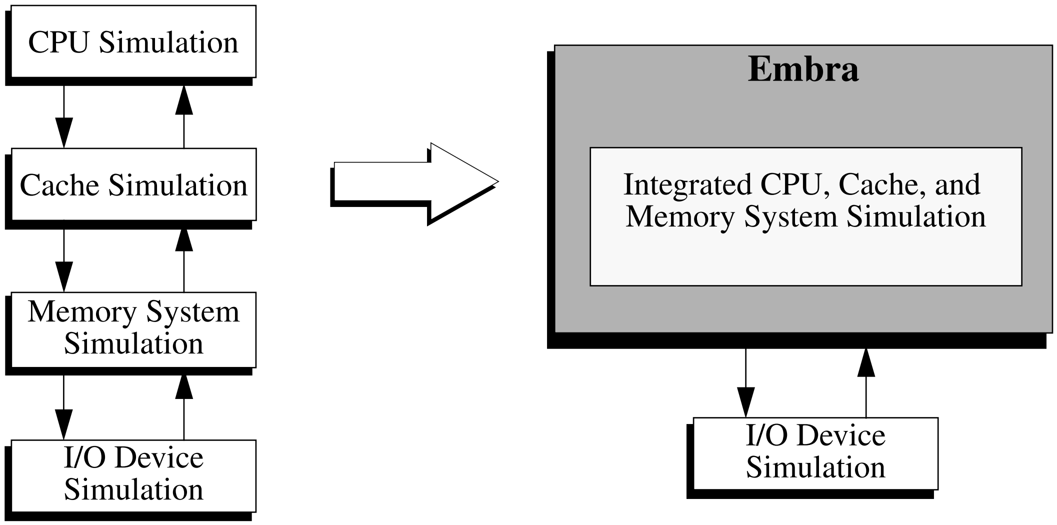

Figure 4.4. Embra's subsumption of multiple hardware component simulators

machine. The approach used in Embra is to handle all of these extensions with additional code incorporated into the translations. For example, Embra augments the minimal binary translation with code that implements the associative lookup done by the MMU on every memory reference. Embra also extends the techniques of Shade to efficiently simulate multiprocessors. Embra connects emitted code translations from each simulated CPU at a very coarse granularity, emulating several hundred instructions for one simulated processor before switching to the next simulated processor's execution. The simulated processors' notions of time are poorly synchronized, but this is acceptable in positioning mode where speed is far more important than accuracy and detail.

In addition to its use of binary translation, Embra improves the speed of providing hardware functionality by avoiding the typical SimOS approach of clean hardware model interfaces. As depicted in Figure 4.4, Embra subsumes the functionality of multiple hardware components. For example, rather than invoking a separate memory system model in order to satisfy a memory reference, Embra incorporates the functionality of a simple memory system directly into its binary translations. This allows Embra to avoid the overheads caused by the use of flexible software module interfaces. Because I/O device activity is much less frequent than CPU and memory system activity, Embra continues to use the modular I/O device interface without significant performance impact.

As an additional optimization, SimOS's positioning mode uses Embra configured to model only the hardware components of the system that are necessary to correctly execute the workload. No attempt is made to model hardware features that are invisible to the software. For example, a processor's pipeline and cache hierarchy play a significant role in the performance of a workload, but do not provide any direct functional utility. To avoid the overhead of simulating these hardware features, Embra models no processor pipeline behavior and references to memory succeed instantaneously. Similarly, positioning mode avoids any detailed I/O device simulation. For example, the disk model is configured to satisfy all requests immediately. In addition to avoiding the performance impact of simulating the disk, this optimization helps speed through uninteresting sections of a workload's execution. Normally, accesses to a disk result in large delays, and a user-level process is descheduled during the latency. During this delay, the operating system either schedules another user-level process or executes in an "idle" loop, waiting for the disk access to complete. By omitting disk access latency, SimOS's positioning mode avoids the portion of a workload's execution time that is spent in the idle loop. As a result, it reduces the time it takes to reach a more interesting portion of a workload.

The initial implementation of SimOS contained an additional high-speed positioning mode based on direct execution of the operating system and the applications on the host platform. The direct-execution approach, described in [Rosenblum95], supported very high speed machine emulation, but was removed in 1996 in favor of the binary translation approach. Binary translation was chosen because it is more amenable to functional extension and cross-platform support than the direct execution approach.

4.4.2 Rough characterization mode

Rough characterization mode is designed to be a compromise between the accurate and positioning modes. As such, it must provide very high-level behavioral information, but provide this information as quickly as possible. SimOS's rough characterization mode extends the functionality of positioning mode by tracking instruction execution time, approximating cache activity, and modeling basic I/O device behavior.

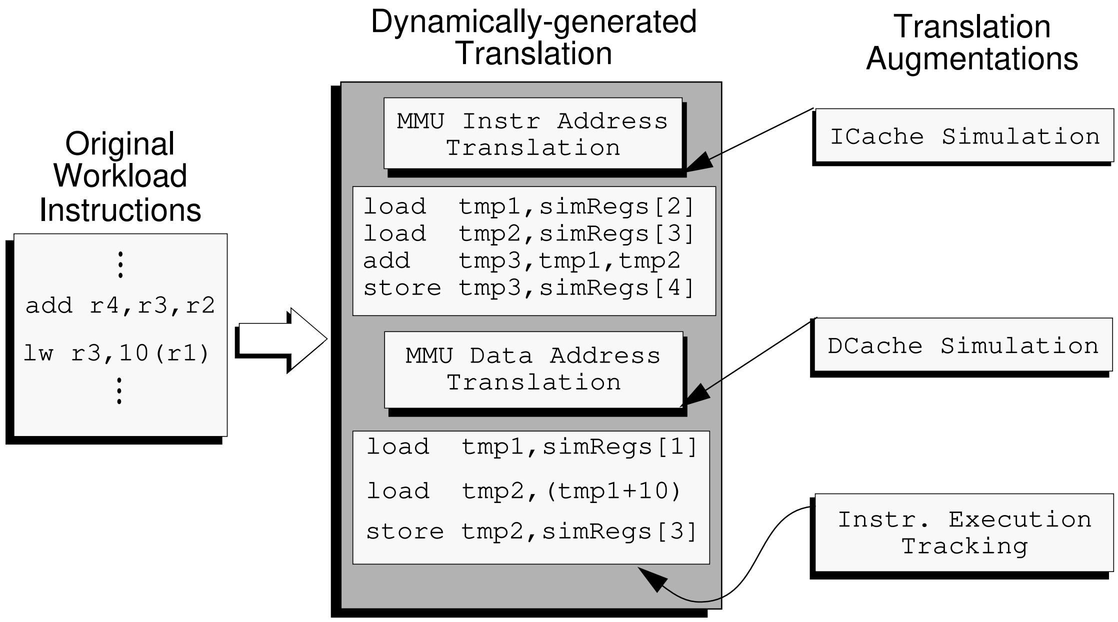

Figure 4.5. Extending Embra's translations with additional detail

SimOS implements a rough characterization mode through extensions to Embra. As shown in Figure 4.5, dynamic binary translation is flexible enough to customize emitted translations for more detailed modeling of the target machine. For example, Embra augments its translations to check whether memory accesses hit in a fixed-latency, unified instruction and data cache similar to the second-level cache of many machines. This data provides an investigator with a high-level understanding of the memory system behavior of a workload. In pursuit of speed, these cache-modeling augmentations are highly optimized to make the common case of cache hits as fast as possible. Additionally, Embra augments its translations to track the execution of instructions, providing basic workload profile information.

To approximate the I/O behavior of a workload, rough characterization mode interfaces with simple, fixed-latency device models. For example, the rough characterization mode disk model charges a fixed latency for each request that a workload makes. This allows it to estimate the impact of I/O activity in a workload while avoiding the simulation overhead required to model the seek, rotation, and transfer time of a more detailed model.

Main Loop:

```
While (TRUE) {
   cycle++;
   HandleInterrupts();
   inst = ReadMem(PC, INST);
   ...
   switch(opcode(inst)) {
   case ADD:
      reg3 = reg1 + reg2;
   case BEQ:
      if (reg1 = reg2)
         newPC = targetAddr;
   case LD:
      va = ComputeAddr(inst);
      reg3 = ReadMem(va, DATA);
      ...
   case ...:
      ...
   }
   if (newPC) PC = newPC;
   else PC = PC + 4;
}
```

ReadMem(vAddr, type);

```
pAddr = Translate(vAddr);

if (type == INST)
   d=ReadICache(pAddr,...);
else
   d=ReadDCache(pAddr, ...);
return d;
```

Translate(vAddr);

```
if (inTLB(vAddr, &pAddr)) {
   if (valid)
      return pAddr;
   else
      EXCEPTION(INVALID);
} else {
   EXCEPTION(TLB_MISS);
}
```

Figure 4.6. Structure of the Mipsy CPU simulator.

4.4.3 Accurate mode

Accurate mode is what people typically think of when discussing simulation. As mentioned above, the goal of this mode is to model a specific machine's hardware with significant accuracy and detail and provide behavioral information regarding its execution. The development of SimOS's accurate mode hardware component models is largely driven by the information needs of particular investigations, leading to the existence of several different implementations. This section describes examples of each hardware component implementation.

CPU Models

SimOS contains accurate mode implementations of two very different processor models. The first, called Mipsy, models a simple, single-issue pipeline similar to that found in the MIPS R4600. As depicted in Figure 4.6, Mipsy simulates basic CPU functionality and timing details using a straightforward fetch-decode-execute loop. Each cycle, Mipsy checks if any devices have requested processor interrupts, and then attempts to fetch a single instruction. Mipsy parses each instruction into its opcode and arguments and then applies the instructions intended effects to simulated processor registers. Instruction and data virtual addresses are translated into physical addresses using a simulated TLB, and the relevant exceptions are raised should a translation fail. To avoid the speed cost of modeling the complexities of modern processor pipelines, Mipsy charges a fixed latency for each instruction. As such, it is not an effective model for detailed processor investigations. However, it can provide less detailed, but still valuable information such as TLB activity, instruction counts, and, as described below, detailed uniprocessor and multiprocessor memory system behavior.

SimOS includes a second processor implementation called MXS. The version of MXS used in SimOS is based a user-level processor model described in [Bennett96], and is extended to support privileged instructions, MMU execution, and exceptions. MXS uses the same high-level loop-based execution as Mipsy, but models a much more complicated processor and with significantly more detail. MXS models a superscalar, dynamically-scheduled processor similar to the MIPS R10000 [MIPS95], complete with register renaming, branch prediction, speculative execution, and precise interrupts. MXS is highly configurable, allowing a user to specify instruction window size, execution unit configurations, branch prediction tables and schemes, and numerous other processor parameters. Furthermore, MXS collects detailed processor behavior statistics such as branch prediction success rate, register renaming effectiveness, and execution unit utilization. This information is particularly relevant to CPU, memory system, and compiler design where very low-level processor pipeline behavior has significant performance ramifications. This information does come at a cost though. As described in Section 4.4.4, the extra complexity and detail modeled by MXS results in simulation speed that is more than an order of magnitude slower than Mipsy.

To support multiprocessor simulation, both Mipsy and MXS encapsulate all hardware state into a single data structure and then replicate this data structure for each simulated CPU. As a result, each access to per-CPU hardware state requires an additional level of indirection. As illustrated in Figure 4.7, a modified main execution loop iterates through the list of hardware state pointers each cycle, allowing the simulated processors' notions

Main Loop:

```
while (TRUE) {
   if (P >= CPUState[NumCPUs]) {
      P = CPUState[0];
      cycle++
   } else {
      P = P++;
   }
   // Execute cycle for CPU "P"
   ...
   P->PC = P->PC + 4;
}
```

CPUState[NumCPUs]

```
Register Reg[NumRegs];
TLBEntry TLB[NumTLBEntries];
Register PC;
// MXS only
BranchInfo BPTable[NumBPEntries];
...
// Per-CPU processor statistics
Stat instructionCount;
Stat exceptionCount;
```

Figure 4.7. Modifications required for multiprocessor support.

of time to remain synchronized. This is a particularly important requirement for accurate modeling of cache and memory system implementations where the interleaving of memory references can have a significant impact on computer system behavior.

Cache models

Using the interfaces described in the previous chapter, Mipsy and MXS attach to a completely configurable cache model. One commonly used cache simulator models a blocking cache with separate first level instruction and data caches and a unified second level unified cache. Each level in the cache model has configurable capacity, miss penalty, associativity, and line size. Every instruction fetch and data access passes through this cache, and detailed statistics are collected regarding the caches' performance. Furthermore, each cache reference can cause the processor to stall for a configurable number of cycles to model the access latencies in modern cache hierarchies. This type of implementation is referred to as a blocking cache because execution is blocked until the cache miss is resolved.

MXS uses the same interface to attach to a non-blocking cache model. Non-blocking caches allow execution to continue even though a cache miss has occurred, and are an essential component of speculative execution. Like the blocking cache model, the non-blocking cache model is completely configurable and collects detailed behavioral statistics including the miss rate, the cause of cache misses, and line utilization. [Wilson96] describes an additional non-blocking cache implementation which provides the above utility, but provides additional accuracy and investigation capabilities through the modeling of cache port contention.

Memory system models

When a cache miss occurs, the memory reference passes through another common interface to reach the memory system simulation models. Once a reference is on the uniprocessor memory bus or multiprocessor interconnect, several different actions and widely-varying latencies can occur, and SimOS includes memory system simulators to model this activity. SimOS has been used in several multiprocessor investigations, leading to the development of several interchangeable memory system models. One memory system implementation models a split-transaction, out-of-order completion memory bus. In its multiprocessor configuration, cache coherence is provided by a cache snooping mechanism with an invalidation-based protocol. This model includes advanced memory system features such as cache-to-cache transfers and reference merging, and maintains detailed statistics regarding the behavior of these and other features.

Many recent shared-memory multiprocessor implementations locate a portion of the total system memory with each processing node. A side-effect of this configuration is that the latency to access memory on a remote processing node can be significantly higher than accessing local memory. This *non-uniform memory access* (or *NUMA*) can have a significant effect on the performance of multiprocessor workloads. To support the investigation of this class of machine as well as the workloads that they support, SimOS includes a memory system implementation that models a directory-based, cache-coherent NUMA memory system. Furthermore, this model is highly configurable and can provide detailed information regarding cache coherence protocol activity, interconnection network performance, queuing delays, and other important memory system behavior.

A final example of a SimOS accurate mode memory system implementation is a cycle-accurate model of the FLASH multiprocessor memory system called FLASHLite [Kuskin94]. FLASH is a MIPS R10000-based shared-memory multiprocessor designed to scale to thousands of processors. The combination of the MXS CPU model with FLASHLite allows SimOS to model the FLASH multiprocessor with significant accuracy, provides very detailed performance information, and, as described in Chapter 6, has enabled several important investigations.

I/O device models

The timing and implementation characteristics of some I/O devices can have a significant impact on the execution behavior of a workload. For example, database transaction processing workloads have significant disk activity, and the particular timing traits of the disk affect workload behavior. When a process requests a file access, the operating system forwards this request to the disk and then deschedules the process. Once the disk has satisfied the request, it interrupts the CPU to indicates its completion, and the operating system can subsequently reschedule the process to continue its execution. The time between the disk request and the processor interrupt varies according to the particular disk's implementation. To provide realistic disk request latencies, SimOS includes a validated simulator of an HP 97560 disk drive [Kotz94]. This model includes is highly configurable and models latency-determining characteristics such as disk head position and rotational delay.

Similarly, a web server workload has significant network activity, and SimOS includes an accurate mode ethernet chip and network model to support and measure this activity. As mentioned in the previous chapter, SimOS allows multiple simulated machines to coexist and communicate with each other through normal networking protocols. When providing this communication capability, the ethernet chip and network model imposes configurable peak bandwidth restrictions, DMA delays, and network transfer latencies. As a result, this model can provide realistic networking delays, improving the accuracy of workload behavioral information.

4.4.4 Performance

The previous sections describe SimOS's implementation of accurate, positioning, and rough characterization simulator execution modes. To better demonstrate the trade-off between speed and detail, this section examines the performance of each simulation mode as they model uniprocessor and multiprocessor computer systems.

Simulating uniprocessors

Table 4.1 compares the time required to run several workloads on a hardware-based uniprocessor machine to the time required to run the same workloads on each of SimOS's simulator execution modes. We run SimOS on a Silicon Graphics Indy workstation equipped with a 133 MHz R4600 CPU and 96 megabytes of memory. Each simulator execution mode is configured to match the Indy configuration as closely as possible while still exploiting the benefits of the mode. For example, the positioning mode has zero latency memory and disk access latencies while the Mipsy-based accurate mode has two levels of caches and uses the detailed disk model. The native execution numbers are the wall-clock time it takes to execute each workload directly on the Indy, while the simulation numbers are the time required to execute the same workloads on top of a SimOS-modeled machine. We divide the simulation wall-clock time by the native execution time to compute the slowdown.

Table 4.1. SimOS performance when modeling a uniprocessor

	Native Execution	Positioning Mode	Rough Char. Mode	Accurate Mode (Mipsy)	Accurate Mode (MXS)
Workload	Wall-clock	Wall-clock (Slowdown)	Wall-clock (Slowdown)	Wall-clock (Slowdown)	Wall-clock (Slowdown)
056.ear	7 sec.	2:57 min. (25x)	5:21 min. (46x)	43:28 min. (326x)	294:14 min. (2,522x)
026.compress	7 sec.	2:25 min. (21x)	5:45 min. (49x)	24:43 min. (212x)	211:26 min. (1,812x)
Java	13 sec.	4:20 min. (20x)	9:44 min. (45x)	47:50 min. (221x)	697:40 min (3,220x)
Compilation	19 sec.	8:16 min. (26x)	11:54 min. (38x)	110:53 min. (350x)	1342:05 min. (4,237x)

The workloads used in the performance comparison are:

- SPEC benchmarks - The 056.ear and 026.compress programs are taken from the SPEC benchmark suite [SPEC97]. While these applications do not effectively demonstrate the capabilities of complete machine simulation, they have been widely studied and provide a reference point for comparing SimOS's performance to other simulators.

- Java - This workload is taken from a java benchmark measured in [Romer96] and consists of the compilation and execution of DES encryption code. The workload converts the DES Java source code into bytecode format and then executes it with the aid of a just-in-time compiler.

- Compilation - This workload is taken from the compile stage of the Modified Andrew Benchmark [Ousterhout90] and demonstrates SimOS's ability to investigate more operating system-intensive workloads. The workload consists of a `gcc` compilation of 17 files followed by the creation of a library archive containing the object files.

The trade-off between speed and detail is obvious. Embra's high-speed binary translation techniques allows positioning mode to execute a workload only 20 to 30 times slower than the native hardware[1]. The rough characterization mode is also quite fast, generating basic cache behavior information at only 40 to 50 times slowdown. The Mipsy-based accurate simulator execution mode causes several hundred times slowdown, but is able to collect quite detailed processor, memory system, and I/O device behavior. Note that the widely differing slowdowns between the different workloads are due to the different hardware simulation requirements of each workload. For example, `056.ear` is a floating point-intensive application, and it takes Mipsy much longer to simulate floating point instructions than to simulate integer instructions. Similarly, applications with higher cache miss rates require more simulation time to model this cache behavior. Finally, the MXS-based accurate simulator execution mode results in several thousand times slowdown. Note that in this example, MXS models a more complex processor than the Indy's MIPS R4600.

The large slowdown resulting from the use of the most detailed MXS-based simulation can be quite prohibitive, even when applied to only the most interesting portions of a workload. To help provide detailed simulation across extended workload execution,

1. This is substantially slower than the performance reported in [Witchel96] and is due to several source code modifications designed to make Embra more maintainable.

SimOS extends the notion of dynamically selectable simulator execution modes. SimOS supports a sampling capability that automatically switches between different levels of simulation detail at user-specified intervals. Sampling enables the use of statistical analysis to estimate the behavior of the most detailed models during the execution of the workload. For example, two recent architectural investigations used SimOS's sampling support to simulate ten out of every hundred workload cycles in MXS, running the remainder in Mipsy [Nayfeh96] [Wilson96]. The resulting information estimates the behavior of the workload on a dynamically-scheduled processor, but at a fraction of the cost of total MXS-based simulation.

Simulating multiprocessors

Table 4.2 presents performance numbers for the simulation of multiprocessor machines. For these simulations, we run the workloads and SimOS on a Silicon Graphics Challenge multiprocessor equipped with four 150 MHz R4400 CPU's and 256 megabytes of memory. The slowdown numbers again compare the execution of the workload directly on the hardware to the same execution on the SimOS-modeled machine.

Table 4.2. SimOS performance when modeling a multiprocessor

	Native Execution	Positioning Mode	Rough Char. Mode	Accurate Mode (Mipsy)	Accurate Mode (MXS)
Workload	Wall-clock	Wall-clock (Slowdown)	Wall-clock (Slowdown)	Wall-clock (Slowdown)	Wall-clock (Slowdown)
Raytrace	5 sec.	5:50 min. (70x)	20:53 min. (246x)	153:36 min. (1,841x)	42:22:05 (30,505x)
Database	7 sec.	11:20 min. (97x)	42:03 min. (360X)	295:04 min. (2,528x)	68:15:00 (35,100x)

The workloads used in this performance comparison are:

- Raytrace - This workload is taken from the SPLASH suite of applications and is a parallel implementation of a widely used graphics rendering algorithm. The workload is compiled as suggested in the distribution and executed as:

```
raytrace -m40 -p4 inputs/teapot.env
```

- Database - This workload consists of a parallelized Informix database server supporting a transaction processing workload similar to TPC-B [Gray93]. The workload contains four processes that make up the parallel database server plus 10 client programs that repeatedly submit transactions to the database.

The trade-off between speed and detail becomes even more pronounced for multiprocessor simulation. In each simulation mode, SimOS models all four processors within a single host platform process, causing simulation slowdown to scale linearly with the number of CPU's being modeled. The accurate mode slowdowns when supporting the database workload are larger than when supporting the raytrace workload due to the overhead of simulating many more cache misses and modeling significantly more disk activity. In either case, the inability to simulate large number of processors in a timely manner is a significant problem, and limits SimOS's applicability to relatively small machine sizes. This important limitation is revisited in Chapter 6.

4.5 Summary

The challenge that has most limited the use of complete machine simulation is the time it takes to generate detailed statistics. Complete machine simulation addresses this challenge by including multiple implementations of each hardware component, each making an trade-off between the detail that it provides and speed at which it provides this detail. The SimOS implementation of complete machine simulation demonstrates that it is possible to implement three important simulator execution modes that provide a wide variety of simulation speed and detail characteristics. Furthermore, complete machine simulation provides investigators with explicit control over the use of these simulator execution modes, allowing them to select the simulation characteristics most appropriate for the changing needs of a workload investigation.

Chapter 5
Efficient Management of Low-Level Simulation Data

The previous chapter describes how dynamically adjustable speed and detail characteristics address complete machine simulation's performance challenge. However, the ultimate goal of complete machine simulation is to help an investigator understand some aspect of a computer system's behavior. This chapter describes how effective simulation data management can provide the information necessary to achieve this understanding. The first part of this chapter describes the data management challenges facing complete machine simulation and how investigation-specific data management addresses these challenges. The second part of this chapter describes how SimOS implements investigation-specific data management. SimOS separates the data management process into event generation and event processing stages and provides efficient mechanisms for flexibly controlling all event classification and reporting.

5.1 Data management challenges

Complete machine simulation provides an opportunity to obtain very detailed computer system behavioral information. Complete machine simulation's hardware models can be heavily instrumented to observe their own behavior, and can thus report all of the low-level hardware activity that occurs during a workload's execution. For example, a detailed CPU model such as MXS can count mispredicted branches, pipeline stalls, or TLB misses. Similarly, cache models can calculate miss rates and the amount of stall time that these cache misses cause during a workload's execution. In fact, the combined activity of complete machine simulation's hardware models completely defines the computer's execution and is thus the source of all behavioral information. However, when studying a complex system, transforming this hardware data into useful behavioral information presents two problems. First, a complete machine simulator's hardware models generate very low-level data that is not particularly useful in its raw form. Second, this low-level data is generated at an extremely high rate, and attempts to organize it into more meaningful information can have a significant impact on simulation speed.

Low-level data

Hardware models produce data that is often at too low of a level for many investigations. This problem arises because the hardware of a computer system works at a different level of abstraction than most software. For example, most hardware caches deal with physical addresses, and a cache model can easily count the number of misses that occur to different ranges of the physical address space. However, applications work with code and data defined by virtual addresses and are unaware of any physical address translation. To be useful to an application programmer, the physical address-based cache miss data must be transformed into virtual address-based data and mapped to the specific application data structures responsible for these cache misses. Similarly, the hardware of a computer system has no notion of operating system abstractions such as processes and their scheduling. Even if the cache model could transform miss data into virtual address-based information, it must be further classified by process ID number to distinguish the behavior of different applications in a multiprogrammed workload.

Even in the evaluation of low-level architectural designs, unorganized data can cause problems. For example, many workloads spend a substantial portion of their execution time in an "idle" mode waiting for I/O requests to be satisfied. This idle mode is typically implemented as a very tight loop where the operating system repeatedly checks for processes that are ready to run, and has excellent pipeline and cache behavior. However, a high-performance idle loop is not particularly valuable and its unrepresentative behavior can obscure the true value of a new hardware design. A workload's cache miss rate or average cycles per instruction (CPI) will be a more useful metric if the hardware activity occurring during the operating system's idle loop is filtered out. Just as with application data structures and operating system processes, the hardware models have no concept of an idle mode, making this process challenging.

High-speed data generation

The challenge of organizing low-level hardware data into more meaningful computer system behavioral information is compounded by the importance of simulation speed. Accurate mode hardware simulators observe tremendous detail during the course of workload execution, and can potentially produce volumes of data every second. Unless the organization and classification of this data is highly efficient, the performance of the simulation will suffer. As discussed in the previous chapter, speed is essential to the effectiveness of complete machine simulation, and any approach to data management must therefore minimize its performance impact.

Complete machine simulation's data management challenge is thus to organize low-level hardware data into more meaningful computer system behavioral information, and to perform this organization as efficiently as possible.

5.2 The solution: Investigation-specific data management

Complete machine simulation addresses these challenges by allowing users to customize all data collection and reporting to meet the specific needs of their investigations. Specifically, complete machine simulation allows a user to incorporate knowledge of the workload under investigation and to specify exactly what information about this workload's execution is desired.

To organize low-level hardware data into more meaningful computer system behavioral information, complete machine simulation allows an investigator to provide higher-level knowledge of a workload's composition. As an example, a CPU model does not have any concept of operating system processes, but an investigator knows that new processes are created by certain system calls and are scheduled by specific operating system procedures. An investigator can inform the complete machine simulator of these process-related events, allowing the CPU model to attribute pipeline stalls according to the processes that cause them. Similarly, a cache model has no concept of an application's data structures, but an investigator knows the virtual address range where important or interesting structures reside. An investigator can inform the complete machine simulator of important address ranges, allowing the cache model to count the data cache misses that occur to these specific data structures. In either case, the addition of higher-level workload knowledge customizes the data management process to a particular workload to obtain more meaningful behavioral information.

To provide high-performance data management, complete machine simulation allows an investigator to specify exactly what behavioral information their investigation requires. For example, a hardware designer might specify that they are only interested in evaluating the behavior of a new cache configuration. The complete machine simulator can avoid much of the overhead of classifying and reporting other hardware data such as processor pipeline or I/O device activity. Similarly, an investigator can specify that they are only interested in the behavior of a single process in a multiprogrammed workload, allowing the complete machine simulator to avoid the overhead of detailed data classification throughout the workload's entire execution. Just as user-selectable speed and detail characteristics allows complete machine simulation to minimize hardware simulation time, specification of an investigation's information needs allows it to minimize data management time.

5.3 SimOS's implementation

The combination of two important implementation features allow the SimOS implementation of complete machine simulation to support the customization of data management to the specific needs of an investigation. First, SimOS separates the data management process into separate event generation and processing stages, allowing users to customize the classification and reporting of data without modifying the hardware simulators themselves. Second, SimOS provides mechanisms for controlling the event processing stage, enabling the incorporation of workload-specific knowledge and the implementation of efficient classification and reporting functionality. This section describes these implementation features and how they support investigation-specific data management.

The data management process of any simulation tool used for computer system behavioral investigation requires transforming simulator data into some form of useful information for reporting. This transformation might be as simple as converting memory reference counts into a cache miss rate or as complex as assigning pipeline stalls to the application basic blocks that cause them. In either case, the transformation process can be reduced to hardware event generation and hardware event processing stages. In the *event generation* stage, the simulation tool generates events corresponding to the activity that occurs during a workload's execution. These events might include the execution of instructions, MMU exceptions, cache misses, I/O device interrupts, or any other activity that the simulator observes. In the *event processing* stage, these events are filtered and classified into higher-level information regarding the computer system's behavior and reported to the user.

The simulation data management process is not always implemented as two separate stages. Many simulation tools combine the event generation and processing stages into a single step. For example, application transformation tools such as ATOM [Eustace95], EEL [Larus95], and Etch [Romer97] provide investigation-specific behavioral information by combining simulation, event processing, and reporting into a single executable program. In order to examine a new application or a different aspect of an application's execution behavior, the simulation tool is recompiled to incorporate the new investigation-specific data management chores. While this is may be an acceptable approach for some tools, it does not work as well for complete machine simulation. The initial version of SimOS combined the generation of hardware events with investigation-specific event processing. For example, information about the IRIX idle loop and important application procedures was compiled directly into SimOS's hardware models, allowing them to collect and present their hardware statistics according to these higher-level concepts. To obtain different types of behavioral information or investigate new operating systems and applications, an investigator would sprinkle code throughout each of the hardware model

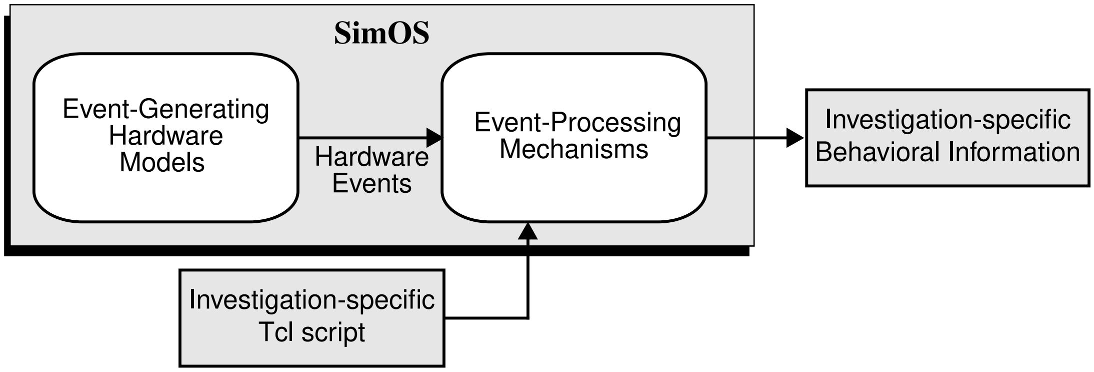

Figure 5.1. The complete machine simulation data management process

implementations and recompile the simulator. However, modification of complete machine simulation's hardware models required significant knowledge, time, and effort, and ended up dissuading many users from substantial customization of the data management process. Furthermore, investigation-specific data management code was often added, but rarely deleted from the hardware models. As a result, significant event processing was always active, providing information that was never used.

To address these problems, subsequent versions of SimOS completely dissociate investigation-specific event processing from the hardware models. As illustrated in Figure 5.1, SimOS's hardware models are responsible solely for generating hardware events, and remain completely free of any investigation-specific information. As a result, they do not need to be modified or even recompiled to support an investigation of a new workload or satisfy different data processing requirements. The hardware events generated by the hardware models provide input to SimOS's event-processing stage where they are filtered, classified, and transformed into investigation-specific behavioral information.

The second form of input is a investigation-specific Tcl script that specifies and controls the event-processing stage. We selected the Tcl scripting language as the event-processing control language because it provides a simple, consistent syntax and is easy to integrate with compiled languages [Ousterhout94]. This Tcl script utilizes several SimOS-provided mechanisms to incorporate higher-level workload knowledge into the data management process and to specify what behavioral information should be collected and reported. This allows SimOS's data management process to be easily customized to satisfy the specific information needs of an investigation. The next section describes the SimOS event-processing mechanisms in detail and demonstrates how they efficiently provide customized computer system behavioral information.

5.4 Event-processing mechanisms

Given the design decision of dissociated hardware event generation and processing, the only run-time input to the SimOS's event-processing mechanisms are hardware events. The goal of each of the mechanisms is to efficiently process these low-level hardware events into some form of higher-level information beneficial to an investigation. SimOS's event-processing mechanisms use hardware events in two capacities. First, hardware events are the fundamental unit of performance, and SimOS's event-processing mechanisms count and classify these events to generate information. For example, a count of mispredicted branches that occur during a workload's execution can help an architectural investigator determine the effectiveness of a branch prediction scheme. Similarly, a count of data cache misses can help an investigator understand the basic data reference locality that exists in a workload. Second, hardware events act as "hooks" for allowing an investigator to incorporate workload-specific knowledge into the classification of hardware events. For example, an investigator could indicate that virtual address 0x80004000 is the entry point to the operating system's idle loop. A hardware event indicating the execution of the instruction at this virtual address could trigger SimOS to classify future cache miss events as occurring in idle mode.

In addition to providing event counting and classification functionality, SimOS's event-processing mechanisms must be easy to use. If the investigation-specific Tcl scripts are difficult to write, investigators will be discouraged from realizing the most beneficial behavioral information. Additionally, each event-processing mechanism must be as efficient as possible. Even when satisfying just the specific information needs of an investigation, inefficient event processing can have a significant impact on complete machine simulation's performance.

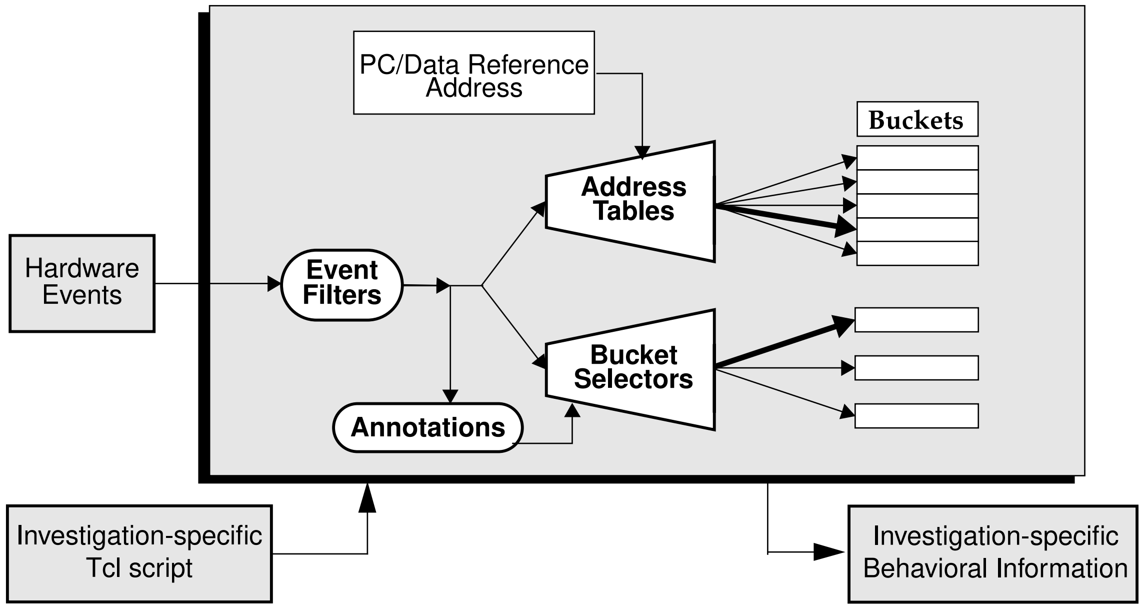

Figure 5.2. Overview of SimOS event-processing mechanisms

Figure 5.2 provides an overview of the SimOS event-processing architecture. The rest of this section describes each of the bold-faced mechanisms and how they provide different event-processing functionality while remaining easy to use and highly efficient.

5.4.1 Annotations

Annotations are one of SimOS's most heavily used mechanisms. Annotations are Tcl scripts that an investigator attaches to the occurrence of specific events. Whenever the specified event occurs, the annotation script is interpreted to provide the specified event-processing or simulation control functionality. The general format of annotation specification is:

```
annotation set <annotation-triggering event> {
   <annotation script>
}
```

To provide the greatest flexibility, SimOS allows annotations to be triggered with any type of hardware event. Among the most heavily used triggering events are:

- **Reaching a particular program counter address.** *Program counter annotations* are among the most commonly used and are invoked whenever a processor executes an instruction at a specified virtual address. These annotations can be used to indicate that the CPU has reached the entrance to or exit from interesting sections of operating system and application code.

- **Referencing a particular data address.** *Data reference annotations* are invoked whenever a specific virtual address is read or written. Data reference annotations can provide a watchpoint-like functionality that is particularly useful in operating system debugging. If a particular data structure is found to be corrupted, a data reference annotation could determine when, where, and why the data structure was improperly written.
- **Occurrence of an exception or interrupt.** *Trap-based annotations* may be set to trigger whenever an exception or external interrupt occurs in the target machine. These annotations can be set on specific types of traps such as system call exceptions, TLB misses, or disk interrupts, and are useful for tracking transitions from user to kernel processor modes.
- **Execution of a particular opcode**. *Opcode annotations* may be set for particular instruction types or classes of opcodes. For example, in the MIPS architecture, an `rfe` (return from exception) or `eret` (exception return) instruction is used to return the processor to user mode after taking an exception. Annotations triggered by these instructions could be used to track transitions from the processor's kernel mode back to user mode.
- **Reaching a particular cycle count.** *Cycle annotations* are triggered when the target platform has executed for a specified number of cycles. Cycle annotations are particularly useful for repeated event processing activity such as sampling of machine state or periodic statistical output.

There are of course many other hardware events that may be desirable for triggering annotations, and their utility depends on the needs of a particular investigation. For example, research into ethernet behavior may benefit from such annotation-triggering events as packet arrival or network collisions.

When an annotation-triggering event occurs, the associated annotation script is interpreted to provide some specific utility. In their most basic capacity, annotation scripts can be used to count the occurrence of particular events. For example, the following trap-based annotation is counts the number of interrupts that occur during a workload's execution:

```
annotation set exception INTERRUPT {
    incr interruptCount
}
```

Similarly, a data reference annotation assigned to an application data structure could count the number of times that this structure is read or written.

Simple counts of annotated hardware events provide valuable information, but annotation scripts can provide much greater capabilities. We extend the capabilities of annotation scripts by giving them access to the entire state of any hardware models in the simulated machine. This state includes the processor registers and caches, TLB entries, I/O devices, processor caches, and main memory. Annotation scripts can exploit this access to better understand the state of the target machine. For example, by reading data structures in the target machine's operating system, an annotation can discover the ID of the currently running process. This information can be subsequently used to classify hardware events according to the process that causes them.

The previous examples describe annotation scripts in a passive capacity, only being used to count events or query the state of the target machine. However, annotation scripts are valuable in an active role as well. Tcl makes it simple to attach script commands to internal simulator procedures to control aspects of the simulation environment. For example, users write annotations to switch between the different SimOS simulator execution modes. To switch from the Embra CPU model to the MXS CPU model, an investigator includes the Tcl command `cpuEnter MXS` in an annotation script. As a result, it is easy to switch between modes at a particular machine cycle, upon entry to a particular application procedure, or upon access to a specified data structure. As described in the next section, active annotation scripts are also used to control more advanced event-processing mechanisms.

An important goal of the event-processing mechanisms is their ease of use, and SimOS attempts to simplify the specification of annotation-triggering events and annotation scripts. For example, annotations support the symbolic specification of all memory addresses. The embedded Tcl interpreter includes knowledge of object file symbol table composition, allowing program counter and data reference events to be specified at a higher level of abstraction. For example, to trigger a program counter annotation at the start of a sorting application's main procedure, an investigator can specify:

```
annotation set pc SortingApp::main:START
```

rather than the less meaningful:

```
annotation set pc 0x00412de0
```

Annotation scripts also have access to symbol table information and all references to memory locations can be made symbolically. As a result, Tcl-based references to data structures are as simple as interactions with a source-level debugger. The following example demonstrates how symbolic specification makes the data collection process as simple as possible. In this example, an investigator wants to know the final value of the `position` field in the `inputData` data structure whenever a particular procedure executes:

```
annotation set pc SortingApp::SortElement:END {
  console "The position is
    [symbol read SortingApp:inputData.position]\n"
}
```

Each time the `SortElement` procedure finishes, the `console` command will output the desired value for the investigator to view. The `symbol` command tells the Tcl interpreter to parse the specified symbol table to determine the appropriate address to read. As future examples will demonstrate, symbolic reference to the contents of main memory allow annotation scripts to be extremely powerful. Not only does symbolic reference ease the burden of annotation specification, it also increases their portability. When an application or operating system is recompiled, the addresses of its text and data sections change. Symbolically-specified annotation events continue to follow their semantic intent while address-specified annotation events would require rewriting.

Because annotations are such heavily used mechanisms, their implementation must be as efficient as possible. SimOS implements annotations by incorporating simple triggering code into each of its hardware models. The interface between the hardware models and the annotation handler is simply:

```
ExecuteAnnotations(<EventType>, <value>);
```

For example, each SimOS CPU model must invoke:

```
ExecuteAnnotations(TrapType, INTERRUPT)
```

each time a device interrupt occurs to invoke any trap-based annotations assigned to interrupts. Note that the hardware models simply trigger annotations and have no knowledge of the annotation scripts themselves. The `ExecuteAnnotations` function is highly optimized, using hashing techniques to handle the common case, events that trigger no annotations, as quickly as possible. With the goal of providing the highest possible simulation speed, the Embra CPU model uses additional techniques to provide efficient annotation support. If Embra were forced to call the `ExecuteAnnotations` function for every single instruction, its performance would suffer. Consequently, Embra queries the annotation subsystem before the translation of each workload basic block to see if any of the basic block's program counter values should trigger annotations. If so, the annotation is invoked directly from the translated code. However they are implemented, the overhead of using annotations is directly proportional to the number that are installed and the complexity of their scripts. As such, a user must only pay the annotation performance cost required to satisfy their investigation's specific information needs.

5.4.2 Bucket selectors

As described above, hardware events act as the fundamental unit of machine performance as well as the hooks for incorporating workload-specific information into the event-processing stage. Counting hardware events such as executed instructions, cache hits, or TLB misses, a complete machine simulator can provide an investigator with a better understanding of their workload's execution. However, event counts aggregated over the entire workload's execution are not as useful as they could be. Additional utility could be provided by maintaining multiple counters of each hardware event type where each counter corresponds to some meaningful portion of a workload. For example, an application writer might desire a count of the number of instructions and cache misses that occur during each of an application's procedures. These counts could be used as profile information, focussing performance tuning on the most time-critical procedures in the application. A large number of cache misses in a procedure would indicate that its lack of data or instruction cache locality needs attention.

Similarly, an architectural investigation might want to track the number of missed branch predictions to determine the effectiveness of a branch prediction scheme. These counts may be more meaningful if they are categorized as occurring in either the idle loop or during normal operating system and application execution. It is possible to use the annotation mechanism to count and classify all hardware events. However, it would require setting annotations on every single event and determining how to categorize each of these events within the annotation script. Not only would this require an investigator to write an extremely large number of annotations, the run-time performance of the simulation environment would suffer. SimOS provides a mechanism called *bucket selectors* to make this common event counting and categorization both easy to use and highly efficient. The goal of a bucket selector is to decompose a workload's total execution time into smaller, more meaningful components and then to customize all data collection to these components.

Bucket selectors requires two types of user input. First, the investigator chooses which events are of interest by specifying the contents of a *bucket*. A bucket is simply a user-specified collection of events that should be counted. For example, an investigator might specify that a bucket should include counts of instruction execution, TLB misses, and cache misses. Whenever one of these hardware events occurs, the corresponding bucket counter is incremented. While every possible hardware event count could be maintained in each bucket, the ability to select particular counts of interest reduces the performance impact to just what is required for an investigation's information needs.

The second phase of bucket selector creation is the specification of execution phases or components. These components might be application procedures, user-level processes, or even more abstract concepts such as individual database transaction or web server requests. Buckets are assigned to these execution components, and at any given time, only a single bucket is active. A *selector* is simply an indication of which bucket is currently

```
# Save current state on stack when an exception occurs
annotation set exception {
  stackPush "stateStack" $currentState
  set currentState kernel
  selector set modes kernel
}

# Restore saved state when the processor executes
# the "return from exception" opcode
annotation set inst rfe {
  set currentState [stackPop "stateStack"]
  selector set modes $currentState
}

# Transition to the idle mode when the PC hits the idle loop
annotation set pc kernel:idle:START {
  set currentState idle
  selector set modes idle
}

# Return to the kernel mode at the end of the loop
annotation set pc kernel:idle:END {
  set currentState kernel
  selector set modes kernel
}
```

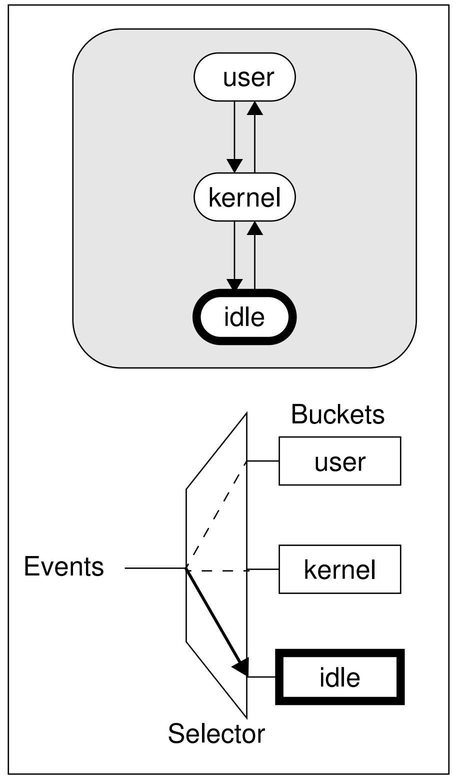

Figure 5.3. Processor mode bucket selector

active. Annotations control selectors, and once an annotation has set the selector to point to a particular bucket, all specified hardware events will be automatically funneled into that bucket until the selector is changed again, signaling the entry into a new phase of execution.

Figure 5.3 illustrates typical usage of the bucket selector mechanism. In this example, a workload's execution time is decomposed into processing user-level code, processing kernel-level code, and spinning in the kernel idle loop. The left side of the figure shows the Tcl source used to implement the processor tracking functionality. The script implements the state machine illustrated in the top right portion of the figure. The state machine controls the setting of a selector, as depicted in the bottom right side of the figure. For readability, the code has been simplified to omit bucket and selector initialization. The script places annotations on exceptions, on the return-from-exception opcode, and at the start and end of the operating system idle procedure. These annotations set the "modes" selector in order to direct event counts into the bucket corresponding to the current mode.

To provide additional utility, SimOS allows multiple selectors to coexist with each funneling events to its own set of buckets. As a result, several execution decompositions can be generated simultaneously and provide different high-level views of the same workload execution. For example, an investigation into a database transaction processing may benefit by categorizing events based on the active database server procedure while simultaneously categorizing events based on the particular transaction being processed.

The bucket selector mechanism has proven to be an extremely effective component of SimOS's data management process. In addition to encouraging investigation-specific categorization of hardware events, the bucket selector mechanism is simple to use and lends itself to an efficient implementation. SimOS implements bucket selector with a single level of indirection in the normal event counting code of each hardware model. For example, a simulated cache model typically counts the number of misses that occur with code similar to:

```
if (InCache(reference) == FALSE) cacheMisses++
```

SimOS adds a level of indirection to this cache miss counting code:

```
if (InCache(reference) == FALSE) (*cacheMisses)++;
```

The bucket selector has complete control over the setting of this pointer, allowing it to funnel cache miss counts to the appropriate bucket. The impact of this level of indirection is relatively minor, requiring a single additional memory reference for the incrementing of each hardware event counter. Furthermore, annotations are only required to change the currently active bucket, making their overhead directly proportional to the granularity at which an investigator decomposes a workload.

5.4.3 Address tables

Automatic hardware event categorization is an effective technique for collecting performance data, but the bucket selector mechanism is not always the best approach. While bucket selectors are good at assigning events to higher-level execution abstractions, more precise data is often desirable. Specifically, it is often informative to categorize events based on the individual line of code or data structure responsible for its occurrence. Categorizing hardware events at this fine granularity would require a tremendous number of annotations to properly control bucket selection. SimOS provides an *address table* mechanism to address this information-gathering challenge.

An address table is designed to efficiently attribute hardware events to the particular instructions or data references responsible for their occurrence. Categorizing data cache miss events by the referenced address can show exactly which data structures are exhibiting poor cache locality. Similarly, categorizing pipeline stalls by program counter address shows which portion of code may benefit from better compiler instruction scheduling. Address tables are a special case of bucket selectors where the active bucket is determined by the address of the event rather than by explicit user interaction. Address table specification consists of two phases. As with bucket selectors, the first phase requires the specification of hardware events that should be counted. An address table can consist of many thousands of buckets, and tracking only the hardware events of interest can greatly reduce the mechanism's memory requirements.

The second phase of specification requires a declaration of the address ranges to be tracked and the granularity at which these ranges should be decomposed. Granularities range in size from individual memory words up to entire pages; each unit of data collection is assigned a bucket. Furthermore, address tables can be either *code-driven* or *data-driven*. Code-driven address tables categorize hardware events based on the current program counter address and are used to understand the behavioral characteristics of an application's source code. Data-driven address tables track hardware events only during load and store instructions and categorize the events according to the data address that is read or written.

Figure 5.4 illustrates how a code-driven address table can help determine the behavioral characteristics of a workload. The goal of this simple example is to determine the particular lines of code that are exhibiting performance problems. Specifically, the script creates an code-driven address table to count the number of TLB and instruction cache misses that occur for each 128-byte long cache line in the sorting application's text address space (0x00400000 through 0x00410000) and for each 4-kilobyte page of the dynamically loaded "C" library's text address space (0x60000000 through 0x600020000).

```
# Create address tables for the sorting application and the
# "C" library. The last parameter is the data collection unit size.
addressTable code "sortCode" 0x00400000 0x0041000 0x80
addressTable data "sortData" 0x10000000 0x10004000 0x80
addressTable code "libcCode" 0x60000000 0x60002000 0x1000

# Specify which hardware events the address tables should count
addressTable bucket "sortCode" {instCacheMisses dataCacheMisses tlbMisses}
addressTable bucket "sortData" {dataCacheMisses}
addressTable bucket "libc" {instCacheMissEvents tlbMissEvents}
```

Data-collection unit	Symbolic name	Icache misses	TLB misses	Dcache misses
0x00400000-0x00400080	main(): line 21	56	53	10
0x00400080-0x00400100	main(): line 23	122	67	88
. . .	. . .	. . .	. . .	. . .
0x10000000-0x10000080	sortStruct.value	Not kept	Not kept	356
. . .	. . .	. . .	. . .	. . .
0x60000000-0x60001000	printf(): lines 22-234	223	10	Not kept
. . .	. . .	. . .	. . .	. . .

Figure 5.4. Code- and data-driven address tables

It also creates a data-driven address table to count the data cache misses that occur in the sorting application's data segment (0x10000000 through 0x10004000).

During the execution of this workload, the selected hardware events are categorized into the specified address ranges, resulting in the tabular event count data. Using the object files' symbol table information, these data collection units are mapped back to concepts understood by the user. For code-driven address tables, event counts are mapped to the line or lines of source code that led to the events, while for data-driven address tables, event counts are mapped to the symbolic name of the responsible data structures. The end result is an accurate determination of the causes of system behavior presented at a level of abstraction that is useful to the investigator. The combination of code-driven and data-driven address table information provides more useful information than either one in isolation. In the above example, data cache miss events are counted simultaneously in both the code-driven and data-driven address tables. These orthogonal views can be crossreferenced to determine what source code is responsible for misses to specific data structures.

Address tables classify a tremendous quantity of low-level hardware events, and an efficient implementation is essential. Like the bucket-selector mechanisms, address tables exploit the level of indirection present in the hardware models' event counts. However, the setting of the pointer is automatically determined by the current PC or data reference address rather than by an annotation script. To determine the correct bucket as quickly as possible, each address table maintains a hash table indexed by PC or data reference address and optimized to the user-selected bucket granularity sizes.

5.4.4 Event filters

The previous sections describe how annotations, bucket selectors, and address tables can efficiently attribute hardware events to components of a workload. However, these hardware events are often at too low of a level to make simple counts particularly useful. An example of this is the occurrence of cache miss events on shared-memory multiprocessors. Knowing that some piece of code or portion of a data structure suffers cache misses does not necessarily tell the programmer if and how these cache misses can be avoided. In order to eliminate these cache misses, it is helpful to know what *type* of cache misses occurred. It is often useful to raise the abstraction level of hardware events, and *event filters* address this need.

An event filter is implemented as a state machine that takes hardware events as input, builds up additional knowledge about these events, and outputs new higher-level events. Event filters are attached directly to the hardware event stream and use state machines to convert the original events into new, more descriptive events. The new events can be used to trigger annotations or can be counted and categorized using bucket selectors and address tables. Figure 5.5 illustrates how an event filter is used to classify the cache misses that occur in a shared-memory multiprocessor. In this example, a cache miss can be either classified as a "cold" miss, an "invalidation" miss, or a "replacement" miss [Dubois93]. To provide this classification, SimOS instantiates a state machine for each cache line-sized

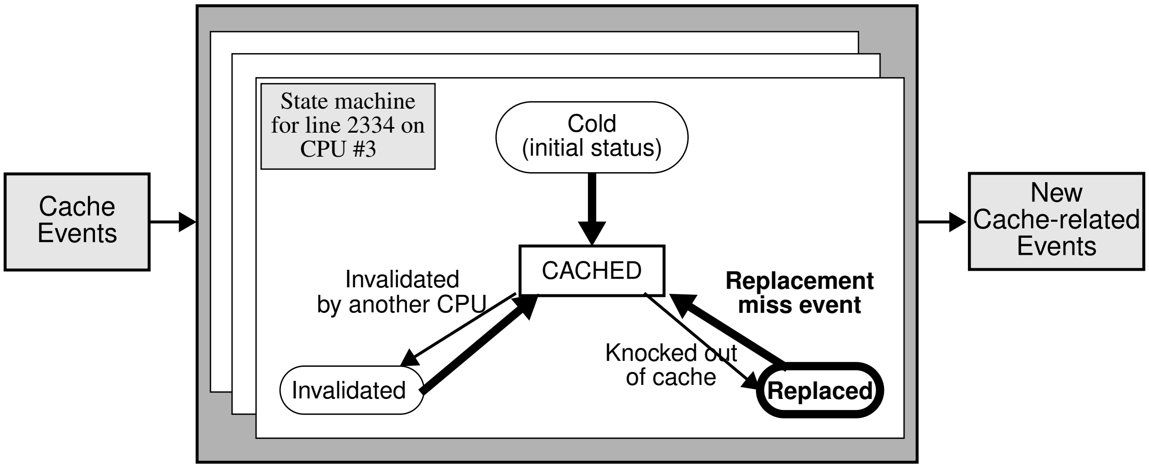

Figure 5.5. Cache miss event filter

portion of main memory and for each processor. At any point in time, the specific portion of main memory is either in a processor's cache or not. In the pictured state machine, bold lines represent cache misses and the normal lines represent transitions that occur either when the line is knocked out of the processor's cache or when the cache line is invalidated in the maintenance of cache coherence. In this example, the referenced cache line is in the "replaced" state, and so this cache miss will be classified as a "replacement miss event" and output for further processing.

The cache event filter is typically used in conjunction with address tables to associate different types of cache misses with particular pieces of code or data structures, information that is valuable to application performance tuning. For example, cold misses are usually unavoidable, whereas replacement misses can often be eliminated by restructuring data structures. Additionally, excessive invalidation misses may indicate that better control of the inter-processor communication is needed. Armed with this information, developers can reduce the number of application cache misses and substantially improve the performance of a workload.

Event filters are useful for improving the level of abstraction provided by other hardware events as well. For example, a state machine could track the state of a CPU's execution units, creating new events that indicate why pipeline stalls occur. In whatever architectural domain they are used, event filters are a simple technique for improving the information content of hardware-level events. As with the other mechanisms, the overhead of event filters is proportional to their use and complexity, allowing an investigator to pay only for the desired level of information collection.

5.5 Building higher-level mechanisms

The previous section described a number of mechanisms providing relatively disparate event-processing functionality. In practice, these mechanisms are often more valuable when used cooperatively. This section describes how the core event-processing mechanisms can be combined to provide more advanced data management functionality.

5.5.1 Annotation layering

Software layering is an essential tool in the creation and maintenance of large software systems. By creating abstractions and interfaces upon which higher levels of software can rely, implementation details can be hidden from the programmer, easing the burden of software creation. Similarly, a thorough understanding of a computer platform can require extensive use of annotations, and software layering is an essential component of this usage. Previous examples demonstrate annotations triggered by hardware events. However, to support software layering, it must be possible create new annotation-triggering events that have a higher-level meaning than individual hardware events provide.

The process of creating new annotation-triggering events is straightforward. At the lowest layer are annotation scripts triggered by hardware events. These scripts can access some component of the target platform's state to determine if it is desirable to generate the new user-defined event. When this new event type occurs, higher-level annotation scripts triggered by this new type will be interpreted. These scripts can perform some form of data collection or could generate even higher-level annotation-triggering events.

Figure 5.6 illustrates a typical use of user-defined annotation-triggering events. A group of program counter annotations are set throughout the operating system's process context-switching code to track the currently scheduled process. These annotations are set on the process management system calls, in the context-switching code, and at the beginning of

```
# Define a new annotation type for process-related events
annotation type process {switchOut switchIn}

# Program Counter (PC) annotation at the end of the
# exec system call (the PID doesn't change)
annotation set pc kernel:exece:END {
    # On an exec, the name of the process changes
    set PROCESS [symbol read kernel:u.u_comm]
}

# PC annotation at the end of the context-switching code
annotation set pc kernel:resume:END {
    # Execute the higher-level event
    annotation exec process switchOut
    # Update executable name and pid
    set PID [symbol read kernel:u.u_procp->pid]
    set PROCESS [symbol read kernel:u.u_ucomm]
    # Execute the higher-level event
    annotation exec process switchIn
}

# Annotation at the beginning of the idle loop
annotation set pc kernel:idle:START {
    # Execute the higher-level event
    annotation exec process switchOut
    set PID -1
    set PROCESS "Idle"
    # Execute the higher-level event
    annotation exec process switchIn
}
```

Figure 5.6. Creation of process-related events and data

the kernel idle loop. Tcl variables maintain the current process ID (PID) and process name. In this example, `u` is a variable in the operating system that gives access to the process table entry of the current process. This set of process-tracking annotations is packaged as a library. While the library doesn't directly generate any performance data, it is used by higher-level annotations to attribute events to specific processes. New annotations can rely on the new *process switchIn* and *process switchOut* events and can use the `PID` and `PROCESS` variables to better understand and classify performance information. This annotation layering capability has led us to create a collection of useful annotation libraries that build up knowledge regarding the activity of the operating system, the standard "C" library, and a number of other commonly investigated workload components. These libraries provide new annotation-triggering events as well as higher-level information for other libraries or scripts to easily incorporate and build upon.

```
main(){
   Sort();
   PrintResults();
}
Sort() {
   read();
   printf();
}
PrintResults() {
   printf();
   write();
}
```

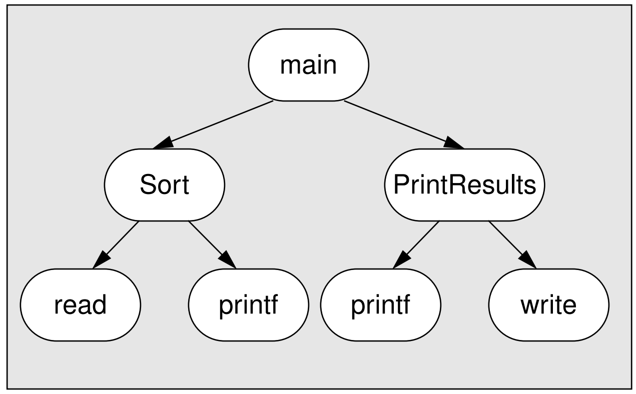

Figure 5.7. Tree-based decomposition of a simple application

5.5.2 Timing trees

One of the advantages of complete machine simulation is the ability to investigate multiprogrammed workloads with all of their associated operating system activity. However, operating system-intensive workloads can be quite difficult to understand. Process scheduling, expensive system call usage, and virtual address translation all contribute to applications' behavior, requiring more elaborate data collection and presentation. *Timing trees* help satisfy this need, providing an easy-to-use mechanism for understanding complex multiprogrammed and operating system-intensive workloads.

As illustrated in Figure 5.7, programs are composed of nested routines, and it is often useful to visualize a program's composition as a tree. More complex applications lead to deeper and wider trees, but the decomposition is still meaningful. The timing tree mechanism extends this type of tree-based decomposition to multiprogramming workloads and easily and efficiently incorporates their use into the data management process. The timing tree mechanism creates a single system-wide tree with a first level node for each process in the system. Whenever a new process is created, a node is automatically added to the system tree. Using annotations set in the operating system's process scheduler, timing trees track the currently executing process, and for each user-level process, timing trees maintain a stack that tracks the process's current routine or phase. This information in turn controls a bucket selector that collects both hardware and higher-level events and assigns them to the active phase.

```
# Use the timing tree library
source "timingTrees.tcl"

# Tell the timing tree what to count for each node
timingTree count { instructions, dataCacheMisses, tlbMisses }
# Indicate the "phases" to assign counts to
timingTree start "Sort" pc SortingApp:Sort::START
timingTree end "Sort" pc SortingApp:Sort::End
timingTree start "fork" pc libc:read::START
timingTree end "fork" pc libc:read::END
...
```

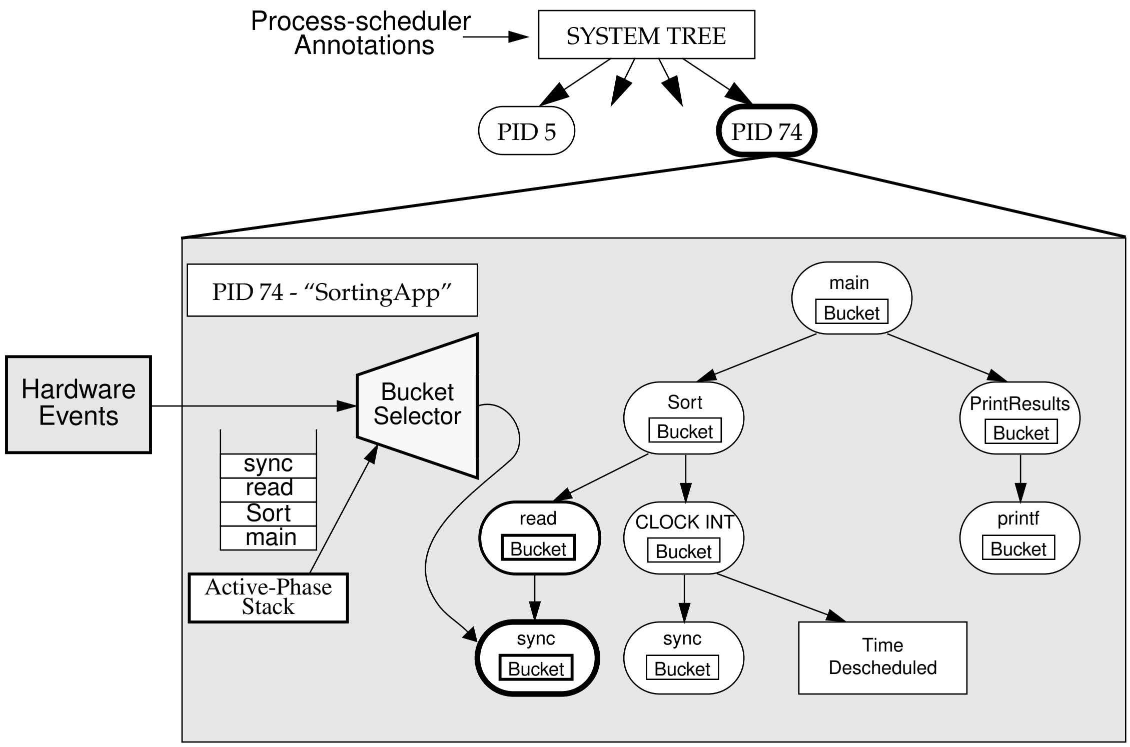

Figure 5.8. Example timing tree decomposition

Figure 5.8 helps illustrate the composition of a timing tree. To specify the structure of a timing tree, an investigator indicates what events should be counted along with the starting and ending points of interesting execution phases. The timing tree mechanism uses annotations and bucket selectors to do the rest. Upon entry into a new phase, the current process's active-phase stack and selector is updated. In the example, the system is currently in the sync phase of the read system call. The bucket selector will thus attribute all instruction, data cache miss, and TLB miss events to this particular node. Timing trees also provide a "time descheduled" bucket for each process. A process can be descheduled at any time, and this extra node allows us to track the effects of operating system scheduling policies on the application's behavior.

When event processing is complete, the resulting timing tree data can be flexibly manipulated and examined to infer many different types of behavioral information. For example, tree branches can be easily collapsed, expanded, or isolated to answer very specific questions regarding workload behavior. In this example, it might be useful to compare the data cache behavior of synchronization acting on behalf of `read` from that occurring during clock interrupt or even during another process's `read` usage.

Timing trees demonstrate how several basic event processing mechanisms can be effectively layered to form new, more sophisticated mechanisms. Additionally, they simplify the specification of a commonly-used execution decomposition, helping customize event-processing to the specific needs of an investigation.

5.6 Data management's performance impact

The previous section introduces a number of different mechanisms and how they efficiently coerce and classify hardware-level data into higher-level behavioral information. This section examines the performance impact of these mechanisms as used in a typical SimOS investigation.

Table 5.1. Operating system library detail levels

Detail Level	Information Provided
0	Nothing (library is not used)
1	Tracks the name and ID of the currently scheduled user process
2	Level 1 + Bucket selector that classifies hardware events according to the current system mode (kernel, user, or idle)
3	Level 2 + Timing tree that classifies hardware events by the system service (system call or exception handler) that they occur during.

A number of our investigations involve tracking operating system behavior during the course of a workload's execution. Recognizing this common need, we have created a reusable library of annotations, bucket selectors, and timing trees that provide operating system data. However, these investigations often require radically different amounts of operating system information, and capturing unneeded behavioral data results in excessive simulation slowdown. To address this problem, the library is parameterized by an integer "detail level" that determines the amount of information that it should collect. Table 5.1 summarizes the library's levels of detail. Increasing the detail level provides more operating system information, but the heavier use of event-processing mechanisms results in higher performance overheads.

Table 5.2 illustrates the performance overhead of each detail level while simulating the uniprocessor compilation workload described in the previous chapter. This workload has significant operating system activity and thus heavily exercises the library. The performance overhead is computed by comparing the simulation time at each detail level to the simulation time of the zero detail level case. For example, the positioning mode runs 8% slower when using the operating system behavior tracking library at detail level than 1 than when not using the library at all.

Table 5.2. Event-processing overheads for the compilation workload

Detail Level	Positioning Mode	Rough Char. Mode	Accurate Mode (Mipsy)	Accurate Mode (MXS)
1	8%	5%	1%	<1%
2	304%	231%	10%	<1%
3	428%	284%	13%	2%

There are two important trends in these overheads. First, within a single simulator execution mode, the performance overhead of event processing can vary quite significantly. More detailed levels (higher numbers) require significantly larger run-time overheads to classify hardware events to higher-level concepts such as execution modes and system services. This trend makes explicit the substantial speed-detail trade-off that can be made within complete machine simulation's data management task. The second trend is seen when comparing the overhead of a single detail level across simulator execution modes. The computation required to support a set number of annotations, bucket-selectors, and timing trees is largely constant, regardless of the simulator execution mode. However, the very high speeds achieved by the positioning and rough characterization modes make this constant overhead quite significant. In contrast, accurate simulation mode's slower speeds allow it to amortize this event-processing overhead over a longer period of time, helping minimize its performance impact. Furthermore, the highest detail levels are almost never used in positioning and rough characterization modes as the accuracy of their results are limited by the accuracy of the simulator mode itself. In practice, users never employ the library in positioning mode, and only use the first detail level with rough characterization mode. To collect more detailed behavioral information, combine the higher detail levels with accurate mode simulation.

The existence of higher-level libraries is essential to the simple specification and use of detailed event-processing. However, even when efficiently implemented, detailed event-processing can have a large impact on complete machine simulation's performance. Just as with adjustable simulator speed and detail characteristics, adjustable event processing and data management allows an investigator to minimize the simulation time required to meet the specific information needs of their study.

5.7 Summary

Complete machine simulation provides an excellent opportunity for helping investigators better understand the behavior of a computer system. However, effectively exploiting this opportunity requires organizing low-level hardware data into higher-level behavioral information and performing this organization as quickly as possible. To address these challenges, complete machine simulation encourages the customization of its data management process to the specific needs of an investigation. This chapter has described how SimOS provides this customization capability through decoupled event generation and processing and with several efficient event-processing mechanisms. As demonstrated in the next chapter, this data management approach has proven to be extremely effective, efficiently providing useful behavioral information for a variety of investigation needs.

Chapter 6
Experiences

The previous chapters describe SimOS and the techniques it uses to address the speed and data management challenges facing complete machine simulation. This chapter describes several of our experiences using SimOS, providing further insight into the effectiveness of the complete machine simulation approach. A tool is only as good as the investigations that it enables, and by this measure SimOS is quite successful. The first part of this chapter describes several SimOS-led investigations and how they have benefited from its use. However, we have also found several limitations with SimOS and the complete machine simulation approach. The second part of this chapter describes the most important of these limitations.

6.1 Investigations enabled by SimOS

Because of its extensive workload support and architectural modeling flexibility, SimOS is an effective tool for a variety of investigations. Furthermore, SimOS's ability to provide both timely, accurate, and customized behavioral data make it an attractive alternative to many existing simulation tools. As a result, SimOS has enabled investigations in many different domains of computer systems research. Recent studies include the investigation of new architectural designs [Bowman97] [Heinrich94] [Nayfeh96] [Olukotun96] [Wilson96], the development of new operating systems [Chapin95b] [Bugnion97], and the performance evaluation of applications [Bugnion96] and operating systems [Rosenblum95] [Teodosiu97] [Verghese97]. This section introduces several of these investigations and how complete machine simulation provided benefits for them.

6.1.1 Characterization of IRIX's performance

One of the earliest investigations utilizing SimOS was a performance characterization of the IRIX operating system executing on both current and future architectural platforms. This goal of this characterization was to help focus IRIX performance tuning efforts on those areas most likely to cause performance problems in the near future. SimOS was critical to this investigation for three reasons. First, the investigation required realistic workloads that stressed the operating system in significant ways; toy applications and micro-benchmarks do not drive the operating system realistically, and thus cannot provide an accurate picture of overall operating system performance. Second, the IRIX operating system is large, complex, and multithreaded. The investigation required flexible data characterization mechanisms to help make sense of the wealth of behavioral information available. In fact, many of the data collection mechanisms of SimOS were developed in direct response to the needs of this study. Third, the goal of the study was to analyze the behavior of the operating system on machines that are likely to appear several years in the future. The flexibility of complete machine simulation allowed us to model hardware platforms well before they were commercially available.

SimOS extensive workload support was essential for observing operating system behavior in a realistic setting. To stress the operating system in realistic ways, we picked workloads that are traditionally run on high-performance workstations and shared-memory multiprocessors: a commercial database workload, a compilation workload, and an engineering simulation workload. SimOS's adjustable speed and detail characteristics enabled the timely preparation and examination of these workloads. First, we used the SimOS positioning to quickly execute past the uninteresting portions of the workloads. This included booting the operating system and running through the initialization phases of the applications. To obtain useful information, we ensured that the system had run long enough to get past the cold start effects due to the operating system boot. Just the

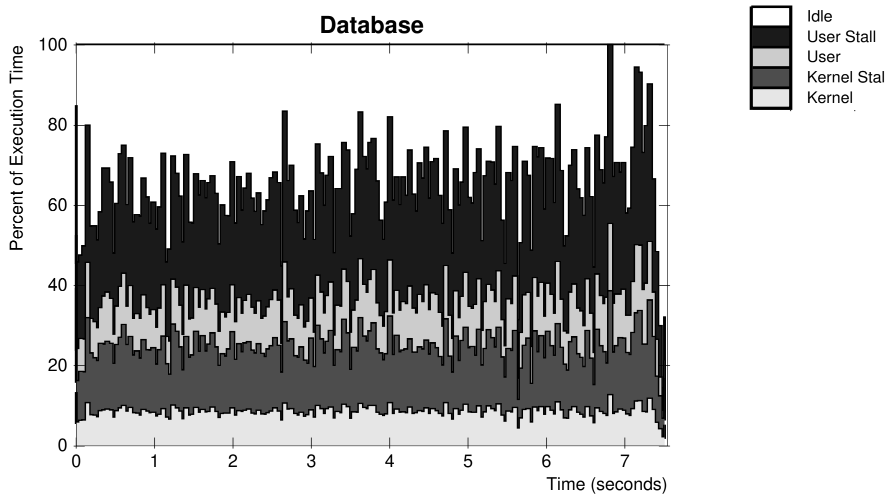

Figure 6.1. Illustration of information generated in rough characterization mode.

initialization of the database workload took several billion instructions, for example, and detailed simulation of this initialization would have taken days of simulation time.

Once the workloads were initialized to an interesting point, we used SimOS's *checkpoint* facility to transfer the entire machine state from the simulation models into a set of files. Just as a simulated machine's hardware state can be transferred between different simulator execution modes, it can also be transferred to the host machine's file system. SimOS allows an investigator to take a checkpoint of the target machine's state at any point during a workload's execution. The checkpoint can be subsequently restored into simulation models to continue execution from the exact point at which the checkpoint was taken. A single checkpoint can be restored in multiple different hardware configurations and characterization modes, and provides a common workload starting point for architectural comparisons.

Each checkpoint was first restored into SimOS's rough characterization mode to obtain a high-level characterization of each workload. The rough characterization mode simulations employed the processor mode bucket selector described in Chapter 5 to decompose each workload's execution into simple, informative components. Illustrated in Figure 6.1 for the database transaction processing workload, the data obtained in rough characterization mode helped determine that the workloads were correctly positioned and had sufficient operating system activity to warrant further investigation. Furthermore, this data showed that the database and engineering workloads' behavior was fairly regular and that examination of a limited portion of the total workload execution time would likely represent its overall behavior. This was essential to reducing them time required for more detailed observation. To obtain the more detailed information, each checkpoint was restored into the SimOS accurate modes, configured to model several different machine configurations. SimOS's architectural modeling flexibility allowed us to create entire machines representative of those shipped in 1994 as well as those likely to ship in 1996 and 1998.

As described in Chapter 5, SimOS provides mechanisms that encourage a user to customize all simulation data to the specific needs of their investigation. For example, they allow a user decompose the execution of a workload into smaller components and then customize all data collection to this decomposition. For this investigation, it was useful to decompose operating system execution according to the particular services that it provides. These services include system calls, virtual-memory fault processing, exceptions, and interrupt processing. Many of the services share the same subroutines and have common execution paths, so a more traditional, procedure-based decomposition would have been less effective. SimOS's timing tree mechanism simplified the generation of this service decomposition. Specification of just over ninety timing tree start and end points completely isolated the activity of each of the operating system's services. The timing tree automatically handles the more difficult aspects of operating system decomposition such as nested interrupts and descheduled processes. With the help of a cache miss event filter, the timing tree provided very detailed information regarding the behavior of each operating system service.

The ability to decompose the operating system's aggregate execution time into more meaningful components was critical to this investigation, allowing a performance comparison of the operating system services across different workloads, number of processors, and generations of hardware. This comparison uncovered significant differences in the performance improvements of different services across architectural trends. This comparison, combined with the specific cache and processor behavior of each service, helped to focus our performance tuning efforts on the portions of IRIX most likely to cause problems in the near future.

SimOS's configurable hardware models, support for realistic workloads, multiple levels of speed, and customizeable data management all played a crucial role in enabling this operating system investigation. The end result of the study was a better understanding of the impact of architectural trends on IRIX's behavior well as a better appreciation for the capabilities of SimOS. Complete details regarding this investigation and its use of SimOS are available in [Rosenblum95].

6.1.2 Design of the FLASH multiprocessor

SimOS has been heavily used to aid the design of the Stanford FLASH multiprocessor [Kuskin94]. The goal of the FLASH project is to build a shared-memory multiprocessor capable of scaling to thousands of MIPS R10000-based processing nodes. To aid the development of FLASH, the architecture team created FLASHLite, a detailed software simulation model of the proposed memory system. FLASHLite was originally designed to be a component of TangoLite, a user-level, execution-driven simulation tool [Goldschmidt93], but development of SimOS provided an opportunity for the architecture group to extend the utility and effectiveness of FLASHLite. SimOS's interface for adding new memory system models minimized the integration effort, and the FLASH design effort was soon realizing several of the benefits of complete machine simulation.

First, SimOS extended the number and type of workloads that could be used to evaluate various FLASH design options. TangoLite is able to capture the user-level memory behavior of applications such as those that comprise the SPLASH [Woo95]. These scientific applications have little or no explicit operating system activity and can thus be easily examined by user-level simulation tools. However, the FLASH multiprocessor is intended to perform well as a general-purpose compute server as well as in support of scientific applications. The use of SimOS extends the evaluation of FLASH design options to include database transaction processing, parallel compilations, and other multiprogrammed workloads that the FLASH machine must efficiently support. These operating system-intensive workloads often exhibit different memory system activity than the SPLASH applications, and have influenced several design decisions. For example, [Heinrich94] found that operating system-intensive workloads exercise the FLASH protocol processor differently than SPLASH workloads, and led to modifications of the protocol processor's data cache organization. Utilization of a wide variety of workload also benefits the design verification effort. The irregular memory reference activity of many operating system-intensive workloads stresses FLASH's cache coherence protocols differently than the SPLASH applications and helped detect and eliminate several deadlock situations in the cache coherence protocols.

Second, SimOS enabled evaluation of the FLASH memory system design options in the context of a complete machine design. For example, the FLASH memory system must maintain cache coherence between processor caches, but it must also support cache-coherent DMA by the disk controller and other system devices. User-level simulators such as TangoLite omit I/O device behavior and are thus incapable of evaluating this aspect of memory system design. Similarly, a multiprocessor memory system interacts closely with a machine's CPU's and caches in supporting memory prefetching, bus error notification, and inter-processor interrupts. SimOS and FLASHLite model all of these interactions, and the full behavioral effects of a proposed FLASH design propagate throughout the simulated system. As a result, the designers were able to obtain accurate information regarding the real-life impact of architectural decisions.

The combination of SimOS and FLASHLite has proven to be an effective source of data for the design of the FLASH multiprocessor as well as for early research into the effectiveness of the FLASH architectural approach [Heinrich94] [Heinlein97a]. In addition to its use in FLASH's architectural design and evaluation, SimOS has aided related compiler and operating system development efforts. These efforts are the subject of the next two case studies.

6.1.3 Hive operating system development

SimOS has also proven to be an effective tool for operating system development. For example, SimOS was heavily used in the design and development of Hive, an operating system designed to improve the reliability and scalability of large general-purpose shared-memory multiprocessors [Chapin95b]. Hive is targeted to run on the FLASH multiprocessor, and provides scalability and reliability improvements through a novel kernel architecture. Rather than running as a single shared-memory program that manages all the machine's resources, Hive partitions the machine and runs an internal distributed system of multiple kernels called c*ells*. This *multicellular* kernel design improves scalability because few kernel resources are shared by processes running on different cells, and also improves reliability because a hardware or software fault damages only one cell rather than the whole system. SimOS provided utility throughout the design and implementation of Hive in a variety of ways. First and foremost, SimOS provided a platform for operating system development. The Hive project began early in the design phase of the FLASH multiprocessor. As a result, Hive development had to begin in the absence of its intended hardware platform. The combination of SimOS and FLASHLite provided early access to the FLASH "hardware" and thus provided a platform for Hive development. Furthermore, early access to potential architectural designs allowed Hive to influence FLASH design decisions. For example, Hive's early efforts at providing reliability suggested that a hardware mechanism for selectively prohibiting remote memory writes across cells would provide significant benefits. This hardware "firewall" mechanism was easily evaluated using SimOS and FLASHLite, and its value was recognized early enough that this feature could be included in the final FLASH design.

In addition to providing early access to a hardware platform, SimOS provides excellent operating system debugging capabilities. SimOS includes an interface for the gdb debugger [Stallman92] that supports examination of the simulated machine. The debugger attaches to SimOS and can set breakpoints, single step through instructions, obtain execution backtraces, and read and write the machine's memory. While these capabilities are typical of most application debugging environments, SimOS supports their use anywhere in the operating system, including the lowest-level exception and interrupt handlers. Debugging efforts are further improved by SimOS's deterministic execution. Many operating system problems are classified as *Heisenbugs* because observation often causes them to behave differently or even disappear. SimOS ensures that operating system behavior is completely repeatable, allowing problems to be more easily examined and eliminated.

SimOS also played a significant role in testing and evaluating Hive's reliability. Hive is designed to minimize the impact of hardware and software faults by detecting when they occur, limiting the spreading of their effects, and then recovering any affected cells. SimOS aided the implementation of this fault containment approach in two important ways. First, annotation scripts were extended with commands for initiating a wide variety of hardware and software faults. For example, these scripts can alter the contents of main memory, corrupt interconnection network routing tables, or even disable entire portions of the machine. Initiation of similar faults on real hardware is significantly more difficult if not impossible. Additionally, SimOS's data management mechanisms provided excellent visibility into Hive's recognition of and reaction to these faults. When a fault was not properly contained, program counter-based annotations placed throughout the kernel helped determine why.

In summary, SimOS was an essential tool in the development of Hive, enabling its complete design and implementation well before its targeted hardware platform was available. Additional information regarding Hive and its use of SimOS is available in [Chapin97].

6.1.4 Improving the SUIF parallelizing compiler

SimOS was also heavily utilized in a recent investigation of automatically-parallelized applications generated by the SUIF compiler [Wilson94]. The SUIF compiler automatically transforms an application designed for uniprocessor execution into one that can exploit multiprocessor hardware. The SUIF group became interested in SimOS because many automatically-parallelized SPEC95fp applications were not achieving the expected performance improvements, and existing tools were unable to identify the reason. The goal in using SimOS was to discover the sources of the application's performance problems and to suggest compiler modifications that would help eliminate them.

Because SimOS can run the IRIX operating system, workload preparation was simple. Researchers compiled SUIF applications to run on a Silicon Graphics workstations and copied the executables and their input files onto the simulated machine's disk. Note that no modifications to the applications were necessary to run on SimOS. Furthermore, because SimOS's accurate simulation mode closely models existing Silicon Graphics machines, the SUIF group could be confident that application performance gains exhibited on the simulated machine would translate into performance gains on the actual hardware.

SimOS's support for multiple levels of simulation speed is essential to investigating the long-running, SUIF-generated applications. SimOS's positioning mode enables fast initialization and positioning of the parallelized SPEC applications. This process takes less than ten minutes of simulation time for each application, including the booting of IRIX. Once the applications are in an interesting position, more detailed investigation can begin. To avoid excessive simulation time due to the long execution times of the SPEC95fp benchmarks, several minutes on today's fastest non-simulated machines, SimOS's rough characterization mode was initially used to observe the application's basic behavioral characteristics. The resulting data illustrated that each application's behavior was quite regular, exhibiting very similar behavior in repeating intervals. This allowed the researchers to limit examination with the accurate modes to a small number of representative intervals. The information obtained during these slowly simulated intervals was extrapolated to apply to the entire execution, further decreasing the simulation time required for this investigation.

SimOS's customizeable data management mechanisms were also essential to this investigation, identifying several performance problems and suggesting solutions. As depicted in Figure 6.2, a SUIF-generated application has a master/slave structure. The master thread coordinates the application's execution, determining when slave threads should help execute parallelized loops. At the end of each iteration, the threads synchronize at a barrier to maintain loop ordering constraints. The regular structure of SUIF applications provides an obvious approach to decomposition. Each application is decomposed into its sequential execution, parallelized execution, and synchronization

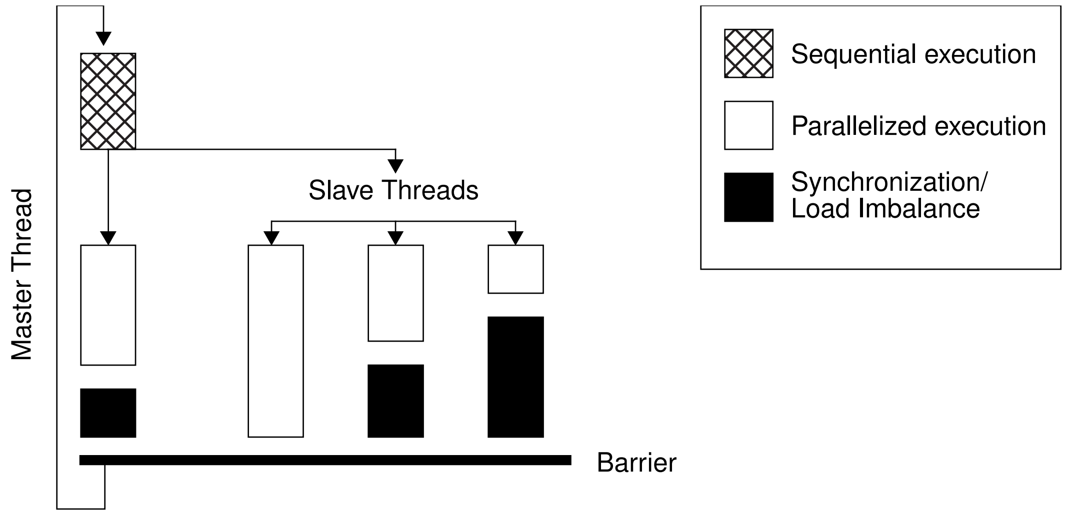

Figure 6.2. Structure of SUIF-generated applications

time. SimOS bucket selectors separated "useful" execution time from compiler-added administration and synchronization overheads. Additional annotations decomposed synchronization time into the time spent executing barrier code and time spent waiting at this barrier when slave execution times were unbalanced. The resulting data suggested that the fine granularity of parallelism exploited by the SUIF compiler was resulting in large overheads due to the barrier codes. This information led to the development of a new SUIF barrier mechanism more optimized for sporadic thread arrival.

SimOS provided more specific performance information through the use of two cache miss classification event filters. The first filter categorizes multiprocessor invalidation cache misses according to the true or false sharing types defined in [Dubois93]. Using this filter, SimOS reported a striking behavior in *cfft2*, one of the NASA7 benchmarks. SimOS reported that 84% of all cache misses on the primary matrix used in the computation were due to false sharing. This suggested that if the compiler were to align all shared data structures on cache line boundaries, these false sharing cache misses could be completely eliminated. Additional cache miss information was generated via the cache miss classification filter described in Chapter 5. This information generated by this filter indicated that conflict misses were a problem for several applications in the SPEC95fp benchmark. Using higher-level annotations triggered by data cache misses, SimOS collected information about the physical addresses of frequently used cache lines. This data suggested that the operating system's page mapping policy resulted in an inefficient utilization of the processor caches during parallel execution. This information directly led to the development of a new operating system page-mapping algorithm that significantly improves the performance of compiler-parallelized applications. The ability of SimOS to precisely locate and classify cache misses was instrumental to the development of the algorithm. The modifications suggested by SimOS have led the SUIF compiler to generate significantly better performing code. One of the most tangible results of these SimOS-suggested compiler improvements was the highest SPEC95fp ratio reported to date. More detailed information on this investigation is available in [Bugnion96].

6.2 Limitations of complete machine simulation

SimOS has provided significant benefit to a number of investigations. However, there are several limitations that we have encountered with SimOS and with the complete machine simulation in general. This section describes the most restrictive of these limitations and suggests some potential techniques for reducing their impact.

6.2.1 Poor scalability

SimOS's most restrictive limitation is its poor performance when simulating large numbers of processors. This problem applies to simulation of multiprocessors as well as distributed systems, and is an inherent limitation of the complete machine simulation approach. This limitation stems from the fundamental hardware emulation task that a complete machine simulator must perform. A complete machine simulator must always model at least enough hardware functionality to support the execution of operating systems and application programs. As such, there is a minimal amount of simulation work that must be performed for each processor or machine. Furthermore, this computation requirement scales at least linearly with the number of processors being simulated, hindering the examination of large multiprocessors and distributed systems. In contrast, analytic models or tools that model software execution at a more abstract level can often avoid this linear performance degradation.

One way to improve performance is to spread the simulation computation across multiple host processors. As reported in [Witchel96], we have had some initial success with a parallel Embra-based SimOS positioning mode. Embra can run as multiple parallel processes where each process simulates a disjoint set of target machine CPU's, enabling high-speed multiprocessor emulation. However, this approach has limited simulation accuracy. The processors in a shared-memory multiprocessors interact extremely frequently. Inter-processor communication takes place within the memory system, and any activity that occurs on the memory bus can affect the timing and behavior of this interaction. To accurately model these interactions requires the simulated processors' notions of time to be closely synchronized, limiting the speedup available via parallel execution.

Simulation of distributed systems provides a better opportunity for exploiting a multiprocessor host or even multiple host machines. Interaction between a distributed system's machines occurs via ethernet or other networking technologies. Because this interaction occurs much less frequently than in shared-memory multiprocessors, there is additional opportunity for performance gains through the use of parallel simulation gains. However, accurately modeling network activity such as ethernet collisions still requires substantial synchronization, limiting the potential performance improvement. The combination of significant resource requirements and need for a centralized notion of time means that an accurate complete machine simulation will always have performance scaling problems. Even if the above parallel execution opportunities were fully realized, the complete machine simulation would only scale to tens or possibly hundreds of processors. This may be acceptable for studying small systems, but not for investigating large-scale multiprocessors or internet-style networks with thousands of nodes.

6.2.2 Non-compatible changes require substantial implementation

The ability to support the execution of applications with all of their associated operating system activity is one of complete machine simulation's biggest advantages, but it also places significant responsibility on the simulated hardware components. In contrast to application-level and trace-driven simulators, SimOS's simulated hardware must fulfill all of the operating system's functional requirements while correctly interacting with the rest of the simulated hardware. As a result, radical architectural changes require substantially more implementation in a complete machine simulation environment than in many other tools. For example, instruction set alterations and other “non-compatible” architectural changes are difficult to evaluate unless an entire workload is modified or recompiled to utilize them. It is far simpler to apply such modifications to simple, isolated applications or micro-benchmarks than to more complex workloads including a complete operating system. As a result, we have found the complete machine simulation approach to be better suited to the latter phases of architectural design. Simpler trace-based tools are often more useful for high-level exploration of a wide design space. Once the basic design parameters have been determined, complete machine simulation is effective at evaluating more specific configuration details.

6.2.3 Interaction with the non-simulated world

Even when running in positioning mode, a SimOS-modeled computer is significantly slower than a hardware implementation. As a result, non-simulated objects that interact with the simulated machine appear to be substantially faster than they should be. For example, a human interacting with a simulated machine’s console would appear to type several hundred times faster than normal. This problem is worsened when simulating operating systems and applications that utilize graphical user interfaces. Proper graphical responsiveness is required, but difficult to provide under positioning mode and virtually impossible to provide in more detailed simulation modes. Furthermore, interaction with humans is non-deterministic, compromising SimOS’s ability to provide repeatable workload executions. There has been some progress dealing with these problems through the use of pre-recorded interaction scripts. SimOS allows investigators to write Tcl scripts that recognize the console’s output and emulate a human response. These scripts provide more appropriately timed interactions and enable repeatable workload execution. However, they do require advanced knowledge of what interaction should occur and have not yet been applied to the non-textual interactions of a graphical user interface.

Similar problems occur when a SimOS-modeled computer communicates with non-simulated computers. Client-server applications such as database engines and web servers require interaction among networked machines, and SimOS provides the required communication capability. However, a SimOS machine is substantially slower than a real machine, resulting in network time-outs, poorly timed interaction, and unrepresentative communication patterns. Additionally, non-simulated machines are not under SimOS's control, and their non-deterministic network interactions result in non-repeatable workload execution. To help address these problems, SimOS can model multiple machines simultaneously. Just as it interleaves the execution of a multiprocessor's CPU's, SimOS can interleave the execution of different machines' CPU's. All communication in this simulated distributed system occurs through normal network protocols and travel across a simulated LAN. This SimOS configuration better coordinates the machines' notions of time and provides deterministic execution of network-based workloads. Unfortunately, there is significant overhead to this approach and workload slowdown is proportional to the number of simulated machines.

6.2.4 Substantial implementation costs

Another limitation of the complete machine simulation approach that should not be overlooked is its substantial implementation cost. Complete machine simulators encompass significantly more functionality than most existing simulation tools, resulting in substantial programming, debugging, and code maintenance costs. As an example, the current version of SimOS consists of several hundred thousand lines of "C" code and has required several man-years of implementation effort. However, complete machine simulation's improved data accuracy and applicability warrant this cost. Additionally, complete machine simulation provides computer system behavioral information that is currently available only through the use of multiple orthogonal tools. This ability to use a single tool for several different research needs can ultimately reduce a research group's long-term tool implementation, training, and deployment costs.

6.3 Summary

In summary, our initial experiences with SimOS have been extremely positive. SimOS has enabled several investigations that were impossible with existing simulation tools and provides valuable infrastructure for many types of computer systems research.

Furthermore, a public distribution of SimOS is leading to further adoption of the complete machine simulation approach in both academic and commercial research. Hopefully this increased usage will lead to many more successful investigations as well as to additional techniques for coping with complete machine simulation's limitations.

Chapter 7
Related Work

This chapter compares complete machine simulation to other popular tools and techniques used to investigate computer system behavior. This chapter focuses on the three most important features of complete machine simulation; providing complete computer system behavioral information, providing behavioral information in a timely manner, and efficient converting low-level hardware data into higher-level workload information.

7.1 Providing complete computer system behavioral information

Several studies have recognized the importance of system-level behavior, emphasizing that it needs significantly more attention in hardware design and performance investigations [Agarwal88] [Anderson91] [Chapin95a] [Ousterhout90]. However, tools for investigating computer system behavior have traditionally been able to observe only the user-level portion of a workload's execution. To address this deficiency, a number of researchers have focussed on the development of new techniques and tools for investigating complete computer system behavior. This section describes several of these techniques and tools and compares them to the complete machine simulation approach.

7.1.1 Trace-driven simulation

Trace-driven simulation is by far the most common technique for computer system investigation. Trace-driven simulation tools use software instrumentation and hardware monitoring to collect “traces” of an existing system’s dynamic execution behavior, and these traces provide input for hardware simulators. Software instrumentation has long been used to trace user-level programs, but recent advances have enabled its use in investigating operating systems as well. For example, the Epoxie tool described in [Chen94] can rewrite object files at link time to record a trace of instruction and memory references. Epoxie addresses many of the challenges of rewriting kernel code and can record the complete memory system behavior of a workload. Similarly, the PatchWrx system described in [Perl97] rewrites binary executable images, “patching” them with code that generates complete address traces in a very compact format. PatchWrx is extremely fast and has supported investigations of the Windows NT operating system. Other software instrumentation approaches capable of operating system tracing include [Maynard94] and [Wall87].

Hardware-based trace collection is a more popular technique for collecting traces across an workload’s entire execution. One of the earliest examples of a hardware-based trace collection tool capable of operating system investigation is the ATUM system [Agarwal86]. In ATUM, the microcode of a VAX 8200 processor was modified to record the addresses of a workload’s memory references. However, reloadable microcode is no longer popular, leading to the development of several new hardware-based trace collection techniques. For example, the Monster [Nagle92] and BACH [Grimsrud93] systems capture signals from modern CPUs to collect instruction and data address traces. Another popular approach utilizes memory bus-monitoring hardware [Chapin95a] [Torrellas92] [Vashaw92]. This hardware collects a trace of all memory bus traffic that occurs during a workload’s execution. In each case, the traces include all operating system and user-level activity, providing hardware simulators with the opportunity to investigate complete computer system behavior.

As described in Chapter 2, trace-driven simulation is often faster and easier to implement than complete machine simulation. However, complete machine simulation's extensive workload support, non-intrusive observation, and ability to propagate hardware effects throughout the system provides more accurate behavioral information. To realize the advantages of both simulation approaches, complete machine simulation supports a trace-generation mode. Simple annotations save the desired trace information to a file and provide input for the numerous trace-driven hardware simulators that already exist.

7.1.2 Hardware counters

A recent trend in CPU design is the inclusion of hardware to count the occurrence of processor-related events. These hardware counters exist in most modern processors including the Intel Pentium [Mathisen94], IBM Power2 [Welbon94], DEC Alpha [Digital95], HP PA-8000 [Hunt95], and MIPS R10000 [Zagha96], and can track such processor events as cache and TLB misses, branch prediction behavior, and pipeline stalls. Furthermore, the counters are integrated directly onto a processor, providing results at very high speeds and with minimal intrusiveness. Hardware counters are often always active, providing detailed information regarding all of a workload's execution behavior. [Chen95] provides an excellent example of the investigative opportunities that these counters enable. In this research, the investigators utilized the Intel Pentium's counters to examine the performance characteristics of personal computer operating systems.

There are however some limitations to the effectiveness of hardware counters. First and foremost, these counters are not extensible; their data is restricted to just those events that were built into the hardware. Furthermore, hardware counters provide detailed information regarding a workload behavior on existing processors, but are less effective at predicting the workload's behavior on future hardware. Despite these limitations, the speed and visibility available with on-chip counters makes them a promising tool for computer system investigation.

7.1.3 Functional simulation

Also called instruction-level or program-driven simulation, functional simulators are software programs that fetch, decode, and execute processor instructions, applying the results of each instruction to a conceptual target machine. Functional simulation has traditionally been applied to tasks without strict speed requirements such as hardware validation or the preservation of historical software [Burnet96]. Functional simulation is also commonly used to investigate just the user-level behavior of applications. Popular examples of these *user-level* functional simulators are MINT [Veenstra94], PAINT [Stoller96], Shade [Cmelik94], and Talisman [Bedichek95].

More recent research has extended functional simulators to the investigation of a workload's user- and kernel-level behavior. For example, [Anderson94] and [Poursepanj94] describe PowerPC instruction-set simulators capable of executing commercial operating system and application code. Similarly, SimICS [Magnussen95] is a functional simulator capable of investigating SPARC-based applications and operating systems.

Complete machine simulation builds upon these and other simulation efforts in an attempt to extend the applicability and usefulness of functional simulation to additional fields of computer systems research.

7.2 Providing simulation results in a timely manner

Speed is always a limiting factor in the effective simulation of computer hardware, and this section compares complete machine simulation's techniques for providing timely simulation results to related research efforts. Specifically, this section examines the use of multiple levels of simulation speed and techniques for providing high-speed machine simulation.

7.2.1 Utilizing multiple levels of simulation speed

One of the earliest tools to exploit the inherent trade-off between simulation speed and simulation detail was the Proteus system [Brewer92], an execution-driven, user-level simulator that models MIMD multiprocessor systems. Proteus was designed with modular interfaces that support the inclusion of interchangeable hardware component models. At compile-time users choose the model of each component that provides an appropriate level of speed and detail. Complete machine simulation extends Proteus's use of multiple levels of simulation speed by allowing *dynamic* selection of simulation components and by extending its applicability beyond user-level simulation.

[Argade94] presents another interesting example of using multiple levels of simulation speed and detail. This system uses a combined hardware and software approach to obtain simulation data as quickly as possible. Their approach uses real hardware for high-speed workload positioning. Once the workload is in an interesting state, the system can save the hardware's state to disk. Just like SimOS's checkpoints, the hardware state provides the input to simulation models for more detailed system examination.

7.2.2 High-speed machine simulation

Critical to the success of complete machine simulation is the availability of high-speed simulation modes that can be used for workload positioning. Several other researchers have recognized the importance of high-speed machine simulation and developed methods for providing it. For example, Talisman [Bedichek95] uses a technique called threaded code to perform very high-speed multicomputer simulation. Talisman is an impressive simulation environments, modeling full processor behavior and achieving timing accuracy relative to a hardware prototype. While Talisman models a processor's supervisor mode, it does not support an operating system; it runs a subset of Intel's NX message passing library. As another example, Shade [Cmelik94] is a cross-architectural, instruction set simulator that can investigate the user-level behavior of most any SPARC application. Shade uses dynamic translation of binary code as well as sophisticated code caching techniques to achieve very high simulation speeds. As described earlier, Embra uses many of the techniques pioneered by Shade, extending them to support a full operating system execution as well as multiprocessor workloads. Research into high-speed simulation techniques is becoming increasingly popular. In addition to its applicability to computer system behavioral investigation, high-speed machine simulation techniques are being widely applied to the domain of cross-platform application support [Hookway97] [Insignia97] [Sun97].

As mentioned in the previous chapter, we are investigating the use of a multiprocessor host machine to speed up the simulation of multiprocessor target machine behavior. Several research projects have applied this approach to the investigation of user-level memory system behavior. For example, the Cerebus Multiprocessor Simulator [Brooks88] was one of the first parallel implementations of a multiprocessor simulator and was used to investigate the behavior of parallel algorithms on configurable memory systems. Similarly, Tango Lite supports a parallel execution mode where each target processor is modeled as a single thread of execution that can run concurrently with other threads. The threads communicate with each other at the memory system level to allow the investigation of program behavior and cache coherence protocols. As another interesting example, the Wisconsin Wind Tunnel [Reinhardt93] uses the memory ECC bits on a Thinking Machines CM-5 to quickly estimate the cache behavior of large parallel programs. Regardless of the implementation, it is clear that parallel simulator execution is essential to the timely investigation of large multiprocessors and distributed systems. Future SimOS implementations will build upon these existing tools to more fully apply parallelization to the complete machine simulation approach.

7.3 Managing low-level simulation data

Every computer simulation tool faces the challenge of converting hardware-level simulator data into more useful behavioral information. This section compares SimOS's approach of investigation-specific data management to related simulation data management efforts. Specifically, it examines the conversion of low-level hardware data into higher level workload information and the use of investigation-specific data management to reduce the overhead of this conversion.

7.3.1 Workload-level data classification and reporting

SimOS classifies low-level hardware data into higher level information that is more useful to an investigator. Numerous tools have recognized this need and implemented different forms of data mapping functionality. For example, gprof [Graham83] is an execution profiling tool that assigns processor time to an application's procedures. Gprof instruments an application's procedures to generate a procedure call graph at run time and uses program counter sampling to provide a statistical estimate of where an application spends its time. Furthermore, this information is categorized by procedure and offers a high level view of where a program may be best optimized.

Memspy [Martonosi92] is another tool that converts low-level hardware data into application-oriented information. Like gprof, Memspy uses software instrumentation to indicate the entry and exit point to all of an application's procedures. At run-time, this additional code helps create a tree-based decomposition of the application's execution activity. Additionally, each application uses modified calls to `malloc()` and `free()` to track which ranges of memory correspond to different data structures. Memspy uses this higher-level workload knowledge to charge all cache misses to the responsible procedures and data structures. FLASHPoint [Martonosi96] provides similar data classification, relying on a programmable memory system controller to charge cache misses to the responsible procedures and data structures. The application-oriented information generated by these tools helps an investigator understand and improve the memory system behavior of an application and inspired the creation of SimOS's address tables and timing trees.

Researchers have also focussed on mapping low-level hardware data to higher level workload information in trace-based simulations. The typical approach is to explicitly alter a workload to incorporate higher-level workload information into the trace. For example, the hardware monitor used in [Chapin95a] and [Torrellas92] could only capture memory references that reached the memory bus. To provide knowledge of workload-level concepts, the operating system was heavily modified to output uncached references indicating the entry and exit points to important procedure. This additional information allowed cache and memory system behavior to be categorized by the responsible operating system procedure.

7.3.2 Customized data management overheads

In addition to providing workload-level behavioral information, a simulation's data management must be as efficient as possible. Complete machine simulation's approach to minimizing the overhead of data management is similar to code annotation tools such as ATOM [Srivastava94], EEL [Larus95], ETCH [Romer97], MINT++ [Veenstra97], and TracePoint [TracePoint97]. These tools provide flexible interfaces that enable users to annotate an application's individual instructions, basic blocks, and data references. Just as in SimOS, these annotations can count and classify events as well as query machine state. Furthermore, if no annotation is inserted at a given location, these tools do not add any code, allowing the minimal degree of data management overhead for a particular application and investigation.

Paradyn [Miller95] provides another interesting example of the interaction between data collection and simulation speed. Paradyn is an execution-driven performance measuring tool designed for the investigation of parallel and distributed programs. Paradyn dynamically instruments an application to collect various types of performance data. During the execution of an application, Paradyn recognizes troublesome sections of code and directs the event generation mechanism (a code annotator) to produce more detailed events for processing. This allows Paradyn provide the minimal level of data processing needed to properly investigate an application's execution behavior, minimizing the time required to perform this investigation.

Chapter 8
Conclusions

The research described in this dissertation attempts to help investigators better understand the behavior of increasingly complex computer systems. This dissertation argues that complete machine simulation is an effective approach for gathering the information needed for this understanding. In support of this argument, the work described in this dissertation makes three primary contributions:

- **Demonstration of the significant benefits that complete machine simulation provides to many types of computer systems research.**

  Complete machine simulation offers several benefits to computer systems research including extensive workload support, accurate machine modeling, and comprehensive data collection. Our experiences with the SimOS implementation of complete machine simulator have shown these benefits to be quite valuable, enabling several studies not possible with existing tools and techniques.

- **Demonstration that adjustable levels of simulation speed and detail help complete machine simulation provide timely data.**

  The biggest challenge facing complete machine simulation's acceptance is its performance, and this work demonstrates how adjustable simulation speed and detail characteristics address this challenge. In implementing the SimOS version of adjustable simulation characteristics, this work recognizes the importance of three general simulation execution modes and the ability to dynamically switch between them during the course of a workload's execution.

- **Specification and implementation of efficient and flexible mechanisms for addressing complete machine simulation's data management challenges.**

  The other major challenge for complete machine simulation is efficient conversion of hardware-level data into higher level computer system behavioral information, and this work demonstrates how supporting investigation-specific data management addresses this challenge. Specifically, this work introduces efficient and flexible mechanisms that allow an investigator to customize all simulation data classification and reporting to meet the specific needs of their study.

Complete machine simulation has fundamentally changed the way that we perform computer systems research at Stanford University: architectural evaluations are driven with more representative workloads, operating system design occurs on a more flexible and forgiving platform, and application performance tuning efforts incorporate all relevant behavioral information. Furthermore, we are extending SimOS to support several new architectures and operating systems and are freely distributing the SimOS source code. The goals of this public distribution are to enable new computer system investigations and to encourage complete machine simulation's acceptance as a critical component of modern computer systems research.

References

[Agarwal86] A. Agarwal, R. Sites, and M. Horowitz. "ATUM: A new technique for capturing address traces using microcode." In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119-127, June 1986.

[Agarwal88] A. Agarwal, J. Hennessy, and M. Horowitz. "Cache performance of operating system and multiprogramming workloads." *ACM Transactions on Computer Systems*, 6(4), pp. 393-431, November 1988.

[Anderson91] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. "The interaction of architecture and operating system design." *SIGPLAN Notices*, vol. 26, no. 4, pp. 108-120, April 1991.

[Anderson94] W. Anderson. "An overview of Motorola's PowerPC simulator family." *Communications of the ACM*, vol.37, no.6, pp. 64-69, June 1994.

[Argade94] P. Argade, D. Charles, and C. Taylor. "A technique for monitoring run-time dynamics of an operating system and a microprocessor executing user applications." *SIGPLAN Notices,* vol. 29, no. 11, pp. 122-131, October 1994.

[Bedichek95] R. Bedichek. "Talisman: Fast and accurate multicomputer simulation." *Performance Evaluation Review*, vol. 23, no. 1, pp. 14-24, May 1995.

[Bennett96] James Bennett and Mike Flynn. "Latency tolerance for dynamic processors." Technical report CSL-TR-96-687, Computer Systems Laboratory, Stanford University, January 1996.

[Borg89] A. Borg, R Kessler, G. Lazana, and D. Wall. "Long address traces from RISC machines: Generation and analysis." *WRL Research Report 89/14*, Digital Equipment, 1989.

[Bowman97] N. Bowman, N. Caldwell, C. Kozyrakis, C. Romer, and H. Wang. "Evaluation of existing architectures in IRAM systems." In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[Boyle87] J. Boyle, R. Butler, T. Disz, B. Blickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, & Winston, 1987.

[Brewer92] E. Brewer, A. Colbrook, C. Dellarocas, and W. Weihl. “PROTEUS: A high-performance parallel-architecture simulator.” *Performance Evaluation Review,* vol. 20, no. 1, pp. 247-248, June 1992.

[Brooks88] E. Brooks, T. Axelrod, and G. Darmohray. “The Cerberus multiprocessor simulator.” Lawrence Livermore National Laboratory technical report, Preprint UCRL-94914, 1988.

[Bugnion96] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. “Compiler-directed page coloring for multiprocessors.” *SIGPLAN Notices*, vol. 31, no. 9, pp. 244-255, October 1996.

[Bugnion97] E. Bugnion, S. Devine, and M. Rosenblum. “Disco: Running commodity operating systems on scalable multiprocessors.” In *Proceedings of The 16th ACM Symposium on Operating Systems Principles*, pp. 143-156, October 1997.

[Burnet96] M. Burnet and R. Supnik. “Preserving computing’s past: restoration and simulation.” Digital Technical Journal, vol.8, no.3, pp. 23-38, 1996.

[Chapin95a] J. Chapin, S. Herrod, M. Rosenblum, and A. Gupta. “Memory system performance of UNIX on CC-NUMA multiprocessors.” *Performance Evaluation Review*, vol. 23, no. 1, pp. 1-13, May 1995.

[Chapin95b] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. “Hive: Fault containment for shared-memory multiprocessors.” In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 12-25, December 1995.

[Chapin97] J. Chapin. “Hive: Operating system fault containment for shared-memory multiprocessors.” Ph.D. Thesis, Stanford University, January 1997.

[Chen93] J. Chen and B. Bershad. “The impact of operating system structure on memory system performance.” *Operating Systems Review,* vol. 27, no. 5, pp. 120-133, December 1993.

[Chen94] J. Chen, D. Wall, and A. Borg. “Software methods for system address tracing: implementation and validation.” Digital WRL Research Report 94/6, September 1994.

[Chen95] J. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Selzer and M. Smith. “The measured performance of personal computer operating systems.” *ACM Transactions on Computer Systems,* vol. 14, no. 1, pp. 3-40, February 1996.

[Cmelik94] R. Cmelik and D. Keppel, “Shade: A fast instruction set simulator for execution profiling.” *Performance Evaluation Review*, vol. 22, no. 1, pp. 128-137, May 1994.

[Digital95] Digital Equipment Corporation. “pfm - The 21064 Performance Counter Pseudo-Device.” DEC OSF/1 Manual pages, 1995.

[Dubois93] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. “The detection and elimination of useless misses in multiprocessors.” *Computer Architecture News,* vol. 21, no. 2, pp. 88-97, May 1993.

[Goldschmidt93] S. Goldschmidt. “Simulation of multiprocessors: Accuracy and performance.” Ph.D. Thesis, Stanford University, June 1993.

[Graham83] S. Graham, P. Kessler, and M. McKusick. “An execution profiler for modular programs.” *Software - Practice and Experience*, vol. 13, pp. 671-685, August 1983.

[Gray93] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, 1993.

[Grimsrud93] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson. “BACH: A hardware monitor for tracing microprocessor-based systems.” In *Microprocessors and Microsystems*, vol. 17, no. 8, pp. 443-459, October 1993.

[Heinlein97a] J. Heinlein, R. Bosch, Jr., K. Gharachorloo, M. Rosenblum, and A. Gupta. “Coherent block data transfer in the FLASH multiprocessor”. In *Proceedings of the 11th International Parallel Processing Symposium*, pp. 18-27, April 1997.

[Heinrich94] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The performance impact of flexibility in the Stanford FLASH multiprocessor.” *SIGPLAN Notices,* vol. 29, no. 11, pp. 274-284, October 1994.

[Herrod97] S. Herrod, M. Rosenblum, E. Bugnion, S. Devine, R. Bosch, J. Chapin, K. Govil, D. Teodosiu, E. Witchel, and B. Verghese. “The SimOS User Guide.” `http://www-flash.stanford.edu/SimOS/userguide/`.

[Hookway97] R. Hookway. "DIGITAL FX!32 running 32-Bit x86 applications on Alpha NT." In *Proceedings of IEEE COMPCON 97*. pp. 37-42, February 1997.

[Hunt95] D. Hunt. "Advanced performance features of the 64-bit PA 8000." In *Proceedings of COMPCON'95*, pp. 123-128, March 1995.

[Insignia97] Insignia. SoftPC product information. `http://www.insignia.com`.

[Kotz94] D. Kotz, S. Toh, and S. Radhakrishnan. "A detailed simulation of the HP 97560 disk drive." Dartmouth College technical report PCS-TR94-20, 1994.

[Kuskin94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford FLASH multiprocessor." In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pp. 302–313, April 1994.

[Larus95] J. Larus and E. Schnarr. "EEL: machine-independent executable editing." In *SIGPLAN Notices*, vol. 30, no. 6, pp. 291-300, June 1995.

[Lebeck95] A. Lebeck and D. Wood. "Active memory: A new abstraction for memory-system simulation." *Performance Evaluation Review*, vol. 23, no. 1, pp. 220-230, May 1995.

[Magnusson95] P. Magnusson and J. Montelius. "Performance debugging and tuning using an instruction-set simulator." SICS Technical Report T97:02, 1997.

[Martonosi92] M. Martonosi, A. Gupta, and T. Anderson. "MemSpy: Analyzing memory system bottlenecks in programs." *Performance Evaluation Review,* vol. 20, no. 1, pp. 1-12, June 1992.

[Martonosi96] M. Martonosi, D. Ofelt, and M. Heinrich. "Integrating performance monitoring and communication in parallel computers." *Performance Evaluation Review,* vol. 24, no. 1, pp. 138-147, May 1996.

[Mathisen94] T. Mathisen. "Pentium secrets." *Byte Magazine*, pp. 191-192, July 1994.

[Maynard94] A. Maynard, C. Donnelly, and B. Olszewski. "Contrasting characteristics and cache performance of technical and multi-user commercial workloads." *SIGPLAN Notices,* vol. 29, no. 11, pp. 145-156, November 1994.

[Miller95] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. “The Paradyn parallel performance measurement tools.” In *IEEE Computer,* pp. 37-46, November 1995.

[MIPS95] MIPS Technologies, Inc. *R10000 Microprocessor User’s Manual*, 2nd edition, June 1995.

[Nagle92] D. Nagle, R. Uhlig, and T. Mudge. “Monster: a tool for analyzing the interaction between operating systems and computer architectures.” Technical Report CSE-TR-147-92, University of Michigan, 1992.

[Nayfeh96] B. Nayfeh, L. Hammond, and K. Olukotun. “Evaluation of design alternatives for a multiprocessor microprocessor.” *Computer Architecture News,* vol. 24, no. 2, pp. 67-77, May 1996.

[Olukotun96] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. “The case for a single-chip multiprocessor.” *SIGPLAN Notices*, vol. 31, no. 9, pp. 2-11, October 1996.

[Ousterhout90] J. Ousterhout. “Why aren’t operating systems getting faster as fast as hardware?”, In *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.

[Ousterhout94] J. Ousterhout. “Tcl and the Tk Toolkit.” *Addison-Wesley Publishing Company*, Reading, Mass., 1994.

[Perl97] S. Perl and R. Sites. “Studies of Windows NT performance using dynamic execution traces.” Digital SRC Research Report 146, April 1997.

[Poursepanj94] A. Poursepanj, “The PowerPC performance modeling methodology.” In *Communications of the ACM*, vol. 37, no. 6, pp. 47-55, June 1994.

[Reinhardt93] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. “The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers.” *Performance Evaluation Review*, vol. 21, no. 1, pp. 48-60, June 1993.

[Romer96] T. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J. Baer, B. Bershad, and H. Levy. “The structure and performance of interpreters.” *SIGPLAN Notices*, vol. 31, no. 9, pp. 150-159, September 1996.

[Romer97] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. “Instrumentation and optimization of

Win32/Intel executables using Etch." In *Proceedings of the USENIX Windows NT Workshop*, pp. 1-7, August 1997.

[Rosenblum95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. "Complete computer system simulation: The SimOS approach." *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 3, no. 4, pp. 34–43, Winter 1995.

[Rosenblum97] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. "Using the SimOS machine simulator to study complex computer systems." *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78-103, January 1997.

[SPEC97] Standard Performance Evaluation Corporation. The SPEC benchmark suite. `http://www.specbench.org`.

[Srivastava94] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools." *SIGPLAN Notices*, vol. 29, no. 6, pp. 196-205, June 1994.

[Stallman92] R. Stallman and R. Pesch. "Using GDB." Edition 4.04 for GDB version 4.5, March 1992. `ftp://prep.ai.mit.edu/pub/gnu/`.

[Stoller96] L. Stoller, M. Swanson, and R. Kuramkote. "Paint: PA instruction set interpreter." Technical Report UUCS-96-009, University of Utah, 1996.

[Sun97] Sun Microsystems. WABI 2.2 Product Information. `http://www.sun.com/solaris/wabi`.

[Teodosiu97] D. Teodosiu, J. Baxter, K. Govil, J. Chapin, M. Rosenblum, and M. Horowitz. "Hardware fault containment in scalable shared-memory multiprocessors." *Computer Architecture News*, vol. 25, no. 2, pp. 73-84, May 1997.

[Torrellas92] J. Torrellas, A. Gupta, and J. Hennessy. "Characterizing the caching and synchronization performance of a multiprocessor operating system." *SIGPLAN Notices*, vol. 27, no. 9, pp. 162–174, September 1992.

[TracePoint97] TracePoint Technology Group. `http://www.tracepoint.com`.

[Trent95] G. Trent and M. Sake. "WebSTONE: The first generation in HTTP server benchmarking." Unpublished white paper, February 1995. `http://www.sgi.com/Products/WebFORCE/WebStone/paper.html`

[Vashaw92] B. Vashaw. "Address trace collection and trace driven simulation of bus-based, shared-memory multiprocessors." Ph.D. thesis, Carnegie Mellon University, 1992.

[Veenstra94] J. Veenstra. "MINT: a front end for efficient simulation of shared-memory multiprocessors." In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp 201-207, January 1994.

[Veenstra97] J. Veenstra. Personal communication. October 1997.

[Verghese97] B. Verghese, A. Gupta, and M. Rosenblum. "Performance isolation and resource sharing on shared-memory multiprocessors." Technical Report: CSL-TR-97-730, Computer Systems Laboratory, Stanford University, Stanford, CA, July 1997.

[Wall87] D. Wall and M. Powell. "The Mahler experience: using an intermediate language as the machine description." In *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 100-104, October 1987.

[Welbon94] E. Welbon, C. Chan-Nui, D Shippy, and D. Hicks. "POWER2 performance monitor." *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, IBM Corporation, SA23-2737, pp. 55-63, 1994.

[Wilson94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam and J. Hennessy. "SUIF: An infrastructure for research on parallelizing and optimizing compilers." *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31-37, December 1994.

[Wilson96] K. Wilson, K. Olukotun, and M. Rosenblum. "Increasing cache port efficiency for dynamic superscalar microprocessors." *Computer Architecture News*, vol. 24, no. 2, pp. 147-157, May 1996.

[Witchel96] E. Witchel and M. Rosenblum. "Embra: fast and flexible machine simulation." *Performance Evaluation Review*, vol. 24, no. 1, pp. 68-79, May 1996.

[Woo95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. "The SPLASH-2 programs: Characterization and methodological considerations." In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[Zagha96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. "Performance analysis using the MIPS R10000 performance counters." In *Proceedings of Supercomputing '96*, November 1996.

[Zeus97] Zeus Server v1.0. `http://www.zeus.co.uk/products/zeus1`.