Full-System Critical-Path Analysis and Performance Prediction

by

Ali Ghassan Saidi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctorate of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:

Professor Trevor N. Mudge, Co-Chair
Associate Professor Steven K. Reinhardt, Co-Chair
Associate Professor Scott Mahlke
Associate Professor Dennis M. Sylvester

© Ali Ghassan Saidi 2009
All Rights Reserved

To my family: for their continuous love and support.

ACKNOWLEDGEMENTS

This dissertation marks the end of a chapter of my life. My journey through school has been aided by many people over the years and I'm thankful for all their help and support.

First, I must thank my family. They have always encouraged me to learn and have been supportive in every way they could be. My mother, Raghda Saidi Henthorne, my father Ghassan Saidi, my step-dad Michael Henthorne, and my sisters, Rania Combs and Rola Hart have continuously provided love, support, and advice. Additionally, I would like to thank Don and Rita Myntti, who I have known since the time I was a small child, and who have never once failed to call me on my birthday.

Next, I have to thank Karen Smid. She provided tremendious love and encouragement while we both finished our dissertations, as well as the occasional bit of last minute editing.

My advisors, Steve Reinhardt and Trevor Mudge, along with the other Michigan faculty, provided continuous guidance throughout the whole process. Steve is a wonderful mentor, and one of the smartest people I know. His calm and rigorous approach to solving any problem taught me to step back and look at the entire issue, instead of fruitlessly focusing on the minutia. When I was particularly stuck on some issue, I could always count on Trevor to pull some inspiration out of thin air and his meetings always broke up the day with laughs and entertainment.

The other graduate students from Steve's group at Michigan were also of great help to me: Nathan Binkert, Lisa Hsu, Ron Dreslinksi, and Kevin Lim. In particular, I must thank Nate and Lisa not only for their insight and assistance over the years, but also for being great friends. nice

During my time in Michigan, I've met some other wonderful people who have become close friends. They have provided great support and laughter that broke up the days into more managable pieces: Ashley Bangert, Chris Kiekintveld, Chris McCleary, Ed Argalas, Ellen Hamilton, Lee Newman, Michael Abowd, and Michael Zeidler. Lastly, Olga Kornievskaia relentlessly encouraged me to stay in shape, and Kate Cappell, Diane Bouis and Astrid Tuin made sure that the last six months—without Karen and while writing the bulk of my dissertation—was as painless as possible.

Contents

DEDICATION ii

ACKNOWLEDGEMENTS iii

List of Figures viii

List of Tables x

CHAPTER

1 Introduction 1
- 1.1 Complex Workloads 2
- 1.2 Simulation 3
- 1.3 Critical Path Analysis 4
- 1.4 High Speed Networks 5
- 1.5 Thesis Statement 6
- 1.6 Contributions 7
- 1.7 Organization 8

2 Network Background 9
- 2.1 Network 9
- 2.2 Network Interfaces 11
- 2.3 Drivers 13
- 2.4 Device 14
- 2.5 IP 14
- 2.6 UDP 15
- 2.7 TCP 15
- 2.8 Sockets 17
  - 2.8.1 User-to-kernel copy 17

2.9 Applications 18
2.9.1 Micro-benchmarks 18
2.9.2 Web server 18
2.10 Summary 19

3 Simulator Background and Validation 20
3.1 Introduction 21
3.2 The M5 Simulator 23
3.2.1 Processor Models 23
3.2.2 Memory and I/O System 24
3.2.3 Ethernet Interface 24
3.3 Benchmarks 25
3.3.1 Memory Microbenchmarks 25
3.3.2 Netperf 26
3.3.3 SPEC WEB99 26
3.4 Methodology 27
3.4.1 Metrics 27
3.4.2 Simulated System 28
3.4.3 Real System 28
3.5 Comparison to M5 29
3.5.1 Memory Latency 29
3.5.2 Network Benchmarks 31
3.6 Sensitivity of Results 34
3.6.1 Issue Window 35
3.6.2 Issue Width 36
3.6.3 Branch Mispredict Penalty 37
3.6.4 Acknowledging Writes 37
3.6.5 Faster Busses 37
3.6.6 TLB Miss Penalty 38
3.6.7 Spatial Stability 38
3.7 Related Work 39
3.8 Conclusion 39

4 Creating an End-to-End Dependence Graph 41
4.1 Basic Technique 41
4.1.1 Explicit State Machines 42
4.1.2 Implicit State Machines 43

4.1.3 State Machine Interactions . . . 44
4.1.4 Critical-Path Analysis . . . 44
4.2 Queue-based State Machine Interactions . . . 45
4.3 Tracking Multiple Connections . . . 47
4.4 Refining Annotations . . . 48
4.5 Implementation . . . 49
4.5.1 Simulator . . . 49
4.5.2 Annotations . . . 50
4.5.3 Annotation Perturbation . . . 56
4.6 Conclusion . . . 57

5 Locating Bottlenecks and Analyzing Data . . . 58
5.1 Processing Data . . . 58
5.1.1 Time and Space Complexity . . . 58
5.1.2 Limiting Memory Usage . . . 59
5.1.3 Analysis Internals . . . 60
5.1.4 Constructing the Graph . . . 63
5.1.5 Tracking Multiple Connections . . . 68
5.2 Critical Path . . . 69
5.3 Most Critical States . . . 69
5.4 Resource Dependence Loops . . . 71
5.5 Predicting Speed-ups . . . 72
5.6 Connection Trace . . . 73
5.7 State Machine Information . . . 74
5.8 Queue statistics . . . 74
5.9 Compact Graph Output . . . 76
5.9.1 High-level Interaction . . . 76
5.9.2 Global State Machine Graph . . . 76
5.9.3 Single State Machine Output . . . 79
5.10 Conclusion . . . 79

6 Application of Analysis . . . 81
6.1 Simulation and Methodology . . . 81
6.2 Streaming Benchmark Analysis . . . 82
6.2.1 UDP Streaming Benchmark . . . 82
6.2.2 TCP Streaming Benchmark . . . 86
6.3 Web Server Analysis . . . 90

6.4 Analyzing a Copy Engine 92
6.4.1 Copy Engine Overview 93
6.4.2 Performance Evaluation 94
6.4.3 Kernel Bug 96

7 Related Work 97
7.1 Critical Path Analysis 97
7.2 Profilers 98
7.3 Other Related Work 99

8 Conclusion and Future work 101
8.1 Future Work 102
8.1.1 Automation 102
8.1.2 Real Systems 102
8.1.3 Different Systems 103

Bibliography 104

List of Figures

Figure

2.1 Network stack transmit and receive paths 10

2.2 Example transmit and receive descriptor rings 11

3.1 mem_lat_read: Memory Latency with Stride 8 29

3.2 mem_lat_read: Memory Latency with Stride 64 30

3.3 mem_lat_read: Memory Latency with Stride 8k 31

3.4 mem_lat_read: Memory Latency Stability with Stride 8k 32

3.5 Absolute Bandwidth (Mbps) for benchmarks 33

3.6 Relative Bandwidth (Mbps) for benchmarks 34

3.7 Processor Utilization over Benchmarks 35

3.8 Sensitivity of results after varying parameters 36

4.1 Conversion of a state machine 42

4.2 Example of queue dependencies 45

4.3 Example state machines: a) simplified IP stack state machine; b) simplified driver state machine. 47

4.4 Example dependence graph showing the interaction of the two state machines of the previous figure. 47

5.1 Illustration of a graph construction. 59

5.2 Example of critical path analysis. 70

5.3 Example of most critical analysis. 71

5.4 Illustration of a resource dependence loop. 71

5.5 Example of resource dependence loop analysis. 72

5.6 Example of connection tracing analysis. 73

5.7 Example of state machine information. 74

5.8 Example of queue statistics output. 75

5.9 Graph of state machines and queues in Linux TCP/IP stack from NIC to application. 76
5.10 Example global state machine graph generated by the analysis program. . . 78
5.11 Example single state machine graph generated by the analysis program. . . 80
6.1 NIC Transmit State Machine – Global bottleneck graph 83
6.2 Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 1. Dotted line is bandwidth produced by initial configuration. 87
6.3 Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 2. Dotted line is bandwidth produced by initial configuration. 89
6.4 Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 3. Dotted line is bandwidth produced by initial configuration. 90
6.5 Actual bandwidth and predictions for four connections on web server. . . . 91
6.6 Evaluated copy engine configurations 94

List of Tables

Table

3.1 Description of CPU Utilization Categories. 27

6.1 Most critical states Linux 2.6.13 w/netfilter 85

6.2 Next most critical states Linux 2.6.13 w/netfilter 85

Chapter 1

Introduction

System optimization—whether tuning an existing platform or designing a future one—requires knowledge of the current system's bottlenecks. Because the performance of modern networked systems is often determined by the complex interplay among application and operating system software, network interface hardware, and network protocol behavior, identifying the source of performance problems is a painstaking and error-prone process [1, 2, 3]. When dealing with interacting and concurrently operating components spanning multiple layers of software and hardware, conventional tools such as software profilers are inadequate for pinpointing true system-level bottlenecks.

This challenge is becoming increasingly relevant for architects as computing becomes more network-centric and as integration expands CPU design beyond microarchitecture to a platform-level effort. Although full-system simulation is becoming more common, understanding the contribution of hardware, operating system, and application layers to a simulated machine's performance is very challenging. What little comprehension can be achieved is typically gained through tedious trial-and-error cycles of forming hypotheses based on aggregate statistics, and then proving or disproving those hypotheses through additional time-consuming simulations. When results do not match expectations, identifying the root cause is extraordinarily difficult.

This thesis proposes an end-to-end critical-path analysis methodology that is capable of identifying performance bottlenecks in networked applications across the user/kernel and software/hardware boundaries. In doing so, I leverage the observability of a full-system simulation environment to annotate both software and hardware components within a system. From the annotated simulations, a complete dependence graph of component interactions is constructed. By identifying the waiting states in each component, a user can generate the end-to-end critical path from a transmitting application in one system, through its operating system and network interface, across the network link, and back up to the

receiving application in another system.

Because the methodology records and analyzes the full end-to-end dependence graph, it can be extended beyond identification of the critical path to performance prediction. By instructing my tool to reduce or eliminate the latencies of certain edges while processing the dependence graph, users can generate a predicted dependence graph with a new critical path length; the ratio of the original critical path length to this new one accurately predicts the system-level speedup achievable by optimizing a particular component.

This prediction is accurate even if the critical path changes qualitatively. By reducing delays on the original critical path until a new critical path emerges, users can predict (1) how much a particular component's performance must be improved before it is no longer the bottleneck; (2) what the new bottleneck will be after that optimization; and (3) how much speedup can be achieved before that new bottleneck is reached. This approach can be applied iteratively: Once this secondary critical path is exposed, latencies along that path may be artificially reduced until the tertiary critical path appears, and its length can be used to make further quantitative predictions. Thus, after a single simulation run, my tool can—in minutes—identify the end-to-end performance bottleneck, and in successive runs, predict the quantitative impact of removing that bottleneck and expose the subsequent bottleneck. In contrast, even with a fast and flexible simulation environment, each step of this process requires hours or even days, starting with tedious manual interpretation of statistics to generate a hypothesis about the bottleneck, possibly days of development to prototype an optimization that addresses that bottleneck, and hours of simulation to measure the impact of that optimization—and possibly further iterations on the same bottleneck if the initial hypothesis was incorrect.

This chapter discusses why these workloads are important but difficult to analyze, and what motivatesi my goal to provide better tools to analyze these systems.

1.1 Complex Workloads

Many interesting workloads today are not limited by the CPU, but rather by the interactions among the CPU, memory system, I/O devices, and the complex software that ties all the components together. These non-CPU bound workloads are commonplace, especially in the server domain. Some examples of these workloads include web serving, file serving, and online transaction processing (OLTP). They stress a wide range of hardware, including disks, disk controllers, network interfaces, memory, and caches. Additionally, these workloads tend to use many kernel services stressing multiple layers of software. This change is evident in the benchmarking suites in use on new hardware. The community has shifted

from using only CPU performance tests such as SPEC CPU [4] to more complex workloads testing OLTP [5], web serving [6], and file serving.

Because of the large number of interacting layers—both hardware and software—present in these systems, locating performance bottlenecks is difficult with conventional tools. The various levels of software and hardware overlap operations to operate as efficiently as possible, thus a simple snapshot of the current system is insufficient. While all the layers of software and hardware may be processing units of data, it may not be clear which is causing the delay at a much higher level.

For example, consider the issues Kegel [3] recounts in improving the performance of a web server benchmark. The issues described span all layers of the system, and include: the web server itself (Apache), the system call interface (`poll()` and `select()`), locking present in a large data copy loop, the network stack, the user-configurable parameters within the network stack, the network device driver, the disk scheduler, the process scheduler, and even the client machine making the request. This performance debugging took years of effort by hundreds of people around the world. The experience described by Kegel is not unique. Cantrill [1] relates similar experiences at Sun Microsystems, noting that while the problems tend to originate at high-levels of abstraction, few tools exist to analyze the system at those levels, so simple tools like CPU utilization are typically substituted. While these methods may eventually lead to the original cause, they don't do so quickly.

1.2 Simulation

In this work I turn to simulation to assist in locating performance issues as simulation provides complete observability into the system without perturbation. Simulation allows researchers to assess an idea without the complexity or cost of building real hardware. The researcher is normally guided through his intuition or other ad hoc means to improve a component that is assumed to be under-performing in relation to the rest of the system. This hypothesis is then tested in simulation and the results guide future experiments. When applied to a complex system involving multiple layers of hardware and software, two problems arise: (1) simulation time; and (2) final cause determination.

In any one of these complex systems there are hundreds if not thousands of parameters that can be adjusted. The quantity of parameters can present a large problem because of the inherent slowdown in simulation. It is simply too time consuming to explore the entire available state space of all the parameters. A system that contains only ten binary parameters results in 1024 possible combinations of those parameters. Normally some pruning is done to remove configurations that are uninteresting, however a large number

of interesting configurations can still exist. This coupled with the extreme slowdown of simulation—resulting in at least a 1000x slowdown—means that a immense amount of processing power and time are required to explore the entire state space.

The second problem with this state space exploration approach is that a result can be produced by the simulator that identifies some particular parameter as improving performance. This parameter in many cases is a proxy for a different problem that must be discovered. For example, consider an experiment in which the researcher halves the latency of main memory. In most applications some performance increase will occur, however the reason for the increase is still unclear. Is the improved performance because the I/O devices could write to memory more quickly? because cache misses can be resolved more quickly? because of something else entirely? Halving the latency to memory is not a realistic solution, however presenting the researcher with the actual root cause would allow improvement to be focused on the actual problem at hand, as opposed to a proxy of the real problem.

1.3 Critical Path Analysis

Critical path analysis (CPA) [7] or the critical path method (CPM) is a technique for scheduling a set of activities. It has been used for a variety of tasks from the original goal of project planning to instruction scheduling in a micro-processor. Generally a dependence graph of all the activities is created in which edges describe the amount of time required to do an activity and the graph's structure reflects dependencies inherent in the process.

For example, consider the task of making a bird house. The activities are: get wood, get paint, cut wood, build bird house, and paint. The painting is dependent on having a bird house built and having paint. Building a birdhouse is dependent on having wood cut to do so, which is in turn dependent on having wood. Having paint isn't dependent on anything so it can happen right away, although it may not be advantageous to get it immediately. In fact, as long as paint is available before the bird house is built, the bird house will be completed in the same amount of time.

In this work the activities are normally some high-level task, such as the placement of a packet in a queue or the calculation of a checksum for a packet. The time for an activity to occur isn't estimated for the purposes of planning, but it instead directly observed from the system in question. The observed data is then used for a posteriori analysis. In particular, the observed data can pinpoint which of the many concurrent activities is really the bottleneck and therefore what activity requires improvement.

1.4 High Speed Networks

Locating performance problems is particularly difficult in high-bandwidth network processing [2], in which there is no single bottleneck that can be easily addressed and the problems are not fully understood. Researchers have taken completely different paths [8, 9, 10, 11] in attempting to improve performance without fully understanding the causes of the problem. Performance losses come from the combination of numerous overheads in interactions among the network protocol, software running on the CPU, the memory system, and the network interface controller [12]. For example, consider the case in which the end-to-end bandwidth between some sender and receiver is lower than expected. There are a number of explanations:

- Transmit data is queued in the network interface card (NIC)'s DMA descriptor ring, but the NIC DMA controller cannot process the descriptors quickly enough. Perhaps the NIC cannot fetch the descriptors fast enough or the I/O bandwidth is insufficient to fetch descriptors and packets at the rate the OS is making them available.

- There is data ready to be transmitted in the kernel, but the device driver cannot fill the NIC's DMA descriptors from the kernel I/O buffers quickly enough. Perhaps the number of allocated buffers is insufficient, or the overhead of reclaiming processed buffers is too high.

- The application has requested a transmission, but the kernel's TCP/IP protocol stack has not completed processing the data so the device driver has not received it.

- There is data ready to be transmitted in the kernel, but the TCP protocol is delaying transmission because the receiver has not advertised sufficient buffer space to accept it. In this case, the receiving system is the bottleneck; a similar exercise must be repeated on that system to isolate the bottleneck to the NIC, device driver, kernel, application, or other source.

- There is data ready to be transmitted in the kernel, but the TCP protocol is delaying transmission because the number of outstanding packets has reached the TCP congestion window size.

- The application has not requested a transmission via `write()` or `send()`. In this case, the application is the bottleneck.

Note that these reasons span from hardware through kernel code up to the application on both the sending and receiving systems. Furthermore, because the transmission process

is pipelined, simply observing snapshots of system state is insufficient: at any given point in time, the application, kernel stack, device driver, and NIC may all be actively operating on different data units without any obvious indication of which one is the bottleneck. Finally, each of the reasons listed above is only an intermediate determination, not a final cause. For example, if the application or protocol processing is the bottleneck, further analysis is required to determine which part of the code is problematic, whether the lack of performance is due to limitations in instruction execution or memory bandwidth. I have found that the complexity of this problem generally stymies attempts to tease out bottlenecks using traditional means such as ad hoc analysis of statistics or iterative testing of hypotheses.

1.5 Thesis Statement

The thesis of this work is that critical-path analysis can be applied to complex systems composed of multiple layers of hardware and software such the interacting state machines that govern the systems behavior can be converted into a dependence graph. That dependence graph can be used to locate performance bottlenecks and predict performance when those bottlenecks are removed.

As architects look beyond the CPU to more integrated platform designs, it is difficult for them to locate bottlenecks in proposed hardware before the systems are built. In this environment, real hardware is unavailable, and simulation is the primary performance evaluation tool. However, simulation is slow and the state space the architects are operating in is extraordinarily large. When results do not match expectations, locating the bottleneck component is difficult. This thesis provides a technique to apply critical-path analysis to all the layers within a simulated system. A prerequisite for critical-path analysis is an event dependence graph representing timing constraints. Unfortunately, since there are many interacting components, the task of developing a global dependence graph through ad hoc familiarity with the subject matter rapidly becomes intractable. First, a method for creating such a dependence graph is presented that converts systems of interacting state machines (both explicit state machines in hardware and the state machines that implicitly govern software behavior) into a global dependence graph suitable for performance analysis, including bottleneck identification. This conversion is done with modest effort and without a deep understanding of the components involved. Next, the dependence graph must be analyzed to distill useful information from the many million nodes and edges it contains. Several algorithms are presented that provide condensed information about bottlenecks. Additionally, a mechanism to locate resource constraints is explored. Because the dependence graph

contains all the information about interactions in the system, it can be used to predict performance when the current and subsequent bottlenecks are eliminated. Finally, I show that multiple bottlenecks can be found using my technique and that the performance predictions provided are extremely accurate even when the predicted performance gain is substantial.

1.6 Contributions

This thesis makes the following contributions:

- I describe an automated technique for converting systems of multiple interacting state machines into a global dependence graph suitable for performance analysis, including bottleneck identification.
- I show that the annotations necessary for this analysis can be developed with modest effort and without detailed knowledge of the particular implementation of the components involved; in particular:
  1. state machines implicit in complex software systems, such as the Linux kernel's UDP, TCP, and IP stacks, can be traced with limited manual effort by relying primarily on control-flow tracing and symbol table lookup
  2. the necessary inter-state-machine dependence annotations can be uncovered using iterative refinement
- I demonstrate an implementation of my technique that uses a full-system multi-machine simulator to identify bottlenecks across hardware/software boundaries.
- My technique is able to deal with complex protocols, multiple connections, and multiple processors through its explicit modeling of shared resources such as queues.
- I identify critical-path "loops" as the characteristic signature of a finite-resource bottleneck, and automate the detection of these loops.
- I demonstrate that my critical-path methodology is capable of not only identifying multiple bottlenecks from a single run, but accurately predicting the quantitative benefit of removing those bottlenecks. This methodology eliminates the need to use statistics to form hypotheses about system bottlenecks, and enables designers to predict in minutes results that may take hours or days to prototype and simulate.

1.7 Organization

The remainder of this thesis is organized as follows. First, some background information is presented about networking in Chapter 2 and simulation in Chapter 3. After this background information, related work is discussed in Chapter 7. The technique that enables the creation of an end-to-end dependence graph for complex systems is explored in Chapter 4. With a method to create an end-to-end dependence graph available, the methodology to analyze this graph and produce useful results is presented in Chapter 5. The technique and analysis is applied to several applications, protocols, and systems in Chapter 6. Finally, I conclude and present future work in Chapter 8.

Chapter 2

Network Background

This chapter briefly describes the software and hardware that interact to enable applications to send and receive packets in a network. Networking is becoming an increasingly important part of computing. As mentioned in the introduction, because of the large number of layers that interact, high-speed networking is difficult to analyze. This chapter takes a bottom-up view, starting with the lowest level and describes the abstractions that are built upon each layer, eventually enabling applications to reliably transmit data from one machine to another. The descriptions here are slightly biased toward the Linux network stack, although most of the discussion is generally applicable. A more detailed description of a TCP/IP implementation can be found in [13]. The general unidirectional flow of packets through the networking stack and devices is illustrated in Figure 2.1. However, packets can, and frequently are, sent concurrently from both directions. Each layer is composed of one or more state machines—either hardware or software—that interoperate transfer packets from the sender to the recipient.

2.1 Network

A network can be a simple point-to-point link or a complex series of routers and links as exists on the Internet. In normal operations the packet is delivered some time after it is sent. The link-layer protocol used to exchange packets govern what guarantees the network makes. The most popular protocol, Ethernet, does not guarantee in-order delivery or delivery at all. Packets can be discarded in the network due to congestion or failure and out-of-order packets are possible because of different path lengths and routers automatically adapting to congestion on their links. Other protocols do provide stronger guarantees, but because of their cost, their cost their adoption is extremely limited.

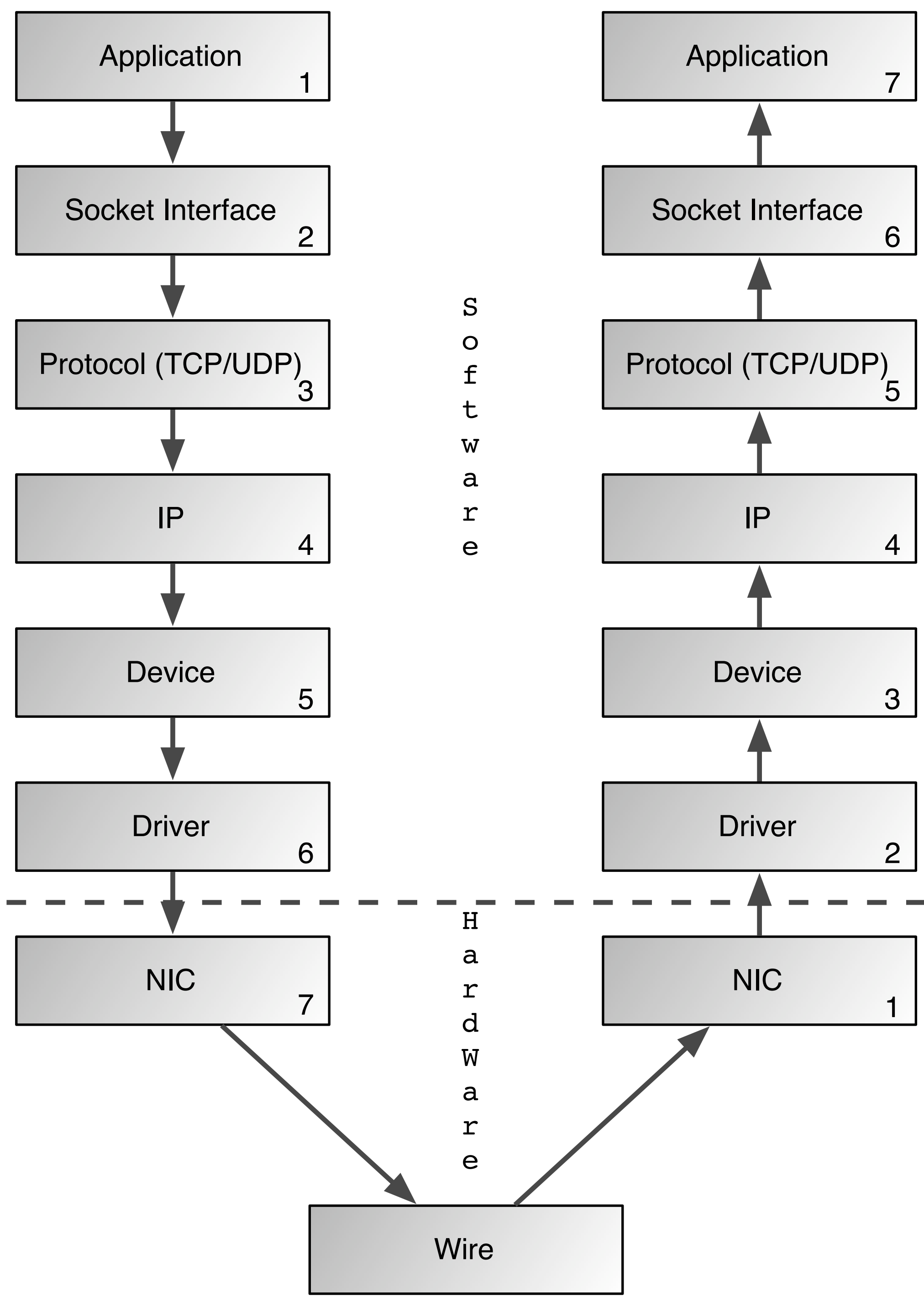
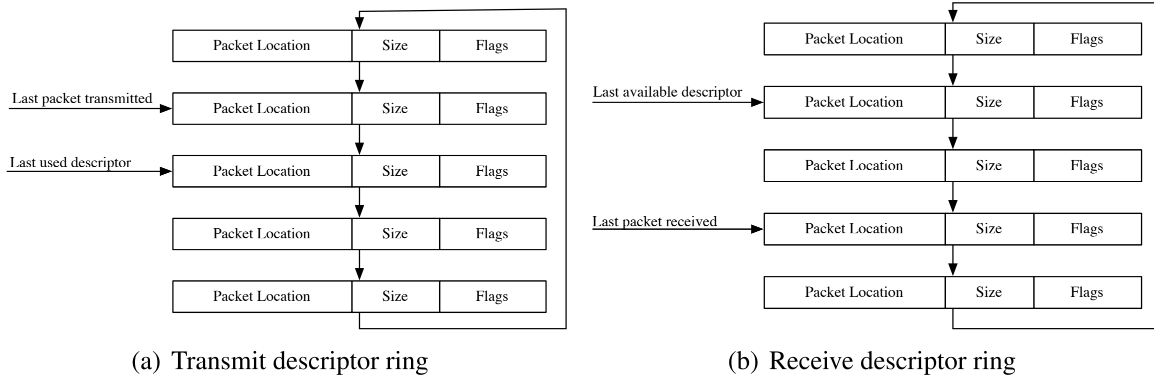


Figure 2.1: Network stack transmit and receive paths



(a) Transmit descriptor ring
(b) Receive descriptor ring

Figure 2.2: Example transmit and receive descriptor rings

2.2 Network Interfaces

Broadly speaking, a Network Interface Controller (NIC) is a physical device that provides connectivity between the network and CPU. In its simplest form, a NIC receives a packet from a CPU and transmits the entire packet to the network adhering to the physical link transmission protocol which, in most cases, is Ethernet. Similarly in the reverse, a NIC can receive a packet from the network link, notify the CPU of is existence, and provide a means for the CPU to retrieve the packet.

Simple NICs operating on low bandwidth links can be interfaced with purely through uncached I/O operations, also known as Programed I/O (PIO). However, as the desired bandwidth and number of CPU clock cycles to access an uncachable memory location increases, this method becomes untenable. Most high performance NICs transfer data through one or more DMA descriptor rings for both the transmit and receive sides of the NIC.

After the NIC has received a packet from the CPU it is placed in an outbound FIFO waiting for the media access controller (MAC) and physical interface (PHY) to send the packet. This buffering exists because the speed at which the NIC receives a packet and the speed at which it must signal a packet onto the network may be different. When a NIC receives a packet from the network it is placed in the inbound FIFO waiting for the NIC to copy it to the appropriate location in memory.

These DMA descriptor rings are located in memory and contain an address of a packet to send, the size of the packet, and some flags that describe precisely what is to be done with the packet. The rings are either a circular buffer or a chain of next pointers (normally wrapping around and creating a ring). The device and driver maintain pointers to entries in the ring and communicate through it.

For example, take the transmit descriptor ring illustrated in figure 2.2(a). In this case,

the NIC maintains the *last packet transmitted* pointer, while the CPU maintains the *last used descriptor* pointer. These pointers can never cross, however both can wrap around the ring. When the driver would like to transmit a packet it places the packet location, size, and flags in the descriptor succeeding the *last used descriptor* and updates the pointer. The NIC can then transmit all the packets between the *last packet transmitted* and the *last used descriptor*. The driver cannot instruct the device to transmit an additional packet if the *last used descriptor* pointer is immediately succeeded by the *last packet transmitted* pointer. In that case the driver must wait for the NIC to send some packets to free up some space in the descriptor ring.

The receive descriptor ring is similar to the transmit side and is illustrated in figure 2.2(b). However, in this case the driver makes many descriptors available for the NIC to populate with packets as they are received from the network. Each descriptor contains an address of an available buffer and the maximum size that buffer can hold. The NIC writes the packet to the location specified by the descriptor and then updates the *last packet received* pointer. If the descriptor ring is full, the *last packet received* and *last available descriptor* point to the same location and the NIC must wait for the driver to provide it with more descriptors. If insufficient buffering exists on the NIC, packets can be lost when no descriptors are available.

The device can either signal to the device driver when action is required (see next section), or the device driver can poll the device at some regular interval. Both polling and interrupts have drawbacks. In the case of polling, the CPU is interrupted at a fixed interval of time if there is work to be done or not. If work needs to be done then the polling mechanism limits the interrupts the CPU experiences to a fixed number over a certain amount of time; however, when there is no work to be done, the polling continues possibly interrupting an otherwise busy or idle CPU. On the other hand, interrupting every time the NIC requires some sort of attention is not realistic either. If a 10 Gbps NIC interrupted the CPU every time it received a packet the interrupt rate could exceed 20M interrupts/sec, which would not leave much time for the CPU to handle other tasks. As such, most NICs include some sort of interrupt moderation scheme. These schemes either limit the interrupt rate to a certain number per second (e.g., 8000), or wait some amount of time after an interrupt would be generated to coalesce several interrupts into one (thus limiting the rate). Finally, a hybrid approach is possible that interrupts during a period of NIC idleness and switches to a polling mode when the NIC is busy.

There are several NIC optimizations that remove some processing and bandwidth requirements from the host CPU. Most high-performance NICs have support for checksum offloading. Checksum offloading calculates and inserts a checksum into a packet. This is

reasonably simple for the NIC to do, since it is going to transmit every byte in the packet, however for a CPU to calculate the checksum it must sum every word in the packet. Another optimization would allow a scatter-gather DMA to assemble a packet. In this case, various portions of a complete packet can be in different memory locations. The NIC can fuse together the data provided by multiple descriptors and send them as one unit. This is especially useful when different portions of the data come from different layers of software (e.g. the application and protocol layers) and prevents additional copying to ensure that all packets occupy continuous bytes in memory.

2.3 Drivers

The device driver is piece of code that manages the device presenting it with packets that are ready to be transmitted. The particular interface to do this is device specific, however the majority of NICs employ a DMA descriptor scheme similar to the one outlined in the previous section. The driver's responsibility is to manage the descriptor rings.

On the transmit side, the driver places packets into free entries in the ring (or allocates a new descriptor and links it into the ring with the next pointer). The driver inserts the address of the packet (or a portion of the packet), size, and any flags. These flags include an end-of-packet marker and various other flags normally dealing with the generation of checksums and if interrupts should be generated when the transmission of this packet is complete. Upon inserting packets into the ring, the device driver updates the ring pointer which informs the NIC that it has more work to do. The NIC moves the pointer it manages when it completes the transmission of the packet.

The device interrupts the CPU (or waits for the polling interval) when it needs attention. For the transmission side of the NIC, an interrupt generally indicates the transmission of packets, and the driver can then clean up the descriptors associated with those packets. This includes resetting the descriptor for future use, and signaling that the NIC is done transferring the packet to the rest of the kernel. Depending on the protocol being used, this may or may not free the data associated with the packet (if the protocol is reliable it may have to be resent).

On the receive side of the NIC, the driver performs actions similar to those on the transmit side. When the NIC has DMAed a packet to memory the driver hands the packet to the network stack for processing. The driver normally checks the descriptor to verify that the network checksum was correct and may do other things such as copying the packet header for small packets so the network protocol headers can be accessed quickly. The used descriptor is cleaned by allocating a new buffer to store a new packet and populating

the descriptor with the new buffer's address and size along with clearing various status bits.

2.4 Device

The device layer exists as an indirection layer as well as buffering for both the receive and transmit sides of the driver. This layer unifies the interface of the driver and the protocol. On the transmit side, if the driver is currently unable to accept a packet because the device is busy, doesn't have enough free descriptors, or is otherwise stopped, the packet is buffered in the device layer. When the driver signals that it is again able to accept packets, packets queued in the device layer are handed to the driver. Similarly, on the receive side the device layer queues received packets until they are processed by the network stack. This queueing allows the device driver to run for a short time and the scheduler to pick an appropriate time for the network stack to process the incoming packets.

2.5 IP

The next protocol in the stack is almost always the Internet Protocol(IP) [14]. The IP layer consists of a header that is attached to the packet that contains the source and destination address, the encapsulated protocol, a checksum, and fragmentation information. The source and destination specify the end points of the connection, and are used to calculate the next hop along the route from source to destination. This next hop may be the destination itself, or an intermediate router to the destination. The encapsulated protocol identifies which protocol is being sent an IP packet; this is normally TCP or UDP, but other types are also utilized. The IP header contains a checksum of the header to verify that the header arrived correctly. This checksum is frequently calculated and inserted by the NIC. If the packet that is being transmitted is larger than the network's maximum transmission unit (MTU), the IP layer fragments the packet into multiple pieces. This fragmentation is reasonably inexpensive to do; however, it tends to be much more costly to undo. Protocols such as TCP that guarantee delivery are rarely fragmented because of the cost of reassembly later. Finally, the IP layer looks up the destination address in a list of devices connected to the system and sends the packet to the appropriate device.

When the IP layer receives a packet it verifies that the checksum is correct. Normally, this verification is done by the NIC, but if the NIC was unable to verify the checksum, it would be calculated here. The IP layer then does any reassembly that is required, determines what protocol the IP header encapsulates, and presents the packet to that layer for

further processing.

2.6 UDP

The User Datagram Protocol (UDP) [15] provides very few transport guarantees. UDP gives an application access to the IP protocol, and provides for data multiplexing by adding source and destination port numbers, and some level of integrity checking with a checksum that includes the data. However, the protocol does not provide for guaranteed in-order delivery and thus leaves end-to-end reliability to the application. This is desirable in some cases, either because the reliability and ordering constrains need to be handled in a special way, or the real-time nature of the data means that any re-transmission or lost or re-ordered packets would be stale. For example, consider the live broadcast of video on the Internet. If one packet is missing in the stream, by the time that missing packet is noticed, re-requested, and received, it is extremely likely that the viewer will have already noticed a disruption and continued watching. The viewer would be disrupted more by the video stopping to wait for the packet containing the missing video frame to arrive.

Since the protocol provides such minimal guarantees, the protocol does very little and just acts as a bridge between the IP layer and the sockets layer.

2.7 TCP

The Transmission Control Protocol (TCP) [16] provides, like UDP, for data-multiplexing via port numbers and checksums the data payload to provide some integrity guarantees. However, unlike UDP, TCP provides for reliable in-order delivery. It is a significantly more complex protocol that has to deal with lossy links, congestion, and re-ordered packets. When TCP packets are received, the corresponding bytes are acknowledged in a packet flowing in the opposite direction. This packet may contain data flowing in the opposite direction as well, or may simply be an acknowledgment. TCP maintains several sliding windows to ensure adequate buffering is available and received bytes are actively acknowledged by the receiving side. It contains a buffer notification scheme to limit the data the sender can transmit to the available buffering on the receiver and contains a congestion avoidance algorithm to limit the rate at which data can be transmitted.

The TCP receiver's sliding window advertises to the sender how much space is currently available for buffering on the receiver. The protocol forbids the sender from having more outstanding unacknowledged bytes in the network than the receive window adver-

tises. The sender itself also has a window that limits the amount of outstanding data in the network. Since TCP is a reliable protocol, each transmitted byte must be buffered locally in case a retransmission is required. Since this buffering requires kernel memory, it cannot grow without bounds. Thus, each connection has a limited amount of memory for local buffering. The amount of required buffering is equal to the bandwidth-delay product. The desired bandwidth times the round-trip delay is the required size for the sender window and the receiver window. The TCP protocol also contains a congestion window that also limits the sender. This window initially limits the sender through the slow-start algorithm that seeks to match the rate that packets are injected into the network with the rate that responses are received. This limits congestion on the network by not completely saturating a slow link on the path from source to destination. If the receive, sending, and congestion windows are all very large, and the path from source to destination traverses a large network, the network may discard some packets. When this occurs the congestion window size is reduced dramatically to slow down the sender and allow the congestion to pass. Normally, the congestion window is then increased at a slower rate. There are additional requirements that may cause the TCP layer to delay sending out some available data. One is TCP attempting to batch together several smaller transmissions into one larger packet (Nagle's algorithm) [17].

When all the window and congestion algorithm requirements are met, a packet has a TCP header applied and is then sent to the IP layer for continued processing. Just like in the IP header, the TCP header contains a checksum that can be expensive to compute. In many cases the NIC can insert this checksum, so the operating system need not be burdened with this calculation.

When a packet is received by the TCP layer a variety of checks must be completed. The TCP receive code verifies that the packets were received in-order; if a segment is not in the correct order it is queued separately waiting for the intervening packets. If the packets are received in order without any special flags or options then the packets can be handled on a fast path as first noted by Van Jacobson [18]. For every two packets that are received normally an acknowledgment is generated. The first packet causes a timer to begin to count down, and if another packet isn't received then the timer fires and generates an acknowledgment. If the timer is already running, the timer is canceled and an acknowledgment is sent out immediately.

2.8 Sockets

The socket buffer level is an intermediate level of code that seeks to unify multiple possible transfer methods behind a common interface. This interface can be accessed through the `send()` and `recv()` system calls or through normally `read()` and `write()` system calls. Socket options can choose between blocking and non-blocking versions of these calls. The blocking versions wait until the requested action has occurred, while the non-blocking ones return immediately and leave the application to re-request the action. For example, in the case of `send()` or `write()`, if the sender window is full, the system call will normally block until enough space is available in the window before returning to the application. In the case of a non-blocking call, the call will return immediately if it will block with an error status. The application will need to try to send the data again later. Once the system call returns to the application, the buffer that was given to the application is again owned by the application, thus all the data must have been copied out of it prior to the system call return. Similarly, when `read()` or `recv()` are called the system call will block if no data is available in the socket; if data is available then it will be copied into the user buffer and returned. Just like the transmission, if the call is non-blocking then the system call will return an error code if no data was available in the socket.

2.8.1 User-to-kernel copy

The copies to or from user space are very expensive operations. User-to-kernel copies can be cache hits since the application normally just prepared data to send. However, kernel-to-user copies are normally cache misses. When the NIC DMAs bytes to memory those blocks in the cache are invalidated. This forces them to be a miss (unless DMAs are injected into the cache via some mechanism). With many of these copies exceeding 1KiB and sometimes being as large as 64KiB, the time to copy 64KiB is rather extreme as it results in many cache misses. Furthermore, since the 64KiB normally consists of many smaller packets, prefetching does not help as much as it potentially could. Some optimizations are done for the copy in the TCP fast path case, if an application is blocked waiting for data the data can be copied immediately to the user buffer rather than first being enqueued in the socket layer.

2.9 Applications

The transmission or receipt of some data begins at the application level. Here the application must prepare the data it wishes to send and then inform the kernel about its intent and the desired destination. It does this with a system call. This is conceptually similar to a function call, however instead of calling more application code, code inside the kernel with a higher privilege level is executed to process the request. If the system call blocks, the application is de-scheduled and will continue re-scheduled in the future when the system call can be completely satisfied.

While it is possible to be more efficient with non-blocking I/O calls, many applications use blocking I/O because of the added complexity the non-blocking system calls entail. Many applications use additional threads that block, as opposed to a single thread that does not.

2.9.1 Micro-benchmarks

Network streaming micro-benchmarks test the throughput available from one application to another (normally across multiple systems). They are rather simple and seek to transmit packets as fast as possible on one end, while receiving them as quickly as possible on the other end. They normally consist of a single stream, and thus are generally limited to a single CPU. Since the application does nothing but transmit or receive data, micro-benchmarks generally use blocking I/O calls.

2.9.2 Web server

A web server is a large application that serves web pages to numerous clients simultaneously. The architecture of the web server varies, with some using a single (or extremely small number) of processes using non-blocking I/O [19], while others may use a multitude of processes and/or threads coupled with blocking I/O [20]. A web server can serve a combination of static and dynamic content. The static content is pre-generated and just needs to be sent to the requester. Dynamic content needs to be generated per user and can vary from minimal processing of a template, to complete data generation. The bulk of the data transmitted by a web server generally is static content.

Sendfile

Normally, when an application would like to transmit some data, it must read or generate that data into one of its buffers before transmitting it. In the case of a web server

this activity can be very costly. The data must first be copied from the kernel's buffer cache (where files read off disk are cached for multiple accesses) to the user application and then again from the user application to the kernel for transmission. This action contains two copies, doubling one of the most time-consuming parts of data transmission. The `sendfile()` system call allows for an application to bypass the two copies by instructing the kernel to send (part of) a file directly to the desired recipient. This replaces two copies with zero, since the buffer cache will keep a copy of the file to be transmitted until the transmission is acknowledged. The web server still needs copies for HTTP headers and dynamic content; however, with `sendfile()`, sending static content is much cheaper.

2.10 Summary

This chapter has illustrated some of the many layers, spanning both the hardware/software and user/kernel boundaries, that interact in network workloads. Each new layer builds on the abstractions presented by the previous layer to enable greater functionality, however as the functionality increases the difficult in locating problems increases as well. Understanding how one high level event effects the system as a whole is a difficult and error prone task.

Chapter 3

Simulator Background and Validation

There are two reasons that simulation is used in this work. First, it allows ideas to be analyzed before hardware for them is built. This is particularly useful as integration pushes CPU design from the microarchitecture into a system-level design. Second, simulation provides observability into the system without perturbation. For the results of the analysis performed to be useful, performance accuracy is a critical, yet it is often a neglected aspect of architectural performance simulators. One approach to evaluating performance accuracy is to attempt to reproduce observed performance results from a real machine. In this chapter, I will model the performance of a Compaq Alpha XP1000 workstation using the M5 full-system simulator version 1.0. (Since this project was completed, numerous new versions of the simulator have been released). There are two novel aspects to this work: a) complex TCP/IP networking workloads are simulated and network bandwidth is used as the primary performance metric; b) performance accuracy is achieved without extremely precise modeling of the reference hardware. Unlike conventional CPU-intensive applications, these networking workloads spend most of their time in the operating system kernel and include significant interactions with platform hardware such as the interrupt controller and network interface device. Instead of exactly modeling the hardware of interest, simple generic component models are used and tuned to achieve the appropriate bandwidth and latencies.

Overall, a high level of accuracy was achieved even with M5's relatively imprecise models, matching the bandwidth of the real system within 11% both in absolute terms and comparing the relative performance of a 500MHz system against the performance of a 667MHz system. Random delays were injected into the ethernet and memory system and resulted in very little spatial variability. Additionally, I performed a sensitivity study on the impact of incorrectly modeling various architectural parameters. I find the change tends to be minor, affecting the relative performance between the two systems by less than 10%.

However, it is interesting to note that normally neglected parameters such as the TLB miss trap cost tend to affect the results as much as parameters that are often considered more important such as the instruction window size.

3.1 Introduction

The computer architecture community makes wide use of simulation to evaluate new ideas. Simulation allows ideas to be tested in a fraction of the time and without the cost and risk associated with building a physical implementation. However, because of the slowdown inherent to simulation, only a small amount of time can be simulated in detail. To provide meaningful results, execution-driven architectural simulators must both be functionally correct and model performance accurately. Functional correctness, though often challenging to achieve, is typically straightforward to test. In many cases, a lack of functional correctness has catastrophic consequences that cannot be ignored. Performance accuracy, on the other hand, is much harder to verify and much easier to neglect. As a result, it generally gets short shrift from the research community. This situation is ironic: Given that the primary output of these simulators is performance data rather than simulated program output, performance accuracy is at least as important as functional correctness, if not more so.

One of the key obstacles to validating performance accuracy is that accuracy measurement requires a reference implementation with which the simulator's performance results can be compared. Because the primary purpose of simulation is to model designs that have not been (and may never be) implemented, this reference usually does not exist. Some of the few simulator validation studies in the literature came from situations in which the simulator was used in the design of a system that was afterwards implemented in hardware, at which point the designers could go back and retroactively measure the accuracy of their simulators [21, 22]. While valuable for the insights provided, these studies do not provide validation before design decisions are made, when it is needed most.

Another approach to performance validation is to configure a parameterizable simulator to model an existing system and evaluate its accuracy in doing so [23, 24]. Although this process does not fully validate the simulator's accuracy in modeling all the various configurations of interest, it identifies common-mode errors and provides a much higher degree of confidence than having no validation whatsoever. Unfortunately, modern computer systems are extremely complex, and modeling the myriad subtleties of a particular hardware implementation is a painstaking effort. Capturing these subtleties may required correlation of the simulator's absolute performance with that of the actual reference hardware. How-

ever, to the extent that these details are orthogonal to the architectural features under study, more approximate models would suffice to provide accurate relative performance measurements. Developing performance models that incorporate numerous potentially irrelevant details may not be the most productive use of a researcher's time.

The difficulty of performance validation increases as researchers attempt to investigate more complex applications. For example, Desikan et al. [24] were able to model the performance of CPU-bound microbenchmarks on an Alpha 21264 with an accuracy of 2%, but complexities in the memory system (including the impact of virtual-to-physical page mappings and refresh timing) caused their modeling error to average 18% on the SPEC CPU2000 macrobenchmarks.

This chapter describes my experience in validating the performance accuracy of the M5 full-system simulator for TCP/IP-based network-intensive workloads. My efforts differ from previous work in this area in two key aspects. First, complex workloads that are both OS- and I/O-intensive are used. In contrast, earlier simulation validation studies used application-only simulation [23, 24] or used workloads that did not stress the OS or perform significant I/O [21, 22]. Second, while the configuration of the M5 simulator is adjusted to model the reference machine (a Compaq Alpha XP1000), I do not strive to model that system precisely. One of my goals is to determine how accurately I can model the overall behavior of a complex system using appropriately tuned but relatively generic component models. Avoiding irrelevant modeling details both saves time and increases confidence that the set of model parameters exposed in M5 captures the most important behavioral characteristics.

Two Alpha 21264-based Compaq XP1000 systems were used as reference platforms, with CPUs running at 500 and 667 MHz. The performance of the M5 simulator was compared to the two real systems to assess both the absolute and relative performance accuracy of the simulator, using network bandwidth as the primary metric. After correcting several inaccuracies in the model, I narrowed the discrepancy between it and the actual hardware to less than 11%. I also evaluated simulations on their ability to model the relative time spent in different portions of the software (application, protocol stack, device driver, etc.). I found that the simulated and actual CPU utilization numbers are strongly correlated.

In addition to the validation, a sensitivity analysis was performed on the modified simulator to determine which parameters have the greatest impact on simulated system behavior. In doing so, I evaluated the spatial stability of these workloads and found that parameter changes affect the achieved bandwidth by an order of magnitude that is greater than inserting randomness into the memory or ethernet latencies. Identifying components and parameters that do not contribute significantly to overall accuracy is important, as the

architecture community can save development time by not writing or validating detailed models for these components, and can save simulation time by not executing those detailed models. I found that incorrectly modeling a single parameter does not greatly affect the achieved bandwidth. However, some often neglected parameters such as the TLB miss trap cost are as important as parameters that are normally considered architecturally important.

3.2 The M5 Simulator

The primary goal of the M5 simulator [25] is to enable research in end-system architectures for high-bandwidth TCP/IP networking. TCP/IP network protocol and device interface code resides in the operating system kernel, so TCP-intensive workloads spend much of their time there. As a result, full-system simulation is a necessity. This research requires additional capabilities beyond previously existing full-system simulators [26, 27, 28], such as a detailed and accurate model of the I/O subsystem, including the network interface controller (NIC), and the ability to simulate multiple networked systems in a controlled and timing-accurate fashion.

A key goal of M5 is modularity. All simulated components are encapsulated as C++ objects with common interfaces for instantiation, configuration, and checkpointing. The following subsections describe the categories of component models most relevant to this study: processors, memory and I/O systems, and the network interface.

3.2.1 Processor Models

M5 includes two processor models used in this study: a simple functional model used primarily for fast-forwarding and cache warmup, and a detailed out-of-order model used for collecting performance measurements. (M5 version 1.0 uses a model derived from Simplescalar's `sim-outorder` [29], but has been almost completely rewritten.) In addition to standard out-of-order execution modeling, the detailed processor model includes the timing impact of memory barriers, write barriers, and uncached memory accesses.

These processor models functionally emulate an Alpha 21164 (EV5). The simulator executes actual Alpha PALcode to handle booting, interrupts, and traps. The 21164 was originally chosen because it allowed the M5 developers to use SimOS/Alpha [26] as a reference platform during initial development of full-system support. Although M5 implements the 21164 control registers and executes 21164 PALcode, both the OS and application code see the processor as an Alpha 21264, with all associated 21264 ISA extensions [30].

To boot Linux, M5 functionally models a Compaq Tsunami platform [31], of which

the XP1000 is an example. M5's implementation includes the chipset and corresponding devices such as a real-time clock and serial ports. These devices are emulated with enough fidelity to boot an unmodified Linux 2.6 series kernel.

3.2.2 Memory and I/O System

The busses and chips connecting the CPU, memory, and I/O devices of the system are key factors in the performance of that system. With M5 the memory and I/O system are built out of memory and device models interconnected by busses and bus bridges.

The bus model is a simple split-transaction broadcast channel of configurable width and frequency. When used as a point-to-point interconnect, as in some places in the M5 Tsunami model, performance is optimistic, as it provides full bandwidth at half duplex (i.e., in each direction, though not concurrently), while the real link provides half the total bandwidth but in full duplex.

The bus bridge model joins two busses and has configurable delay and buffering capacity. Bridges are used to connect busses with different bandwidths and to model the latency of chip crossings. These bridges also have the capability to acknowledge writes back to the requester before passing them on, which is needed to accurately model the timing of the Compaq Tsunami platform (see Section 3.6.4).

Using these simple components, a user can assemble an entire memory hierarchy from a CPU through multiple levels of cache to DRAM or I/O devices as desired. I/O DMA transactions are kept coherent by having the cache hierarchy snoop them. DMA reads get modified data from the cache if present, and DMA writes invalidate any cached copies. Since a uniprocessor system is being modeled, a more complex coherence scheme is not required.

3.2.3 Ethernet Interface

The NIC model used in the simulator was a National Semiconductor DP83820 network interface controller (NIC), as it was the only commercial gigabit Ethernet interface for which documentation waspublicly available at the time of this validation. This card supports full-duplex transmission at 1Gb/s and is comprised of three units: the BIU, MAC, and PHY.

The Bus Interface Unit (BIU) connects to the PCI Bus on one side and to the Media Access Controller (MAC) on the other. This unit holds configuration and status registers that can be accessed by the device driver through programmed I/O, as well as a DMA engine, to transfer packets to and from main memory. It participates in the memory model

as a first class device, arbitrating for the PCI bus and transferring data like any other device in the memory hierarchy.

Data received from the network or on its way to be transmitted on the network is sent to the MAC where it is buffered. In the case of the DP83820 the transmit and receive buffers are 4kB and 32kB respectively. As the media is available the data in the MAC FIFOs is transmitted to the NIC's physical interface (PHY). The PHY provides a connection between the MAC FIFOs and the physical link, modulating and demodulating signals on the link (copper or fiber). Since the PHY contains no buffering and represents minimal delay, it is not modeled explicitly.

The Ethernet link modeled in M5 was a lossless full-duplex link of configurable bandwidth and delay. The time on the link was calculated by dividing the packet size by the link bandwidth. If the result was less than the wire delay, only a single packet was allowed on link at a time. If the delay was large, then it was possible for the link to have multiple packets in flight. If insufficient buffer space was available at the receiver when a packet exits the link, the packet was dropped.

The DP83820 was used in many actual gigabit Ethernet cards including the NetGear GA622T, D-Link DGE500T, and SMC 9462TX. The NetGear GA622T is used for the experiments with real hardware. The DP83820 has a bug which requires it to DMA to a 8-byte aligned address. A NIC can normally DMA to an arbitrarily aligned address so that the payload within the Ethernet packet can be aligned at the expense of unaligning the Ethernet header. This bug does not exist in the model simulated in M5. However, to be true to the actual card, I removed that fix. As a result, the experiments take many unaligned access traps on both the real and simulated Alpha hardware.

3.3 Benchmarks

Several workloads were used to compare the performance of the simulated systems and the two Alpha XP1000s. Two memory microbenchmarks, the `mem_lat_rd` utility from LMBench [32] and a custom Linux kernel module, provided detailed memory timing information. To measure networking performance, the netperf [33] microbenchmark is used along with a modified version of SPEC WEB99 [6].

3.3.1 Memory Microbenchmarks

The memory system and I/O device delays in M5 were calibrated with two different microbenchmarks. The first is a small custom Linux kernel module that calculates the

latency to a given uncachable memory location. It accomplishes this by using the Alpha `rpcc` instruction to time the execution of loads and stores to the address in question. With this kernel module on a real machine I was able to discern the read and write latency to chipset registers and I/O device registers. These measured delays were used to calibrate the bus-bridge timing in the M5 simulator appropriately.

The `mem_lat_rd` benchmark from LMBench helped me calibrate and verify the DRAM and cache timing parameters used in the system. This tool allows the user to select a stride, a region, and a size of memory for testing. The benchmark then builds a linked list in the memory region with each memory location holding a pointer to a location a stride away. The pointer dependence forces each load to complete before the next load can issue. By adjusting the stride it is possible to cause a variety of behaviors in the cache hierarchy and thus determine timing information for access to different levels of cache and for events like TLB misses.

3.3.2 Netperf

Netperf is a collection of network microbenchmarks developed by Hewlett-Packard, including benchmarks for evaluating the bandwidth and latency characteristics of various network protocols. Of the various benchmarks, I selected TCP stream, a transmit benchmark, and TCP maerts, a receive benchmark. In both of these benchmarks, the client informs the server of the benchmark and the server acts either as a sink (for the transmit benchmark) or as a source (for the receive benchmark), consuming or producing data as fast as it can. These benchmarks are simple, just filling a buffer and calling `send()` or `receive()`. Thus they spend most of their time in the kernel TCP/IP stack and interrupt handler, and very little time executing user code.

3.3.3 SPEC WEB99

SPEC WEB99 is a popular benchmark for evaluating web server performance. It simulates multiple users accessing a mix of both static and dynamic content over HTTP 1.1 connections. The benchmark includes CGI scripts to do dynamic ad rotation and other services a production web server would normally handle. The web server used for the benchmark is Apache version 2.0.52 with the mod_SPECWEB99 module that replaces the generic reference CGI scripts with a more optimized C implementation and is available from the SPEC and Apache websites.

The M5 developers also wrote their own client request generator that is more appropriate for a simulation environment while preserving the workload characteristics of the

Category	Description
Alignment	Time spent processing reads or writes to unaligned addresses.
Buffer	Time spent dealing with buffer management issues.
Copy	Time spent copying packets around in the kernel and to/from user space.
Driver	Time spent executing code from the network driver.
Idle	Time spent with the CPU being idle.
Interrupt	Time spent handling I/O and timer interrupts (not including device driver code).
Other	Time spent in the kernel that doesn't fall into any of the other categories.
Stack	Time spent processing packets in the TCP/IP protocol stack.
User	Time spent executing a user level process.

Table 3.1: Description of CPU Utilization Categories.

benchmark. In this work, I used their request generator. The standard SPEC WEB99 score is the maximum number of simultaneous clients a server can support at a minimum bandwidth per connection. Each client requests a reasonably small amount of data at a maximum rate of 400kbps and expects that request to be serviced within a certain time. The server's score is normally determined by configuring a very large testbed of client machines and iteratively tuning the number of connections and various server parameters. Given the enormous slowdown incurred by simulation, this iterative approach is clearly impractical for simulation. Since I was interested in the performance characteristics of the benchmark and not the actual score achieved, I used the M5 developer's request generator that enabled a single client to scale up its performance to saturate the web server, thus avoiding the iterative tuning step. This request generator is based on the Surge [34] traffic generator, but using the statistical distribution of requests from the stock SPEC WEB99 client.

3.4 Methodology

This section describes the metrics used to compare the simulated results to the real hardware and the methodology employed to gather data.

3.4.1 Metrics

Two metrics were used to compare the simulated and real systems: bandwidth and CPU utilization. The primary comparison focused on the bandwidth achieved by the server, since that is of greatest concern for network-oriented benchmarks. The data is presented both in terms of absolute performance between simulated and actual configurations and relative performance between two configurations running at different clock rates. This enables

me to determine how useful the simulator is at predicting the magnitude of performance increases even if it cannot provide absolute numbers. The secondary metric was CPU utilization. CPU time down is broken down into several categories (Idle, Other, User, Copy, Buffer, Stack, Driver, Interrupt, and Alignment) and the relative amount of time spent by the simulated and actual machines is compared. See Table 3.1 for a description of the categories.

3.4.2 Simulated System

Full-system cycle-level simulation is orders of magnitude slower than real hardware, making it impractical to run benchmarks to completion. To cope with this limitation, I turned to checkpointing, fast-forwarding, cache warmup, and sampling to reduce the simulation time while maintaining the characteristics of the benchmark. These techniques, however, cannot be applied blindly since the TCP protocol is self-tuning. Sampling too soon after switching from a function to a detailed model can produce incorrect results [35].

To apply fast-forwarding I functionally executed the code with a perfect memory system. This allowed me to quickly boot the system and reach an interesting point of execution, at which point I create a checkpoint. From this checkpoint I then switched to a simple CPU with a memory hierarchy and warmed up the caches for a tenth of a second. At this point I switched to a detailed CPU model and took 8 samples of a tenth of a second each. These samples were then averaged to create the results shown below.

The system of interest varies depending on the benchmark; the system under test was the client for netperf and the server for SPEC WEB99. To ensure that the system under test can perform to its fullest capacity, the other machine was simulated with a perfect memory system, making it artificially fast.

I sampled the program counter every 100 cycles to measure the amount of time the CPU spent executing different categories of code. With the use of the kernel symbol table I found the symbol closest to that address, keeping track of how many times each symbol was seen during the execution. In a post processing step these function names were aggregated into the broad categories presented above.

3.4.3 Real System

My testbed consisted of one Compaq Alpha XP1000 running with either a 500MHz EV6 or 667MHz EV67 processor and a National Semiconductor NS82830 based Gigabit Ethernet card. To stress the Alpha I used a dual-processor Opteron with a Tigon III Ethernet card. For the real system I had the luxury of running benchmarks to completion at the price

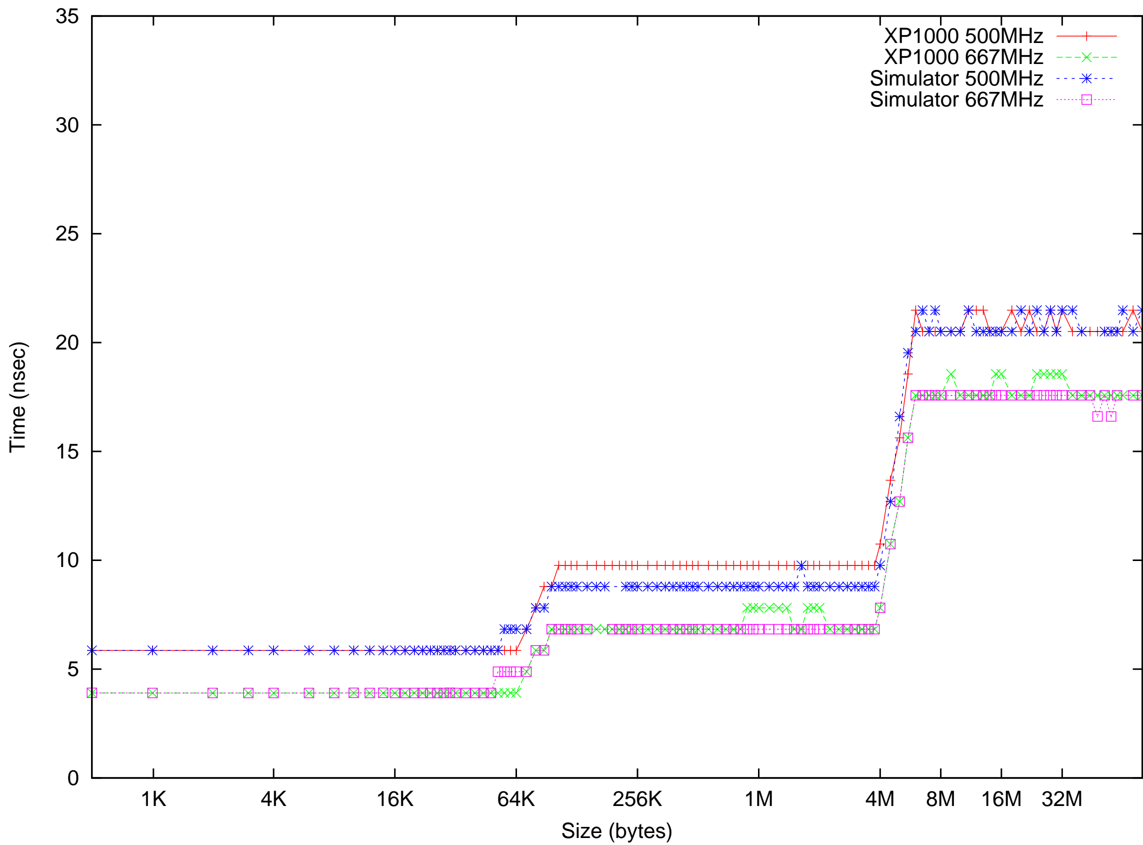


Figure 3.1: mem_lat_read: Memory Latency with Stride 8

of the data gathering slightly perturbing the results. To reduce this interference as much as possible I ran for hundreds of seconds sampling the transmit and receive byte counts on the Opteron at 30 second intervals and using OProfile[36] sampling every 100,000 cycles to obtain the Alpha's CPU utilization breakdowns.

3.5 Comparison to M5

In this section, I compare my measurements of two real Alpha XP1000 systems to the simulator's results for similarly configured systems. The first section describes my results for the memory microbenchmarks I used to calibrate M5's modeled memory latencies. The second section discusses the network benchmarks that are of primary interest.

3.5.1 Memory Latency

Figures 3.1 through 3.3 compare the simulated system to the real hardware.

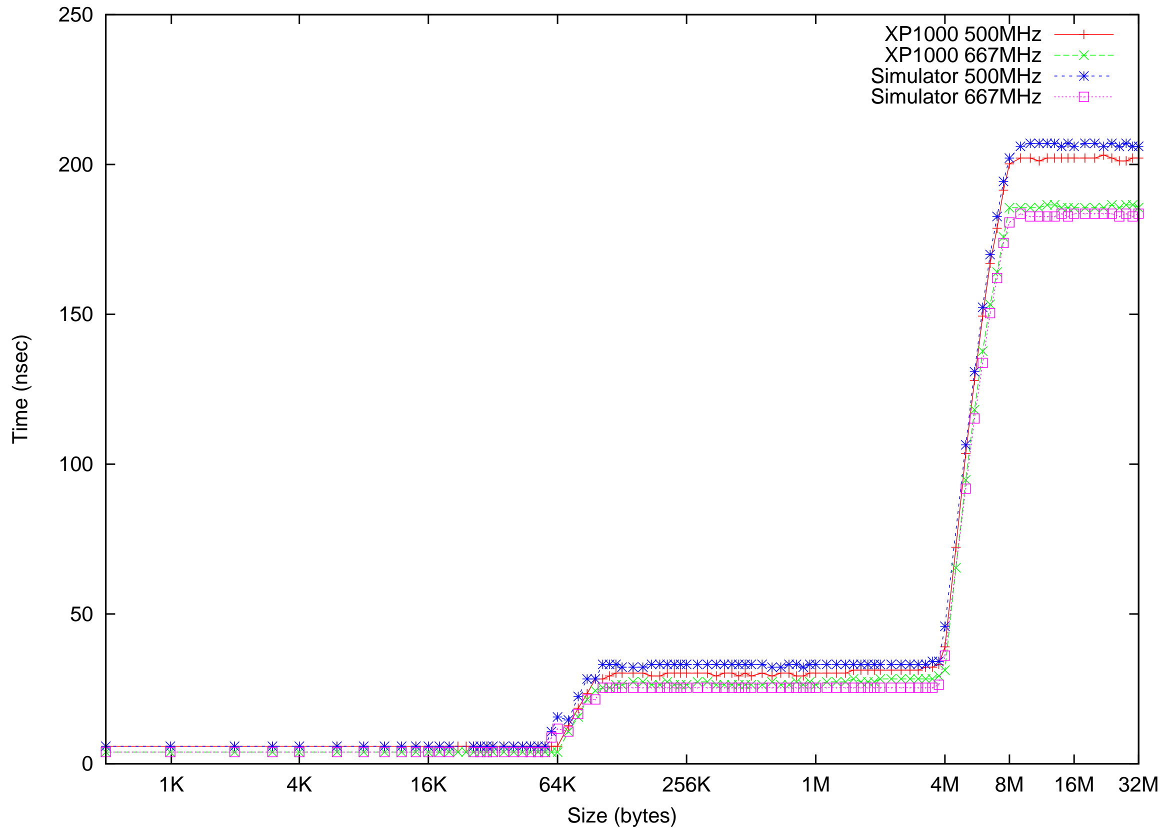


Figure 3.2: mem_lat_read: Memory Latency with Stride 64

In Figure 3.1 I configure `mem_lat_rd` to read every word sequentially. This setup incurs an L1 cache miss every 8 accesses. For sizes over 4MB, the L1 misses also miss in the L2. I tune my simulated memory system latencies so that my data matches the real data rather closely. The little bump around 64kB is due to page mapping issues. The Alpha has a virtually indexed physically tagged L1, while M5 currently supports only physically indexed caches. Thus pages within a 64KB array could conflict in the L1 whereas they cannot on the real machine. The other discrepancies observed for the 500MHz system are within the timer resolution latency.

Figure 3.2 shows results for a 64-byte stride, and is mainly concerned with L2 hit time. For array sizes larger than the L1 cache, all accesses are L1 misses. Again, I have successfully tuned M5 to model the real hardware latencies. The simulated main memory access times seen at the right end of the graph are within 4ns of the measured times.

Figure 3.3 focuses on TLB miss latency by showing results for a stride of 8kB. Below 4MB, each access is a TLB miss but an L2 hit. At the right end of the graph, accesses miss in both the TLB and the L2 cache. I was able to model the TLB miss latency precisely for

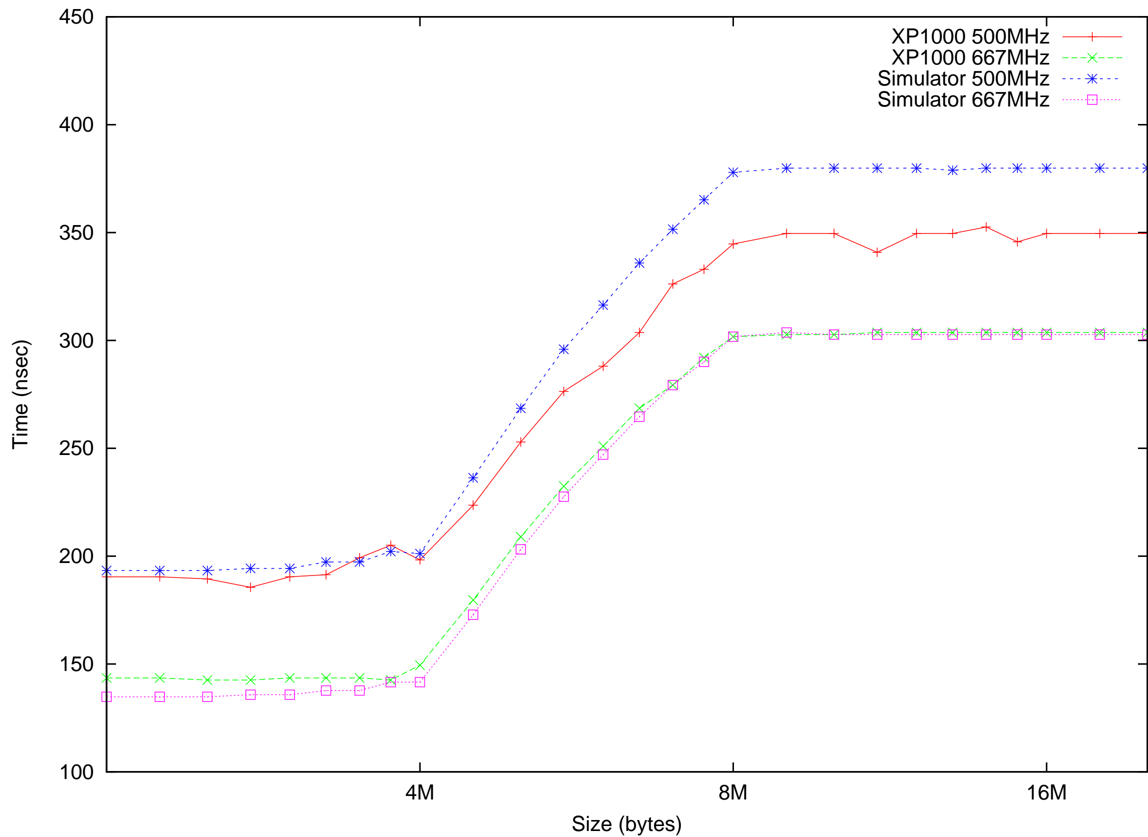


Figure 3.3: mem_lat_read: Memory Latency with Stride 8k

both situations on the 667MHz machine. However, the 500MHz machine poses a greater challenge. Using the same TLB miss overhead (in CPU cycles) as in the 667MHz machine, I was able to correctly model the cost of a TLB miss that hits in the L2 cache. However, the latency for a combined TLB/L2 miss on the 500MHz system takes approximately 30ns longer on the simulator than on the real machine, in spite of having an accurate latency for main memory accesses on this system (see Figure 3.2). I am unable to explain this phenomenon, and it doesn't appear to occur on the 667MHz system. Figure 3.4 illustrates this effect by plotting just the TLB miss penalty (subtracting the memory access latency) for multiple runs of the microbenchmark. The reason for this difference in stability also eludes me.

3.5.2 Network Benchmarks

In Figure 3.5 I show the absolute difference in performance between the workloads running on the simulator and on real hardware. Then in Figure 3.6 I show the percent error

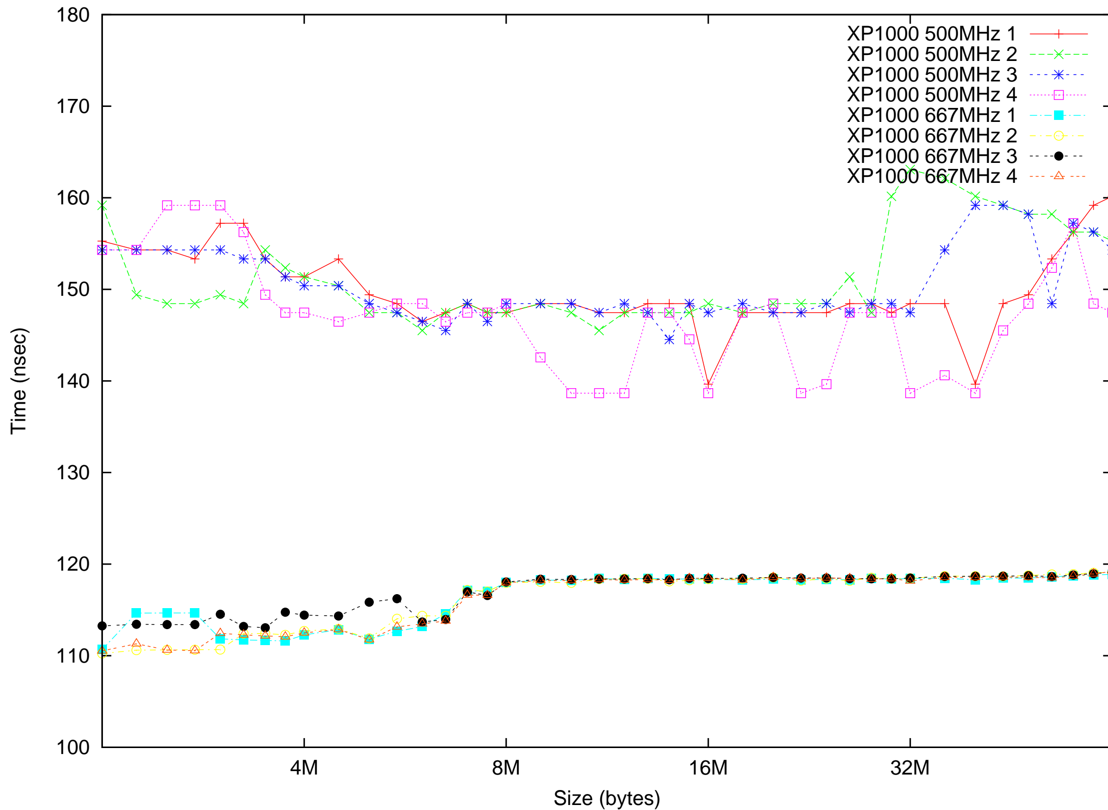


Figure 3.4: mem_lat_read: Memory Latency Stability with Stride 8k

in terms of absolute bandwidth between the simulated and real machine, and relative error comparing the speedup between the 500MHz machine and the 667MHz machine on both the simulator and the actual hardware.

The results in Figure 3.5 show that the simulated and real systems' network performance deviate by no more than 20Mb/s. In general the results tend to underestimate the performance of the microbenchmarks a little bit and overestimate the performance of the macrobenchmark. Turning to Figure 3.6, the blue and purple bars show that M5 models the performance of the systems within 11%. Additionally, comparing the relative performance increase of the 500MHz system to the 667MHz system shown in the yellow bar, here too M5 deviates by no more than 11.5% from the real hardware.

Since M5 does not model the hardware in excruciating detail and only uses generic components to construct a virtual system, these results are very good. There are many potential sources of error in the M5 models; for example, M5 does not model clustered functional units like the real EV6 does. Although the model of the ethernet interface is

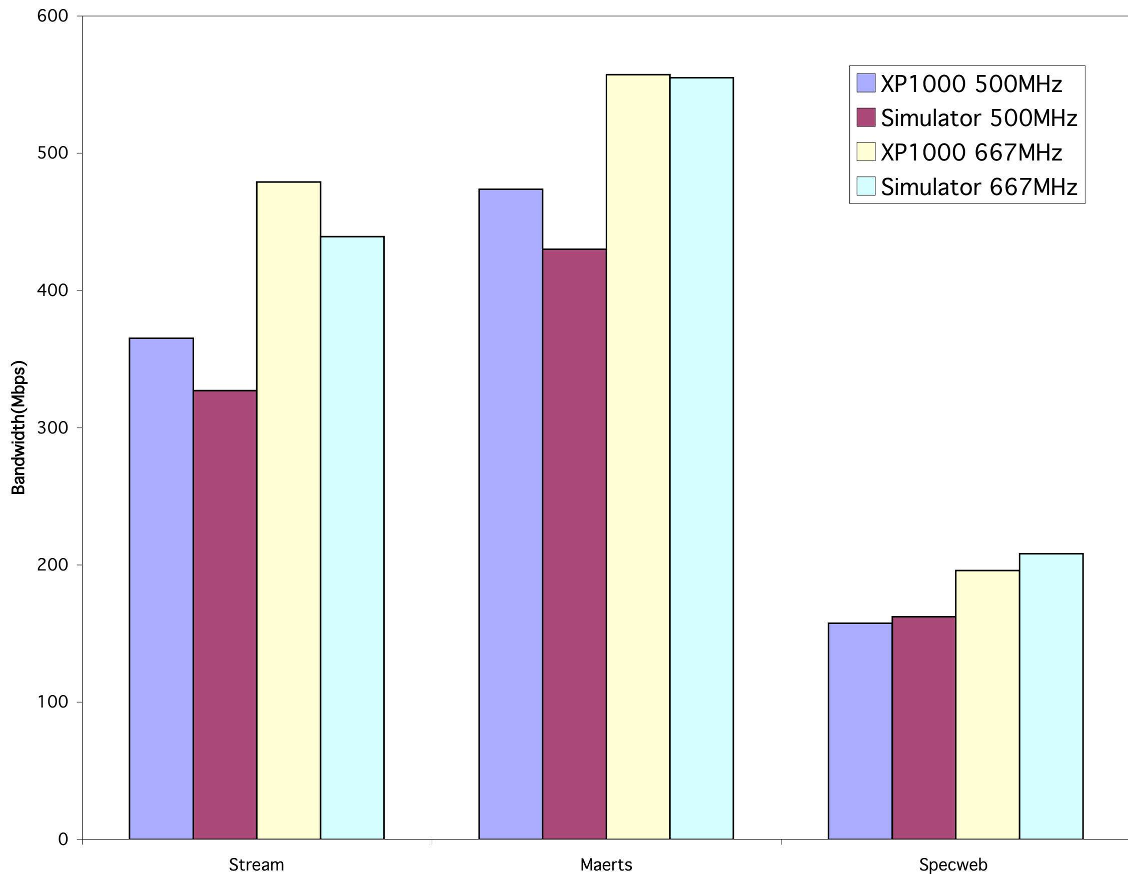


Figure 3.5: Absolute Bandwidth (Mbps) for benchmarks

based on the datasheet for the real device, the internal aspects of how the device operates, what operations it can do concurrently, and how fast it can transition between states of operation are unknown and thus force assumptions to be made.

In Figure 3.7 I compare the fractional CPU utilization broken down into the categories mentioned above. As mentioned previously, the data was gathered from OProfile on the real hardware and using high-frequency sampling (every 100 cycles) on the simulator. Note that the simulator spends more than twice as much time in interrupt code as the real system. I believe this is due to OProfile not being able to interrupt the CPU for a monitoring event if the CPU is at a higher interrupt priority level. This includes all the PALcode, and any points in the kernel that are uninterruptible. Unlike OProfile, M5's sampling is not affected by the CPU's interrupt priority level and thus it makes sense that the results show a larger amount of interrupt time.

The other two discrepancies seen in the utilization graph are a difference in time spent in TCP/IP stack processing and the lack of idle time in the simulated stream runs. I hy-

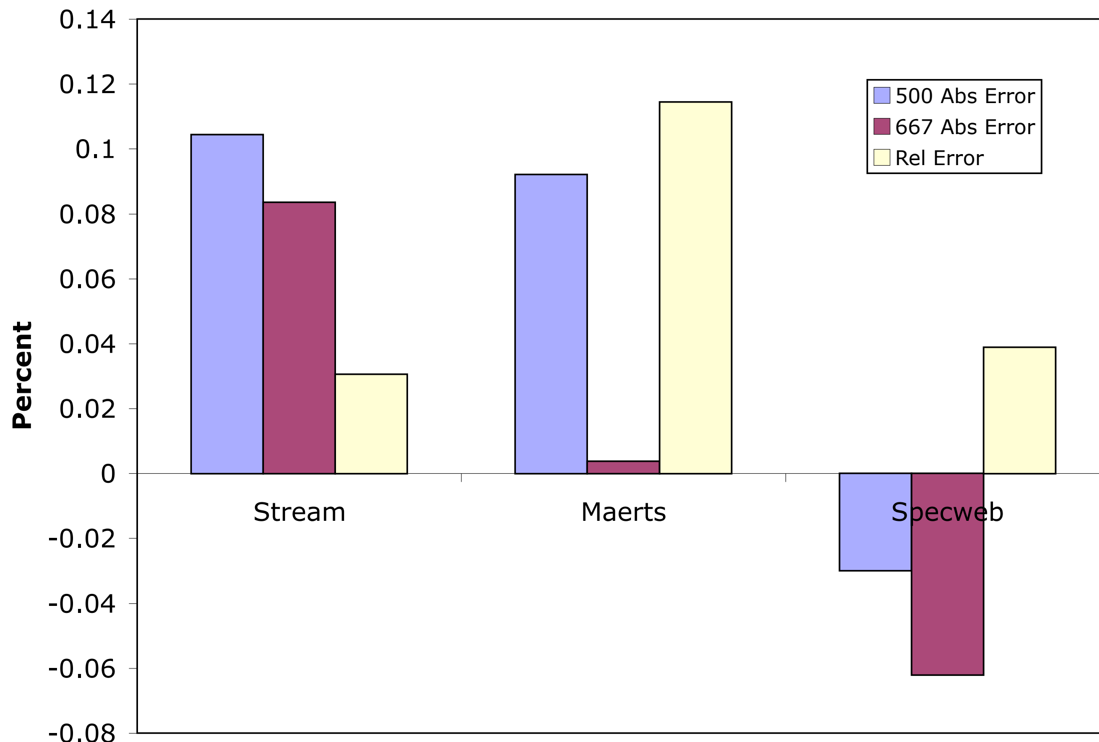


Figure 3.6: Relative Bandwidth (Mbps) for benchmarks

pothesize that this is due to M5's CPU model underestimating some aspects of the real CPU's performance. This effect is not surprising, as I spent most of my tuning effort on the memory and I/O system.

Even considering the above problems, there is a strong correlation between the simulated utilization breakdowns and the actual hardware.

3.6 Sensitivity of Results

In this section I look at the impact of various architectural parameters both in terms of performance change for the benchmarks and their stability across different CPU speeds. Additionally, I evaluate the spatial stability of the chosen workloads and verify that the changes observed by changing architectural parameters are not simply noise.

Figure 3.8 plots the percent change between the simulator's nominal output for a benchmark at the given speed and the simulator's output with the parameters on the X axis. If the bars corresponding to a single benchmark at different speeds show a similar increase then even though the absolute bandwidths may not be the same, the relative increase will

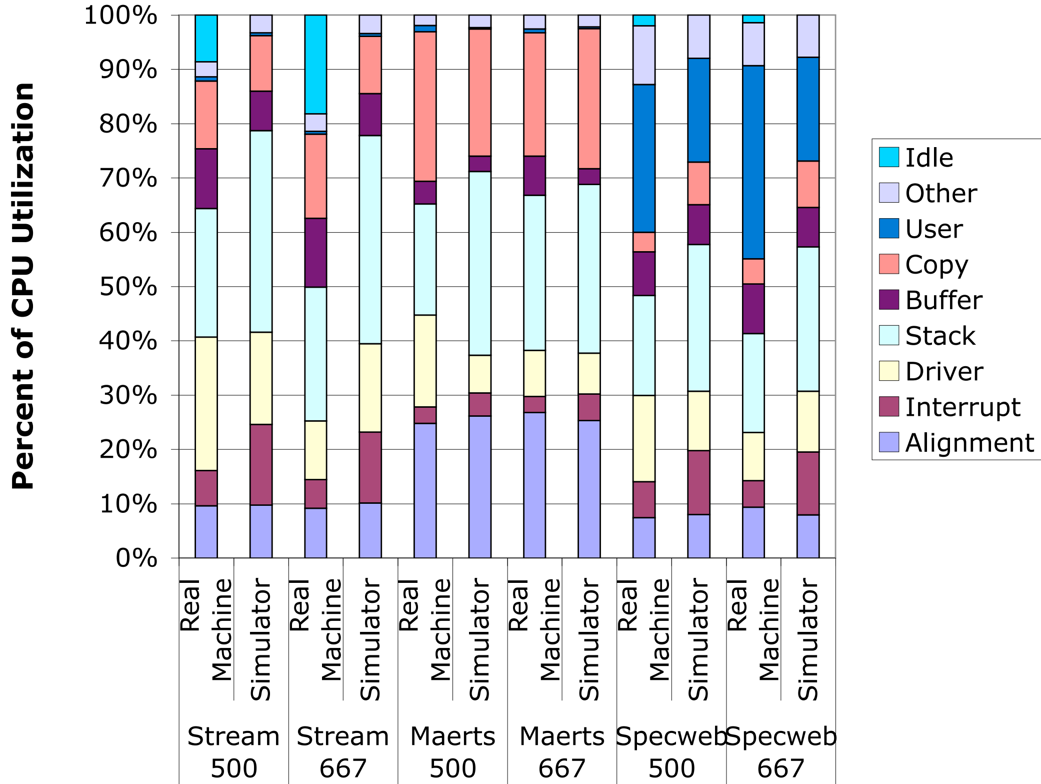


Figure 3.7: Processor Utilization over Benchmarks

be approximately the same.

3.6.1 Issue Window

The first part of Figure 3.8 shows the impact of varying the instruction window size. The big instruction window triples the size of the ROB and instruction queue, yielding 240 and 192 respectively, while the small IW halves the size of the ROB and IQ, yielding 80 and 32 entries respectively. Across all the benchmarks these changes are stable as the change between the two processor frequencies is minimal. In most cases there is also a minimal performance change, except in the case of the maerts benchmark, which is receive biased. In the receive case, packets have been copied to memory by the NIC and when the kernel is going to copy the data to user space it must take cache misses. The bigger instruction window allows more misses to get started and reduces the copy time from 18% to 8%. The impact of this isn't seen in the more transmit oriented benchmarks because the data to

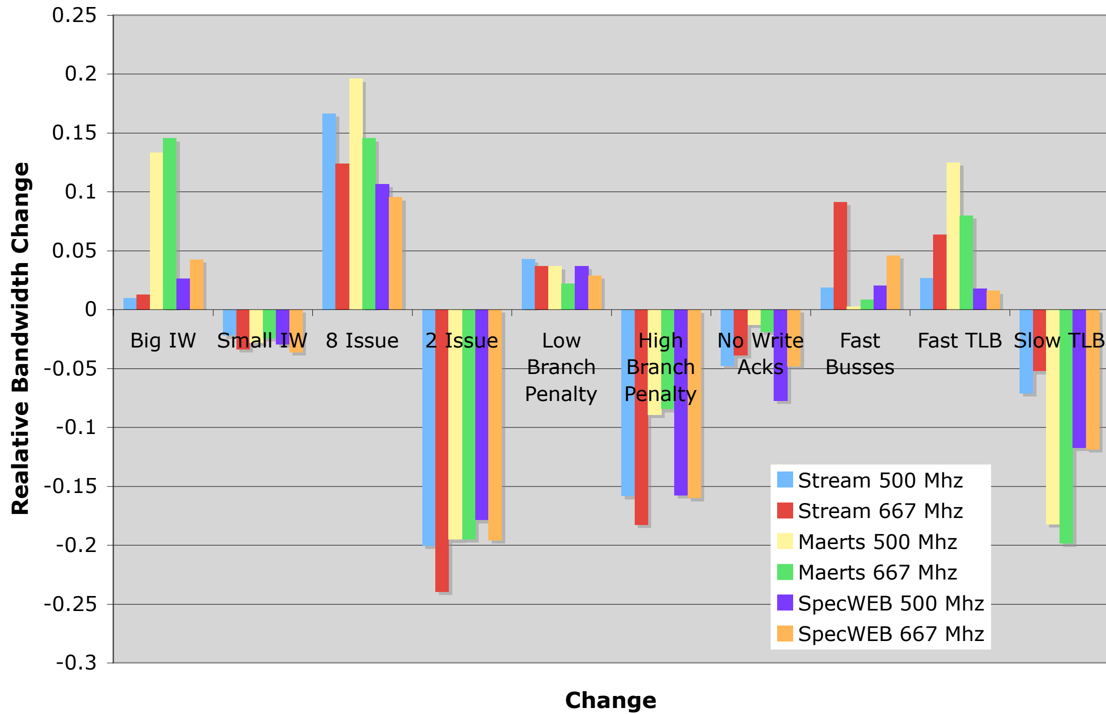


Figure 3.8: Sensitivity of results after varying parameters

transmit is already in the cache and therefore there aren't expensive cache misses during the copy.

3.6.2 Issue Width

Varying the issue width of the machine greatly affects the performance. The results show an increase of at least 10% and as much as 20% by doubling the width, and a decrease by 20% for halving the issue width. These results are some of the least stable as a machine with a slower clockrate can benefit more from an increased issue width if the workload is stalling the processor regularly waiting a constant time for long latency miss. In particular, the stream benchmark benefits more from increased width and is hurt less by a decreased width. The more complex macrobenchmark doesn't exhibit this change because it isn't as I/O bound in the 667MHz configuration. SPECWEB has a lot more processing than the microbenchmarks so the increase in issue width doesn't improve the performance of one system more than the other.

3.6.3 Branch Mispredict Penalty

Normally an Alpha EV6 has a branch mispredict penalty of 3 cycles. In the above graph I change these values to 1 and 10 cycles for the low mispredict penalty and high mispredict penalty, respectively. The larger penalty adversely affects the stream and SPECWEB benchmarks because they have more difficult branches. On average the mispredict rate for these benchmarks is 6-7% higher than the maerts benchmark. Even though changing the penalty affects the benchmarks differently, the relative performance increase is still stable, changing by only a couple of percentage points.

3.6.4 Acknowledging Writes

While analyzing a previous version of the CPU utilization numbers mentioned in Section 3.5, it was evident that the time the simulated system spent in a function updating the chipset interrupt mask register was much higher than it was on the real machine. An investigation into this issue led me to conclude that on the real machine, uncached writes to this register were getting acknowledged by the memory controller before the write had actually propagated to the device register. As a result, the register's write latency is substantially smaller than the read latency I measured with the kernel microbenchmarks described in Section 3.3.

The effect of the corrected write acknowledgment timing on network bandwidth depends on the workload. The number of uncached register writes is correlated with the number of network packets transmitted per interrupt. Since the SPECWEB benchmark has the most think time, and therefore transmits the fewest packets per interrupt, it sees the largest change in bandwidth of about 8% because of this error. The stream benchmark transmits packets with a 5% change in bandwidth, and finally the maerts benchmark experiences little to no change since it only transmits acknowledgement packets. Acknowledging writes earlier and reduces the time spent in the device driver and interrupt handler, bringing the utilization breakdown more in line with the real system. Again, this change doesn't appear to affect the relative performance increase between the two CPU runs, and is thus a stable change.

3.6.5 Faster Busses

For this parameter I doubled the frequency that the internal busses in the simulator up to the CPU frequency. The results shown in Figure 3.8 are not stable. This shows that the stream benchmark running at 667MHz is I/O limited, while the stream running

at 500MHz and the maerts benchmark are not. The 667MHz stream run with the faster bus is 9% faster than the normal run, while the 500MHz run is only 2% faster. The faster bus does help the SPECWEB benchmark a little bit as well, but since there is much more computation involved in this benchmark it doesn't help to the extent that is shown in the stream benchmark.

3.6.6 TLB Miss Penalty

To further match the observed TLB miss penalty, I added an arbitrary delay of 20ns to the invocation of the TLB miss handler. Eliminating this delay (labeled Fast TLB in Figure 3.8) causes a 10% change in bandwidth for the maerts benchmark and doubling the delay (labeled Slow TLB in the same figure) results in a 20% change in bandwidth. Thus, for some benchmarks what is normally considered an unimportant parameter can have a huge effect on performance. The bandwidth seen from the stream and SPECWEB benchmarks is reduced by about 10% with the longer TLB miss penalty and increases performance by only a few percentage points with the shorter TLB miss penalty.

These results are not stable as the relative increase in performance varies by as much as 10%. In general the maerts benchmark is more sensitive to the TLB miss penalty because it uses a large amount of system memory for receiving packets and since it doesn't reuse the buffers that the translations often miss in the TLB. The other benchmarks aren't affected as much by variations in the TLB latency, however the TLB latency can play almost as large a role in the performance of a system running network intensive workloads as the issue width.

3.6.7 Spatial Stability

To address the possibility that the variation between runs with different parameters is not due to the relatively short sampling window and more optimal scheduling decisions I modified the M5 simulator to be able to inject random variations in the DRAM access delay and the delay on the ethernet link [37]. Over eight runs the coefficient of variation for the microbenchmarks was under 1% and for the macrobenchmark the variation was under 5%. Given this information we can be reasonably certain that the differences in performance are not due to spatial variability, but are caused by the architectural changes.

3.7 Related Work

Bedicheck [21] validated the Talisman simulator, designed to model the Meerkat prototype multicomputer, against the final hardware. He achieved remarkable accuracy, but beyond a wide range of microbenchmarks he reports only on a few small kernels without any OS or I/O activity. Meerkat was based on the in-order, single-issue Motorola 88100 CPU, so detailed CPU modeling was not required.

Gibson et al. [22] validated the various simulation models used to develop the Stanford FLASH prototype (including SimOS-based full-system models) against the prototype hardware itself. In accordance with the focus of the FLASH project, the workloads used for validation were parallel compute-bound applications from the SPLASH-2 suite [38], which did not involve significant OS or I/O activity. Interestingly, they found (as I did) that the overhead of software TLB miss handling indicated a significant discrepancy between their simulator and the real machine. They also found that a fast, simple in-order processor model often gave results as accurate as a more detailed out-of-order model.

Desikan et al. [24] created and validated a model of a Alpha 21264 microprocessor against a Compaq DS-10L workstation. They spent considerable effort modeling detailed aspects of the 21264 processor core, such as replay traps and clustered execution, resulting in an error of less than 2% for CPU-bound microbenchmarks. However, their inability to model complex interactions within the memory system caused their error on SPEC CPU2000 macrobenchmarks to average 18%. This error reflects in part issues that are effectively non-deterministic and cannot be accurately matched in a simulator, such as physical page mapping effects on cache and DRAM page conflicts and DRAM refresh timing.

3.8 Conclusion

This chapter discussed the validation of the M5 simulator by comparing M5 models of two Compaq Alpha XP1000 servers with their real-world counterparts. I compared network bandwidth and CPU utilization for two network-intensive micro-benchmarks and a macro-benchmark. The M5 models were able to get within 11% of the real machines' bandwidth in all benchmarks. Furthermore, the simulation results accurately reflect the impact of varying CPU frequency. This level of accuracy is quite good given the relatively generic and imprecise models used in the simulator, and the fact that my only major effort in tuning for the reference machine was to correlate memory and device register latencies.

Additionally, this work shows that incorrectly modeling one of a variety of architectural parameters usually doesn't affect the absolute bandwidth attained by a benchmark

by more than 10% and many parameters don't affect the relative performance seen by two configurations. However, some parameters do greatly affect the performance of these workloads. The issue width is a commonly considered parameter, however the TLB miss penalty affects performance almost as much as the issue width, although the parameter is rarely considered important.

Chapter 4

Creating an End-to-End Dependence Graph

Critical-path analysis has been applied in many domains to find bottlenecks in complex, concurrent systems. In addition to providing an analytical means for bottleneck identification, the resulting dependence graph also provides information about slack and criticality, which architects, system designers, and developers can use to understand how altering the execution time for a single event can change the overall performance of a system.

Critical-path analysis requires a dependence graph that represents the timing constraints between events. For a simple system, it can be easy to construct the required graph; however, the difficulty of building such a graph can increase rapidly as systems become more complex. In the most extreme case, the number of interactions in a system is proportional to the square of the number of events.

This chapter describes a key technique that facilitates the creation of a dependence graph for large and complex systems without detailed knowledge of all the components involved. The creation of this dependence graph enables applying critical path analysis to a much wider scope. The resulting dependence graph can locate bottlenecks, pinpoint the under-provisioned resources, and estimate the performance when these limitations are removed.

4.1 Basic Technique

To create a dependence graph I algorithmically map the execution of state machines that govern software and hardware component behavior on a global dependence graph. The technique involves two steps. First, the execution of each individual state machine is converted to a portion of the dependence graph. Next, dependences between the state machines are recorded in the form of interactions on a shared data structure, principally a FIFO queue.

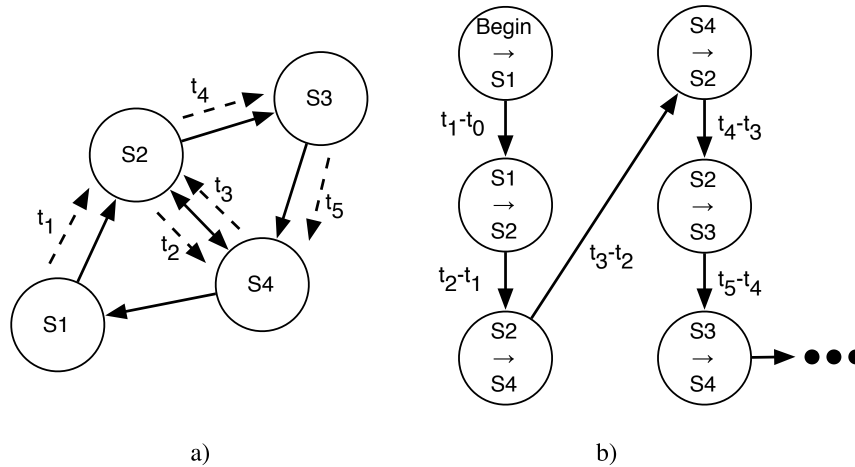


Figure 4.1: Conversion of a state machine - a) The state machine being converted; nodes are labeled with the state name; transitions are labeled with the time they are traversed. b) The state machine converted to a dependence graph; nodes are labeled as the transition between two states; edges are labeled with the time spent in the corresponding state.

This chapter begins by describing the conversion of an explicit state machine to a dependence graph. While some systems, such as hardware devices, are designed as state machines, software normally is not. Thus, for software systems, I have the additional challenge of generating a meaningful implicit state machine from the program execution. This process is substantially automated by using program flow information to identify states. Finally, identifying interactions between state machines must be performed manually; however, this can be done in an incremental, iterative fashion that eliminates the need for substantial up-front effort.

4.1.1 Explicit State Machines

The execution of a state machine can be converted to a dependence graph through a simple algorithmic transformation. The events of interest in a state machine's execution are the transitions between states. These transitions, which traverse the edges of the state machine diagram, become the nodes of the dependence graph. Conversely, edges in the dependence graph, which represent the time spent waiting between events, correspond to the time spent in a particular state (node) of the state machine.

Figure 4.1 illustrates the conversion process. The state machine on the left begins at t_0 in $S1$ and transitions to $S2$ at t_1, having spent time $t_1 - t_0$ in $S1$. At time t_2, it transitions to $S4$, thus spending time $t_2 - t_1$ in state $S2$. Next at time t_3, the state machine transitions back to $S2$, spending $t_3 - t_2$ in $S4$. This process continues for the rest of the arcs illustrated

in the graph. In the right-hand part of the figure, I show the resulting dependence graph. Instead of accumulating time in the nodes of the graph, like the state machine graph does in the left-hand side of the graph, time is accumulated on the edges. Additionally, unlike the canonical state-machine graph, transitions result in additional nodes being added to the graph.

4.1.2 Implicit State Machines

Some system components, such as hardware controllers, are normally designed as explicit state machines, however many other components are not. This is frequently the case for software; even when the software is based on a state machine, the states and transitions are difficult to extract from the code. To construct a dependence graph as outlined above, I have the additional challenge of decomposing software into meaningful states.

While an expert may be able to identify the state boundaries in a complex program, such experts are rarely available. To enable non-experts to identify the state machines—even in such substantial pieces of unfamiliar software as the network stack in the Linux kernel—I can automatically create an initial decomposition by using function entry and exit points (calls and returns) to delineate state boundaries. These events can be recorded at runtime with the aid of a binary recompiler, JIT, simulator, or VM. If the symbol table is available, the states can be labeled according to source function names.

Given this automatic decomposition, the manual effort is reduced to two tasks, both of which can be performed incrementally. The first task is refining the automatic decomposition in cases where the function decomposition results in states that are too coarse. The second task arises when a single executable binary (e.g., an application or the Linux kernel) contains multiple state machines. In this situation, users must indicate when the CPU's execution path leaves one state machine and enters another. This event typically corresponds to a call from one subsystem to another, e.g., from the protocol stack into the device driver, and is not too difficult to recognize. The CPU may also switch from one state machine to another when executing an interrupt handler or time-slicing between applications. The manual effort takes the form of placing annotations or markers in the code that identify the state machine and state boundaries. These annotations are further discussed in Section 4.5.2.

Quantifying the difficulty of a particular task is complex; however, in my experience to date the effort required to instrument substantial pieces of code has been minimal. For example, fewer than 100 annotations were required to instrument the entire UDP code path—from application to driver—in a Linux 2.6 series kernel. Similarly, the TCP code

path requires approximately 300 annotations and can be completed in a few days. In both cases iterative refinement was used to hone the annotations.

4.1.3 State Machine Interactions

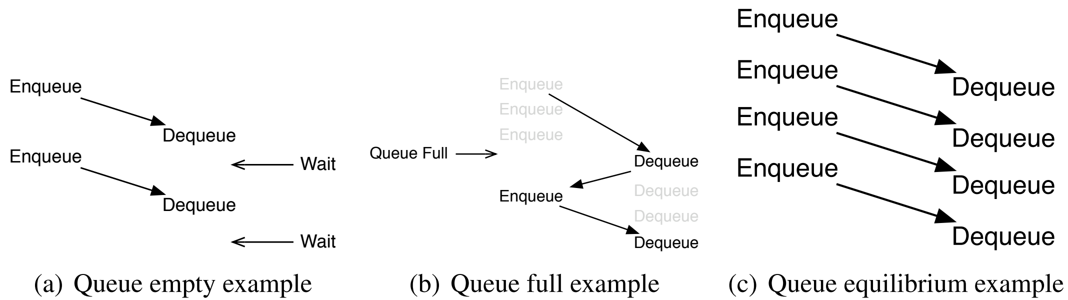
Given a system decomposed into state machines, a global critical path can be constructed by tracing the execution of the individual state machines and their interactions. The interactions consist of situations in which one state machine waits for an event to occur in another state machine (such as the production of a data value or an interrupt signal) before making a transition itself. As with the detailed decomposition of software state machines, these interactions can be identified fairly quickly by non-experts using iterative refinement.

These state machine interactions are modeled in two ways. The principle method of interaction is via a queue, although other data structures are possible. State machines interact through a queue when one state machine produces a value that another state machine consumes. Additionally, state machines can interact when the queue is full; in this case a state machine that produces a value can be dependent on a state machine that consumes a value. Section 4.2 provides a more detailed explanation of the queue interactions.

An additional method of interaction is used to allow code to be segmented into different state machines when there isn't any shared buffering, but rather a function call in a software state machine. There are several cases in which a software state machine can span multiple separate pieces of code or one distinct piece of code utilized several times as a sub-module. Some examples of this include an application executing a system call or different layers of the TCP/IP stack that do not have buffering. In these cases, it is worthwhile—although not strictly necessary—to label the state machines differently so that, at a high level, information about where time is being spent can be quickly observed by a user. When a function call that should be segmented occurs, an edge is created between the two state machines and an edge back to the original state machine when the appropriate return occurs.

4.1.4 Critical-Path Analysis

Once the state machines and their interactions have been identified, the global critical path can be extracted from an execution trace automatically. A global dependence graph is constructed using observed latencies as edge weights. The weights on edges representing waiting states are set to zero, so that the longest path between two nodes represents the critical path of execution. The critical path between any two nodes of interest—generally starting where a request is made and ending where the corresponding response is received—can



(a) Queue empty example (b) Queue full example (c) Queue equilibrium example

Figure 4.2: Example of queue dependencies

then be found with standard graph analysis techniques.

4.2 Queue-based State Machine Interactions

State machines often interact with each other indirectly through shared resources. For example, a producing state machine may provide data to a consuming state machine through a queue. There are three possible cases that are each illustrated in Figure 4.2. If the queue is empty (Figure 4.2(a)), the consuming state machine waits idle for the producing state machine to produce an element. Similarly, if the queue is bounded and full (Figure 4.2(b)), the producing state machine waits idle for the consuming state machine to remove items from the queue. Finally, if the queue is at equilibrium (Figure 4.2(c)), the state machines still interact, though they never idle waiting on each other since the queue is never empty or full.

Capturing these indirect interactions is key to several aspects of my analysis, including the ability to track multiple connections (see Section 4.3), to detect resource-limited critical paths (see Section 5.4), and to predict speedups (see Section 5.5).

Dependencies are expressed as operations on a (logical) queue between state machines. (To date, I have been able to model all interactions via a queue with FIFO ordering, although other shared resources or more complex manipulations would be possible.) I have identified the following queue operations as sufficient for my work to date:

(1) enqueue(n): add n items;

(2) dequeue(n): remove n items;

(3) peek(n): look at, but do not remove n items;

(4) reserve(n): reserve space for n items to be added;

(5) size(n): resize the queue to contain exactly n items;

(6) wait_empty(n): wait because n items are not available in the queue;

(7) wait_full(n): wait because space for n items is not available in the queue.

Note that the "items" stored in each queue are abstract entities. In most cases, single objects such as packets are placed in the queue, and the *enqueue* and *dequeue* operations have a default argument value of 1. However, these queues are also used to represent storage limits for structures such as the socket buffer, in which case n may be set to the number of bytes in the current send request or packet. Queues are also used to represent the TCP flow control and congestion control windows. For example, one queue tracks a connection's TCP send window, with the TCP transmit state machine consuming items from the queue (and blocking when it is empty), while received ACKs add items to the queue. The *size* annotation is used only when the TCP stack resizes one of the protocol windows.

I record these queue manipulation events in the same fashion as other state-machine transitions. In hardware models, the manipulations are annotated directly in the models. In the case of software state machines, I annotate the source code with pseudo-instructions.

This queue-based model as applied to connect the pieces of the dependence graph produced by transforming each execution into a state. Any time one state machine produces an element that is buffered and consumed by a second state machine, the producing and consuming events are linked with an inter-state-machine edge. For example, a dequeue or peek operation creates a dependence between the state machine node entered by the peek or dequeue and the state machine node that originally enqueued the referenced item(s). The weight on this edge corresponds to the communication latency between the producer and the consumer. When the rate of enqueueing and dequeueing are mismatched, the queue will become empty or full. I explicitly record when these conditions occur and, once observed, zero the weights on intra-state-machine dependence-graph edges for the waiting state machine so that the critical-path analysis will properly follow the inter-state-machine edge.

To better illustrate this process, consider the state machines shown in Figure 4.3. These are simplified versions of state machines that might exist in an IP stack and a driver. In this case, the stack state machine and the driver state machine interact via a single queue. The stack state machine places packets in that queue and the driver state machine removes packets and places them in a DMA descriptor that is given to the NIC.[1] The dashed arrows

[1]In a real system, the actual state machines would interact with other state machines and have many additional states; this simplified version exists only to illustrate the creation of a dependence graph.

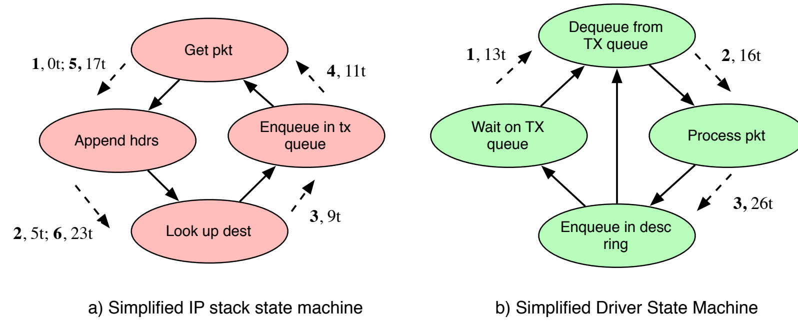


Figure 4.3: Example state machines: a) simplified IP stack state machine; b) simplified driver state machine.

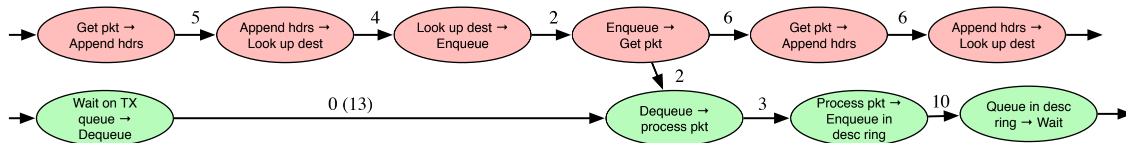


Figure 4.4: Example dependence graph showing the interaction of the two state machines of the previous figure.

represent observed transitions the state machine made in the form of (sequence number, latency). Thus, at time 0, the stack state machine transitioned from *Get pkt* to *Append hdrs*, and at time 5 the state machine transitioned from *Append hdrs* to *Look up dest*, etc.

Figure 4.4 illustrates a dependence graph created from the observed transitions in Figure 4.3. Initially, the driver state machine is idle, waiting for a packet to be present in the queue. Although 13 time units elapsed in this state, that edge is given a weight of zero because the state machine was stalled waiting. A value of 2 time units is placed on the edge between the stack and the driver state machine, because the driver state machine waited 2 time units between the data becoming available and being consumed. This is the communication delay between the stack state machine and the driver state machine.

4.3 Tracking Multiple Connections

A state machine implementation often supports multiple concurrent instances. For example, the kernel's TCP stack and a web server application can both process multiple connections. To analyze modern systems, I must identify which instance of a state machine is transitioning when an annotation is encountered. To this end, a unique key must be identified to distinguish instances for each state machine, and the current value of that key must be provided with each annotation. The value is treated by my extended analysis program as an opaque identifier. For example, in the TCP stack, the pointer to the per-connection

transmission control block (TCB) is used as the key, while in an application, the file descriptor of the connection's socket could be used. For a machine with multiple NICs, the Ethernet MAC is used to distinguish the independent device driver state machine instances.

These queue-based annotations enable the analysis program to automatically associate state-machine instances (e.g., which application file descriptor goes with which kernel TCB value) by noting which instances of two state machines communicate. This association is effective even when shared queues or state machines interpose; for example, I can associate the related TCBs on the client and server though the connection's packets even though they pass through a single shared state machine at the device driver layer.

I even automatically identify which state machines have per-connection instances and which are shared among connections by noting whether a single instance processes queue entries corresponding to only one or to multiple instances of the application-level connection state machines.

4.4 Refining Annotations

An important feature of my approach is that annotations can be developed iteratively, using the results of an analysis on an incompletely annotated system to indicate where further effort is required. My queue-based model requires additional information in the annotations, but also provides extra feedback to help users generate that information correctly.

A state machine graph for an incompletely annotated system can be quickly scanned for incongruous states. Spurious edges connecting disjointed components tend to indicate that a state machine begin or end annotation is missing. Queue annotations are constantly checked for consistency at runtime and can produce a variety of warnings. These can include notification of a dequeue operation on an empty queue, a state machine waiting to dequeue from a non-empty queue, etc. I also provide an *assert(n)* annotation that will raise an error if the queue being maintained in the model does not contain exactly n items.

Another technique that helps to identify shared resources and the state machines that wait on them is to analyze a pair of applications communicating at low bandwidth. The critical path should be in the application state machine, waiting for the next message to be sent. All other state machines should be mostly idle; any kernel or hardware state machine that remains active indicates a missing wait annotation.

4.5 Implementation

This work has been implemented in the context of a full-system simulator, which provides deterministic results and complete visibility of all software execution and the hardware models. The analysis is performed in two phases.. First, the simulator simulates the system(s) of interest and generates a file with the resulting annotations that I process with an analysis program in the second phase. This decoupling of data generation and analysis allows for interactive processing which would not be possible if the analysis occurred during the simulation. While a simulator is used in this work, it is possible that a similar system could be created in the context of a real implementation using a binary recompiler, JVM, JIT, or VM, to instrument the system. In this case the visibility into the hardware would be greatly limited, so the analysis would end at the software's edge.

4.5.1 Simulator

The M5 simulator [25], as discussed in Chapter 3, was used for this work. The latest development releases were modified to support the recording and output of annotation data for later analysis. Hardware state-machine transitions and interactions are recorded by inserting function calls directly in the simulator models to record relevant information on each occurrence. Software components are annotated through a combination of observing CPU execution within the simulator and the addition of pseudo-instructions into the relevant binaries. These pseudo-instructions are inserted explicitly in the source code via annotation macros, and exploit unused opcodes to instruct the simulator to record the occurrence of a state-machine transition or other event of interest. These pseudo-instructions must be fetched and executed by the simulated CPU, so they do perturb the execution; however, they are reasonably rare and execute instantaneously, so their effect is minimal. The particular annotations used are detailed in Section 4.5.2.

Simulator Output

The simulator periodically records and buffers the annotations writing the buffer to disk. The annotations are stored in a binary form along with a "key" that allows the records to be converted to meaningful names where required. The ability to store and later process annotations provides flexibility in the type of analysis done, but the cost of this flexibility is the storage space of the annotations. For a typical detailed simulation, the storage cost is modest—hundreds of megabytes per second of simulated time.

Frequently in the realm of architectural simulation simulations are started from check-

points which store the entire state of the simulation. These checkpoints allow for a simulation to be quickly functionally executed to a point of interest and then a checkpoint can be created. The checkpoint can be used to run multiple experiments as long as they don't alter the architectural state of the machine (e.g. various bus widths), but they can't be used if some system-visible parameter is changed (e.g. the size of main memory).

Because dependencies are tracked through queues, the queue state must be preserved across simulator checkpoints. When a checkpoint is created, the entire internal state is saved. In addition, the various records that are still in each queue are also saved to the checkpoint. When restoring from a checkpoint, these records are placed in the appropriate order in the queue and immediately emitted to the output file. Thus, the first set of records in an output file that is the result of a restored checkpoint are queue operations to populate all of the queues that have items in them. Without this data it would be extremely challenging to match up enqueue and dequeue pairs when the data is being analyzed, thus limiting the usefulness of the data.

4.5.2 Annotations

The annotations are used to capture two classes of information state transitions and state machine interactions. The state transitions are simply when a state machine transitions from one state to the next, while the interactions capture when one state machine creates an output that induces a change in a different state machine. These interactions are generally modeled as queues so the annotations add or remove elements from a queue, and by keeping track of the elements in that specific queue interactions can be established.

Annotations are used to identify state machines and their interactions in both hardware and software. These annotations are captured by the simulator at run time and eventually are written in a packed format to a simulator output file, as described above. The hardware and software annotations do largely the same thing, however because they're being used in different places the syntax varies slightly. Specifically, the software annotations can manipulate states that the simulator retains and can modify the behavior of future annotations, while the hardware annotations are stateless and thus every annotation fully specifies the transition or interaction that is occurring. In principle, software could be annotated by specifying every transition and interaction; however, that would increase the burden required to use the tool.

Identifying State Machines

A state machine that is being annotated is uniquely identified by the system to which it belongs and an instance identifier that makes multiple instances of the state machine unique. The value of this identifier depends on the the state machine being annotated but it could be any unique value such as the MAC address in a NIC, or the socket structure inside a kernel. The state machine name, system, and instance identifier are combined to form a unique integer, the state machine identifier, which is a globally unique ID.

Identifying Queues

Queues are identified in a manner similar to state machines. The queue name alone is not sufficient to uniquely identify an instance of a queue, so the queue is coupled with a unique identifier. This opaque identifier could again be any number of values such as the MAC address or the socket structure that belongs to the current connection. Just as in the state machine identification the system, queue name, and queue instance id are combined to form a globally unique queue ID. Unlike the state machine identification however, the queue can potentially reside in another system (e.g. the end-point of a network link probably does not exist in the current system), so the queue identifier is created from the system in which the queue exists, not in which the annotation was seen.

Flags

A variety of flags may be passed to the annotations to alter their behavior in some way. Some annotations have a number of flags that are used while others have none, although most annotation operations allow the specification of flags.

Hardware Annotations

There are seven hardware annotations that are used for the annotating TCP/IP flows between two systems.

- **`hwBegin(flags, system, instance, state_machine, state)`**
  The begin annotation marks the transitioning of the given *state_machine* from the previous state it was in to the state given by *state*. To further identify the state machine in question a *system* is provided along with a *instance*. These values uniquely identify the state machine that is transitioning as described above in Section 4.5.2. The only *flag* that may be recorded with this annotation marks the state the state machine currently entered as suspect. This is normally done to states that either signify

an error condition of some type, or to mark a piece of a state machine that hasn't been annotated yet.

- **`hwQ(flags, system, instance, state_machine, queue, q_instance, q_system, count)`**
  The queue annotation enqueues *count* units of data in the specified *queue*. As described above in Section 4.5.2, the *system*, instance, and *state_machine* are combined to identify the state machine instance performing this enqueue operation. The *queue* parameter names a queue that is being operated on and, as described in Section 4.5.2, the *queue*, *queue_instance*, and *queue_system* are combined to form the unique queue ID that is having data enqueued in it. If the *queue_instance* or *queue_system* parameters are not specified, then the *system* and *instance* parameters are used. There are currently no *flags* that modify the behavior of the queue operation.

- **`hwDq(flags, system, instance, state_machine, queue, q_instance, q_system, count)`**
  The dequeue annotation operates in the exact same fashion as the enqueue operation; however, in this case items are removed from the queue as opposed to being added to the queue. If there are fewer units of data available in the queue then were requested by *count*, an error is printed and the annotations need to be further refined. If a special *count* of -1 is specified, all items are removed from the queue by this dequeue operation. This is useful for pseudo-queues such as the list of packets yet to be acknowledged. One acknowledgement will acknowledge all packets that have been received, so all items can be removed from the queue in this case.

- **`hwPq(flags, system, instance, state_machine, queue, q_instance, q_system, count)`**
  This peek annotation operates in the exact same way as the dequeue annotation, but items are not removed from the queue. The dependency between the state machine that did the queuing operation and the state machine that did the peek operation is still added. This annotation is used in cases in which a state machine wants to examine the item in the queue before it is consumed. The item will eventually be dequeued, but the analysis can not happen without the element present. Therefore, a dependency must be set up when the value is used, as well as when it is removed.

- **`hwRq(flags, system, instance, state_machine, queue, q_instance, q_system, count)`**
  The reserve annotation operates in the same manner as the queue annotation, but no

items are added to the queue when a reserve occurs. The reserve creates a dependence between the dequeue of an element that provided space in a queue and and activity that a state machine will now do because of the available space. This annotation is useful when a state machine verifies there is enough space in a queue before it takes some lengthy action to produce more data. If space isn't available that action is delayed, therefore there should be a dependence between the space becoming available and the state machine producing data to fill it. Normally, state machines produce data and then wait until they can place it in a queue. However, in a few cases this annotation is required.

- **`hwWe(flags, system, instance, state_machine, queue, q_instance, q_system, count)`**
  The wait empty operation has the exact same parameters as the above described queue operations. The wait empty operation signifies that the given state machine is blocked waiting on some *count* units of data to be available in the specified queue. When this annotation occurs, the state machine stops accumulating time and some bookkeeping is updated, indicating that the state machine is in this state. At this time, if the queue has the requested number of units of data in it, a warning is printed as it normally indicates improper annotations[2] When the state machine next does a dequeue operation on this queue the dequeue operation will set up the dependency as normal. However, an additional weight will be placed on the edge that connects the two state machines. This weight is the time between when the data was available and when the waiting state machine removed the data (the communication latency). This ability is extremely important. Frequently, the communication latency can dominate a critical path and performance gains can be achieved by reducing that latency.

- **`hwWf(flags, system, frame, state_machine, queue, q_instance, q_system, count)`**
  The wait full operation is similar to the empty condition, however here the given state machine is blocked waiting on *count* units of space to be free in the given queue. In this case, when an enqueue operation follows a wait full operation, a dependency is set up between the state machine that removed the items from the queue and the state machine that needed space in the queue. The dependence (edge) is again weighted with the elapsed time between when the space was available and the queue operation happened.

[2] It can also indicate the communication latency between the consumer and the producer is large. In this case an item may be available in the queue, however the consumer has no way of knowing that information.

Software Annotations

The software annotations ultimately generate the seven hardware annotations that are described above along with a few additional ones. The additional ones are not necessarily specific to software, but since the end-points of interest normally exist in software, and software queues can be arbitrarily created and destroyed, some operations tend to be useful only in software. Unlike the hardware, parameters and even complete annotations are extracted directly from the execution as opposed to being completely specified. This reduces the effort required to analyze a given system substantially.

Begin annotations are automatically created from the execution and symbol tables. When the simulator observes the system executing a branch-and-link instruction it automatically creates a begin annotation for the target symbol.

- **`swBeginStateMachine(state_machine, instance, flags)`**
  This operation maintains a set of stacks of state machines, one for each execution context in the machine. The begin state machine pushes the active state machine identifier on the top of the stack that belongs to the current execution context of the machine. It creates the identifier exactly in the same manner the hardware annotations above did. This operation combines the system the annotation was executed on, along with the *state_machine* and the *instance* to create a unique state machine identifier. A *flag* can be specified to indicate that the previous state machine was directly interacting with this one (e.g. a system call, or some layering of the kernel). This flag creates an additional `link` annotation that informs the analysis program that the two state machines are directly interacting at this point. In addition, the fact that this link occurred is stored in a map so that when the current state machine ends, a `link` operation can be automatically emitted to link the state machines in the reverse direction.

- **`swEndSm(state_machine)`**
  The end state machine annotation provides the reverse operation of the begin state machine annotation, it removes the top most state machine identifier from the stack of state machine identifiers for the current execution context. The annotation takes a single parameter, the name of the state machine that is ending. While this is not strictly necessary since it is simply removing the top most element on the stack, the state machine name allows M5 to verify that state machines are nesting properly. If the stack is not unwinding correctly, a end state machine annotation is probably missing. As mentioned in the previous description, if a link flag was passed to the

`swBeginStateMachine()` operation then a link operation is emitted to return the previous state machine.

- **`swBegin(flags, state)`**
  Software states are normally begun automatically by observing the program execution. The program counter, privilege level, and currently running process information are used to locate the appropriate symbol table and create a new begin annotation each time a branch-and-link instruction is observed. The topmost element in the state machine stack provides the state machine identifier for this annotation. In some cases, the user may want to subdivide a large function into smaller pieces. In this case, a software begin annotation can be manually inserted. Like the hardware version of this annotation, a flag can be specified that marks the current state as "bad," indicating that this path has not been annotated or is not expected to be executed. The state is converted into a unique ID per state machine for more compact storage.

- **`swQueue/swDequeue/swPeek(flags, queue, q_instance)`**
  The software queue annotations operate in the same manner as their hardware counterparts. The only difference is that the state machine that is doing the operation is implicitly specified by the state machine identifier stack for the current execution context just as is done for the software begin operations.

- **`swWaitFull/swWaitEmpty(queue, q_instance, count, state_machine)`**
  The wait annotations occur in the same way as the hardware annotations occur, except that like the other software annotations the current state machine identifier is specified implicitly. Additionally, the wait annotations have a *state_machine* parameter causes the wait annotation to be immediately followed by an end state machine annotation when it is specified.

- **`swSizeQueue(flags, queue, q_instance, sizes)`**
  The size queue operation changes the size of the queue either by queueing additional elements or removing some elements. Much like the other queue operations the *queue*, *q_instanced*, and current system are combined to find a unique queue. If this queue presently has fewer elements than *size*, additional elements are queued to make it *size*. If the queue is too large, elements are removed from the back of the queue until it is the appropriate size. This operation is used when the underlying storage of a queue is expanded or contracted (e.g. when the TCP receive window changes size). An optional *flag* can specify that the queue is to be completely cleared and then requeued. This is useful in cases in which a queue is reused for a different connection

and was never completely emptied. Stale elements in the queue can create improper dependencies in the graph and must be removed.

- **`swAssertQueue(queue, q_instance, size)`**
  The assert queue operation is used to assist the annotator in adding the manual annotations. The annotation causes a warning to be printed if the specified queue has a size other than the *size* specified by the annotation. It is used to verify that the annotations are correct and pinpoint the location where a problem is occurring.

- **`swGetId()`**
  The get identifier operation returns a unique connection ID and marks the state machine that created the annotation as belonging to that connection. This is used to trace connections from start to end. The ID is returned to the caller for future use.

- **`swIdentify(id)`**
  This operation creates an identity for the specified *id*. It is used to verify that the state machine has not changed the ID of the connection it belongs to unless it has done so intentionally.

4.5.3 Annotation Perturbation

Since annotations are inserted directly into software they have the ability to perturb the system. I analyzed the perturbation that the annotations introduced into the system by measuring the change in bandwidth between an annotated system and an unannotated system. To verify that these annotations are not introducing a large overhead, I compared the performance of various configurations studied in this dissertation (see Chapter 6 for details).

The perturbations I saw are a result of several things:

- A larger binary due to the inserted pseudo-instructions
- Different code emitted by the compiler (since functions are of a slightly different size the compiler could choose different optimizations for the function)
- Interrupts and scheduling occurring at different times in the simulator due to the difference in the code size and instruction counts

The annotations generally had very little effect on the bandwidth produced by the system. On average, the bandwidth produced changed by only 3%. This variation included both a slight increase in the bandwidth produced, by as much as 2%, and slight decreases in

the bandwidth produced, by as much as 7%. Generally, experiments that were CPU limited were affected more by the annotations than ones that were otherwise limited. This result seems reasonable as the CPU limited experiments are most likely to be slowed down by larger code size and resource consumption within the processor.

While the change in bandwidth is minor, it would be possible to completely remove all the annotations in the code. The compiler could generate a mapping between address and annotation that could be used for the same purpose in a simulator.

4.6 Conclusion

This chapter has shown how critical path analysis can be applied to a large computer system consisting of multiple layers and machines. A dependence graph can be constructed without detailed knowledge of the underlying implementation by algorithmically mapping the execution of the state machines that govern the system into a global dependence graph. The resulting graph describes all dependencies and interactions within the system and can be used, as shown in the following chapter, to locate bottlenecks and predict performance of hypothetical systems.

Chapter 5

Locating Bottlenecks and Analyzing Data

While having the data necessary to create a dependence graph is interesting, the data alone does not provide any analysis or insight. This chapter describes the analysis program that takes the raw data, constructs a dependence graph, and provides higher level metrics to interpret the new graph. First, the task at hand is described, including the time and space complexity of the analysis and how the complexity can be reduced. Next, the methodology for creating the graph, finding the critical path and tracking multiple connections simultaneously, is presented.

The remainder of this chapter is dedicated to describing the analysis that is provided by the analysis program. While the dependence graph contains a huge amount of data, high-level conclusions about the system are not readily available from the graph without aggregation and analysis. The analyses I have developed to draw high-level conclusions from the graph are presented in multiple ways.

5.1 Processing Data

The analysis program processes the data provided by the simulator and creates a useful representation of that data that can be used for the analysis. The program has to be cognizant of the temporal and spatial complexity of the data processing and analysis. Since millions of annotation records are processed, small inefficiencies in the code can lead to unacceptable performance.

5.1.1 Time and Space Complexity

A naive way to process the data would be to simply create all the nodes and edges corresponding to the recorded annotations. In this case, adjacency lists would be preferable

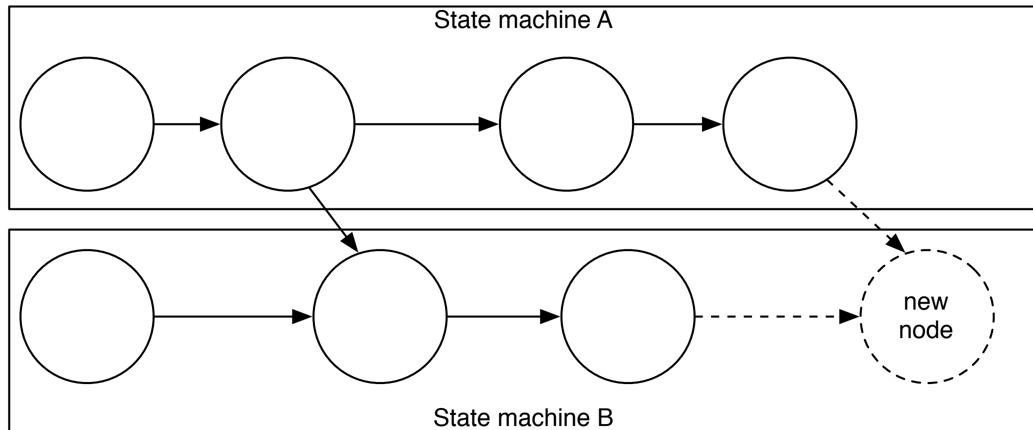


Figure 5.1: Illustration of a graph construction.

to an adjacency matrix since there are a large number of nodes and they are sparsely connected ($\Theta(V + E)$ vs. $\Theta(V^2)$). After the graph is created it can be topologically sorted, and the longest path can be found in the graph using standard graph analysis techniques. The topological sort would take $\Theta(V + E)$, initializing the bookkeeping information takes $\Theta(V)$, and doing the edge relaxation for the entire graph takes $\Theta(E)$[39]. While none of these algorithms are inefficient, there is the possibility for many millions of nodes. Even an efficient representation can take gigabytes of memory, which makes it hard to store and difficult to process. Even when the entire graph can fit in memory the topological sort can take quite some time to construct and requires $\Theta(V)$ additional memory.

5.1.2 Limiting Memory Usage

As mentioned above the amount of RAM required to build a graph in memory depends on the number of edges and vertices in it. A graph with millions of nodes can consume an extremely large quantity of memory so steps must be taken to limit the memory usage. Furthermore, the only reason to build the graph is to locate the critical path, as visualizing many millions of nodes is not feasible, nor would it be particularly useful.

The structure of the graph and data provide a low-cost method to find the critical path without building the entire graph in memory. Since all the annotations recorded by the simulator are done so in a total order, if the records are processed in order the next record always happens after the current record. Additionally, each annotation in the file has a strict total order between it and other records belonging to the same state machine.

Nodes in the graph are connected to a single node in the same state machine (the last node that was inserted into the graph belonging to the same state machine) and possibly another state machine. Thus, when a node is inserted into the graph the critical path is

either from the previous node in that state machine, or a node that the state machine just interacted with (see figure 5.1). In other words, all the annotations recorded for analysis are already topologically sorted. Therefore, for each state machine in the graph there is only one critical path to its end. Rather than keeping the longest path information for each node, it can instead be kept for state machine. Since the number of state machines in the graph is orders of magnitude less than the number of nodes, the longest path can be found with significantly less storage and without creating the entire graph. A similar method was utilized in the online critical path processing work of Hollingsworth [40]. Depending on the desired data, the critical path to that point can be recorded, or if a more memory-optimal implementation is desired, statistics about the edges traversed (their name and total time) can be stored, rather than the longest path (which would contain many instances of edges with the same name). Since the graph is too large to visualize, the data must be aggregated in some way for analysis. Thus, it's beneficial to just aggregate the data up front, summing the time in each state and transition count instead of building the graph.

5.1.3 Analysis Internals

This section provides information about the internals of the analysis program. It begins by describing how the data is read, parsed, and fast-forwarded, and then proceeds to discuss how the dependencies are created.

Data File

The various annotations that are captured are written to a file for later processing. The file consists of a header containing version information, an offset, and length to all the records that were recorded, and an offset and length to a key that can convert the unique state machine and queue IDs into readable names.

The majority of the file is composed of fixed length records that contain a time stamp, the CPU on which the event occurred, the operation, the state machine ID, the state ID/queue ID, any corresponding data, and a set of flags. The key provides a set of arrays and dictionaries that map the various unique integer IDs to readable names. An array of all the state machine and queue IDs map each ID to a name, instance id, and system ID. Another array maps system IDs to system names. Finally, a dictionary of state machines' names maps each state machine to an array of state names that can be indexed with the state ID.

DataParser

A DataParser class is used to provide an abstraction layer between the data file and the analysis program. This abstraction layer allows changes to the data file format and layout without requiring changes to the core of the analysis code. The DataParser first verifies that it is compatible with the version of the analysis file it is being given to read, and then creates descriptions of the various records in the file. The key associating strings with the unique IDs stored in the file is given to the NameResolver class so it is able to return strings given to the unique IDs in the future.

Queue

A `queue` object is created for every queue in the system. It tracks the elements currently in the queue and where they came from. The queue also tracks which state machines are waiting on the queue (either to have space or have an element) and provides the appropriate dependence information when an element is added or removed. The queue tracks various statistics such as the occupancy, time spent waiting for space in the queue, time spent waiting for data, the communication latency through the queue, service times, and the data arrival and departure rate.

A queue or dequeue is split into two phases. During the first phase, the dependency is retrieved and during the second phase the queue is updated to reflect the operation state. Dependencies are passed through the queue with the DepInfo object that contains the node in the graph that did the enqueue operation, the critical path to that node, the critical path length up to that point, the time the operation was done, the number of units of data that are being queued, and the connection ID that is doing the enqueue operation.

When a state machine is waiting on a queue, either because of a full or empty annotation, the state machine calls the waitFull() or waitEmpty() functions in the queue object with the state machine, the time, and the number of data units that it is waiting for/on. These functions check that the operation makes sense. These tests include:

a) If a state machine is waiting on the queue to have a number of units in it there shouldn't be that many units available in the queue at the present time [1]

b) If the state machine is waiting on the queue to have space for a number of units and the queue is empty

c) If the state machine was already waiting on the specified queue

[1] It is possible that the queue does contain an element, but the communication latency in the system hasn't made that information visible to the requesting state machine yet.

d) If the state machine was already waiting on the specified queue and the condition that it was waiting on has been satisfied

When the first phase of an enqueue operation happens (`queueDeps`), the state machine that is doing the queue is checked against a list of state machines that were waiting to queue an item. If a match is found then there is a `DepInfo` object containing information about the state machine that removed the item that made space available in the queue. This item is returned along with the time from when the space was made available to the time this queue operation happened. If the state machine wasn't waiting to queue an item then nothing is returned. In either case, various internal statistics are updated.

In the second phase the `DepInfo` object is placed in the actual queue that the queue object manages. The number of units of data in the queue is updated along with various statistics. If there are any state machines waiting on the queue to have data and the current insertion satisfies their request the waiter is transitioned from the waiting structure to the dependence structure, which is searched during the first phase of the removal.

The first phase of the dequeue operation (dequeueDeps) is similar to the queueDeps phase above. First the queue checks that there are enough units of data available for the request to be satisfied and raises an error if not enough units exist. Then the various statistics that the queue keeps are updated and the dependence information along with the communication latency is returned to the state machine that dequeued. In the dequeue case, dependence information can always be provided since all data is being tracked. On the other hand, since the size of the queue may not be known, dependence information can not always be provided in the queue case.

The second phase of the dequeue operation proceeds similarly to the second phase of the queue operation. Here, the structures that describe which state machines are waiting on the queue are updated to reflect that the state machine that just dequeued an element may have made enough room in the queue. Now, the appropriate number of data units are removed from the queue.

The final operation that the queue does is the resize operation. This operation removes items from the back of the queue to make the number of data units in the queue equal to the given size. For instance, this sometimes happens in TCP, when the congestion window shrinks.

Fast-forwarding Records

The data file created by the simulator starts when the simulator started running either initially or from a checkpoint. In the case of starting from the beginning of execution all the

queues in the system will have items inserted in them and that information will be stored in the data file. When restoring from a checkpoint, the checkpoint contains the current state of all the queues in the system and that information is inserted into the data file before the simulation resumes. To maintain the queue state all the records from the start of the data file to the record of interest must be minimally processed. During this fast-forwarding time only Enqueue, Dequeue, and SizeQueue operations are processed. Many fields of the DepInfo object are left unpopulated when passed to the queue. When the actual analysis begins, the queues have the correct items, however since there are no nodes to create dependencies with they do not have that information.

5.1.4 Constructing the Graph

Each record is processed in order to create the graph. The analysis program first compares the record type to each known type. In the case of the identify operation the current connection for the given state machine is changed to be the connection ID of the connection that the record describes, and that connection is added to the set of all connections for the given state machine. In all other cases the analysis program uses the previously identified connection ID for the state machine that is executing the operation. Next, the state ID is found. For a begin record this information is encoded in the record itself, for the queue operations the Queue object initially creates a new ID for the operation and caches it, and for the link operation the analysis program creates an appropriate ID if one doesn't already exist.

If a state machine has not been observed before a dummy state is created so that the initial node has a starting state in the case of a software state machine the analysis program next checks if the currently executing state machine is the same as the previous state machine that executed on this system and CPU. If not, some bookkeeping is updated to reflect that the previous state machine stopped executing at the current time. This information is later used to display how much time a state machine that was otherwise ready to execute spent waiting for the shared processing resource.

A node is now created describing the transition from the previous state of this state machine to the new state. If the node doesn't exist in the graph it is added to the graph.

After the node is created, if the operation is a enqueue, dequeue, or peek, the first phase of these operations is executed by getting the dependence information from the queue. Additionally, the time spent waiting on this queue is recorded to display later and the current connection may be updated to the connection ID of the item being removed from the queue. Further information about connection tracking is provided in Section 5.1.5.

The new node could be dependent on a previous state machine because of a link operation. The link's dictionary is searched for a matching entry and if one is found the current connection is also updated to the linked connection. A single node can not get dependence information from both a linking and a queue operation.

If the operation identifies the given state machine as waiting because a queue is full or empty, that state machine is marked as waiting on a queue and any time that would normally accumulate is zeroed. Additionally, any requests that the user made to reduce the time that a state machine spent in a particular state are honored (this is used for finding secondary critical paths as described below).

State transitions (nodes) and states (edges) are inserted into a compact representation of the graph. Only one node or edge exists for every unique transition and state, even if they occur multiple times during the analyzed execution. If the edge does not exist it is inserted into the graph. Otherwise, statistics on the edge are updated to reflect how many times that state was seen and how much time was spent in that state. The statistics that are collected include:

- The total time spent in its state (edge)
- The amount of time spent waiting in its state.
- The amount of time the critical path spent in its state.
- The amount of time the critical path spent waiting in its state.
- The number of times this edge was visited.
- The number of times this edge was visited on the critical path.

Next, the critical path information is updated. See Section 5.1.4 for information about the update process.

Lastly a new dependence info object is created in the operation was queue operation (enqueue, dequeue, etc.) and the second phase of the queue operation is called. If the state machine was waiting on a particular queue, and it executed an operation that satisfied that dependence, it is no longer marked as waiting and its time is accumulated as normal.

When all the records have been processed the critical paths from the start state machine/state to the ending state machine/state are converted from the internal format to a list. The critical path statistics on the edges are updated and a list of connection IDs and the state machine IDs that they ended on is created. This list traces the start of every connection to all possible endpoints where data from that connection is processed.

State Machine Interaction

There are two ways a pair of state machines can interact. They can either do so through queues, or they can interact directly. The direct interactions are not required, but rather act as a construct for simplifying output and grouping state machines into smaller units than they might otherwise be.

Most state machines in a system interact through a series of queues. These queues normally are bounded in size and some producing state machines place an item in a queue that is eventually consumed by the consuming state machine. If no items are available in the queue the consuming state machine is idle waiting on data to consume, and similarly if the queue is bounded and full the producing state machine is waiting on the consuming state machine to consume an item before it can continue.

The simple queue and dequeue operations set up the basis for interactions between state machines. When an item is placed in a queue and is subsequently removed by a state machine at a later time, the two state machines are said to interact and a dependence can be created between the state machines through the queue.

More information can be obtained if one of the state machines is observed to be waiting on the queue. If the consuming state machine is blocked while waiting for an element to be present in the queue, then the time from when an element is made available in the queue and when that element is removed from the queue is the communication latency between the two state machines in this particular instance.

If the sizes of the queues are known, a reverse dependence can be set up between the removal and subsequent insertion of items in a queue. If the size isn't known, but the producing state machine is seen waiting here too, a dependence can be created. Just like in the case of an empty queue the communication latency between the item removal and the producer placing the next item in the queue can be calculated.

When there are large state machines or various blocks of code are used by several state machines, it can make sense to logically separate these into smaller blocks. In this case the state machines are said to directly interact as there is no queue or buffering between them. Since the interaction here is purely one of convenience, and does not reflect two separately interacting entities, the critical path must follow the interaction.

Critical Path

When a new annotation is processed by the analysis program the critical path must be updated to reflect the edge and transition implied by the annotation. A critical path begins when a transition is processed that belongs to the state machine and system that the user

requested the critical path be found for. This process is accomplished as outlined in the following pseudo-code:

```
1  if dep = NIL or dep.cp = NIL
2     then if crit-path[sm].exists
3             then crit-path[sm].append(edge, weight)
4           elseif sm = begin_s m ▷ This state machine is the starting state machine
5             then crit-path[sm].start(edge, weight)
6  elseif dep .cp ▷ The interacting state machine has a critical path
7     then if not crit-path[sm].exists or
                 crit-path[sm].length + weight < dep.crit-path .length + comm-lat
8             then crit-path[sm] = dep.crit-path
9                  crit-path[sm].append(dep.edge, dep.weight)
10            else
11                 crit-path[sm].append(edge, weight)
12    else
13          crit-path[sm].append(edge, weight)
```

As illustrated in the code above, if the new node is only dependent on the previous node in the same state machine, then the critical path and the new node is the previous critical path length, plus the time spent in the current state. If no critical path exists at the previous node, no critical path can exist at the next node, except in the case in which the node is the first transition observed for this state machine and the state machine has been identified by the user as the state machine where the critical path calculation should begin. In that case, the critical path is created at the current node.

If an inter-state-machine dependence exists then the critical path at the current node is longer than the critical path at the two parent nodes plus the edge weight (in the case of the intra-state-machine edge) or the communication latency (in the case of the inter-state-machine edge).

Because the captured data is processed in order, at any given point there can be at most one critical path for every connection in every state machine. This fact allows the critical path to be found without building the entire graph of all interactions. For each state machine, a *crit-path* object stores the critical path from the start, including all processed and relevant records.

The Python code, which is used to find the critical path, makes extensive use of shallow copies in Python. To avoid the expensive copying that would have to occur every time a critical path from one state machine is merged with the critical path from another, an

innovative format is used. The critical path starts out containing a list of tuples of the edge, total length, and current length up to the current end of the critical path. Any time a merge happens and the critical path from a different state machine becomes the critical path in the current state machine, the list is emptied and a new critical path is created. The first element of this critical path is a tuple of the shallow copy of the incoming critical path and its current length when the shallow copy occurred. The current length is stored because between the time when the critical path is begun (by a enqueue or dequeue) and finally created (by a corresponding dequeue or enqueue), additional items could be appended to that portion of the critical path for that state machine. If other items were added it wouldn't be possible to know where the interaction occurred. After a merge, simple tuples of three items are added.

After all the data is processed this format can be turned into an array containing the entire path. This is done by walking the list in reverse, at the state machine of interest, appending each item found to a new list. Any time a shallow copy is found, the process follows the copy. Eventually, a critical path will be found that has no more shallow copies in it. When this occurs, the complete path has been found, and a huge number of deep copies have been eliminated. This is illustrated in the code below:

CONVERT-CRIT-PATH(cp)

1 $ncp \leftarrow$ COPY(cp)
2 $ncp.reverse()$
3 **while** IS-SHALLOW-COPY($ncp[last]$)
4   **do** $(jmpcp, jmplen) \leftarrow ncp.pop()$
5         $next\text{-}chunk \leftarrow$ COPY($jmpcp[:jmplen]$) $\triangleright$ Copy first $jmplen$ elements.
6         $next\text{-}chunk.reverse()$
7         $ncp.extend(next\text{-}chunk)$
8 $ncp.reverse()$
9 **return**ncp

It is worth noting that if the exact traversed path through the graph isn't required and only aggregate information is desired the same process can be applied; however, rather than keeping a list of encountered edges, a single aggregate edge is stored and the total time spent on that edge is incremented as nodes are processed.

5.1.5 Tracking Multiple Connections

The connection tracking information is used by the analysis program to provide two functions. It is used to understand which state machines interact together. In a system with multiple connections it can be useful to understand that state machine A on system X communicates with state machine B on system Y. This is done by appending the current connection ID to the data stored in the queue about each element. In this way connection information flows through the queues binding together state machines. Some state machines operate on a single connection, while others operate on many or all connections in the system. For example, in a system with only one NIC all connections must travel down the same part of the network. Second, the connection tracking enables the calculation of the correct critical path in a system with more than one connection.

Given a system that may contain multiple connections, it would not be sensible for one connection of a critical path to traverse a state machine that never handled that connection. To prevent this from occurring, the connection tracking must know which state machines belong to a connection and which do not.

Broadly speaking, when a simulation begins, no state machines have connection information. After a state machine uses an identify annotation to identify itself as a connection, the connection information is then passed to other state machines the initial one interacts with. This is done both through the queues and the non-queue state machine linking. Those state machines, in turn, continue to pass the connection information to the state machines they interact with and thus eventually all state machines of interest in the system have connection information in them.

There are some cases in which the queue should not pass connection information. This occurs when a state machine reuses an allocated resource, rather than creating a new one. For example, the device driver may reuse NIC descriptors that were recently freed. However, in this case, the connection that previously used the descriptor and the connection that will use the descriptor may not be related. Thus, the state machine should not assume the old connection of the descriptor when it is dequeued. This can be done one of two ways. A flag can be used on the queue operations to inform the analysis program that this is occurring, or the names of queues that shouldn't pass connection information can be otherwise stored. The latter approach is currently used in the analysis program.

There is one more case in which the queue can not blindly pass connection information between state machines. These cases occur when a state machine initializes a buffer of elements before it has received a connection ID. A good example of this case is the TCP send window. Initially, some number of bytes are available in the window, and thus they are queued when the connection is set up; however, all these queues don't have a connection

ID. Anytime this occurs the state machine would return to an unknown connection ID. Thus, there are certain queues that should not reset the connection ID if they do not have one. These queues are limited to queues that only interact with a single connection, such as those that represent TCP windows. These queues are easy to recognize because the connection information will never flow out of these state machines.

Critical paths are only created for state machines that have connection information. Otherwise, it would be impossible to enforce the restriction that one connection's critical path can never traverse state machines that do not handle data for that connection. When a state machine interacts with another state machine, that state machine's critical path for the current connection ID is merged with the incoming critical path, choosing the longest, as described above.

5.2 Critical Path

The critical path analysis simply prints the critical path from beginning to end for each connection. Along with the name of the edge, the cumulative time to the edge currently being printed, the additional time the current edge added, and the amount of time the current edge was executing but not active (another state machine like an interrupt must have been active) is printed. Figure 5.2 provides an example of the data provided by this analysis. Non-active time is not shown in the figure due to space constraints.

5.3 Most Critical States

The critical path analyzed in the previous section can be used to see what interactions are traversed between the starting and ending state machines of interest. However, as it grows in size, it becomes difficult to interpret the delays along the critical path quickly. When this occurs a summary of the most important states along the critical path is desired. The most critical states analysis accomplishes this by calculating the percentage of time each state in the critical path contributes to the total critical path length. While every edge in the critical path is distinct, the states that the edges correspond to are more than likely repeated. This task is easily accomplished by summing the time spent in each distinct state and dividing by the total length of the critical path. Generally, the analysis program prints any state that corresponds to more than 1% of the critical path length.

An example of this analysis is presented in Figure 5.3. In this particular experiment the system is bound by the CPU on the transmitting system. The majority of the critical path

```
Name                                                                  Cumulative(ns)  Inc(ns)
---------------------------------------------------------------       -----------     -------
testsys:Ip Send_0xc1880:__sched_text_end
                                                                      2512             5
testsys:Ip Send_0xc1880:dev_queue_xmit
                                                                      2515             3
testsys:Ip Send_0xc1880:pfifo_fast_enqueue
                                                                      2523             8
testsys:Ip Send_0xc1880:Queue->testsys:Device TX_0x86e80:Peek[DeviceTxQ0xe86e80]
                                                                      2558             34
testsys:Device TX_0x86e80:Peek__DeviceTxQ0xe86e80
                                                                      2563             5
testsys:Device TX_0x86e80:dev_hard_start_xmit
                                                                      2569             5
testsys:Device TX_0x86e80:e1000_xmit_frame
                                                                      2609             40
testsys:Device TX_0x86e80:Dequeue__TX Free Descriptors0x2
                                                                      2652             43
testsys:Device TX_0x86e80:Queue__TX Unused Descriptors0x2
                                                                      2655             2
testsys:Device TX_0x86e80:pci_map_single
                                                                      2655             0
testsys:Device TX_0x86e80:pci_dac_dma_supported
                                                                      2656             0
testsys:Device TX_0x86e80:pci_map_single
                                                                      2656             0
testsys:Device TX_0x86e80:pci_map_single_1
                                                                      2657             1
testsys:Device TX_0x86e80:e1000_xmit_frame
                                                                      2695             37
testsys:Device TX_0x86e80:Dequeue__TX Free Descriptors0x2
                                                                      2696             1
testsys:Device TX_0x86e80:dprintk
                                                                      2706             10
testsys:Device TX_0x86e80:Queue->testsys:TX Desc Fetch_0x2:Peek[TX Unused Descriptors0x2]
                                                                      2706             0
testsys:TX Desc Fetch_0x2:Peek__TX Unused Descriptors0x2
                                                                      2706             0
testsys:TX Desc Fetch_0x2:Resv__TX Descriptors0x2
                                                                      2706             0
testsys:TX Desc Fetch_0x2:Prepare Fetch Desc
                                                                      2756             50
testsys:TX Desc Fetch_0x2:Fetch Desc
                                                                      9145             6389
testsys:TX Desc Fetch_0x2:Fetch Complete
                                                                      9145             0
testsys:TX Desc Fetch_0x2:Dequeue__TX Unused Descriptors0x2
                                                                      9145             0
```

Figure 5.2: Example of critical path analysis.

```
Most critical edges (time spent) on connection id 1 critical path:
------------------------------------------------------------------
testsys:TcpSendMsg_0xbd880:do_softirq              01.33 (99.58% waiting)
testsys:TXQ_2:Queue->drivesys:RXQ_1:Dequeue[WireQ] 02.01 (00.00% waiting)
testsys:TcpSendMsg_0xbd880:__kmalloc               02.35 (47.67% waiting)
testsys:TcpSendMsg_0xbd880:tcp_sendmsg             02.58 (17.51% waiting)
testsys:TcpSendMsg_0xbd880:__alloc_skb             05.11 (29.12% waiting)
testsys:TcpSendMsg_0xbd880:release_sock            12.74 (99.82% waiting)
testsys:TcpSendMsg_0xbd880:__copy_user             34.62 (31.82% waiting)
testsys:TcpSendMsg_0xbd880:tcp_push_one            34.75 (99.72% waiting)
```

Figure 5.3: Example of most critical analysis.

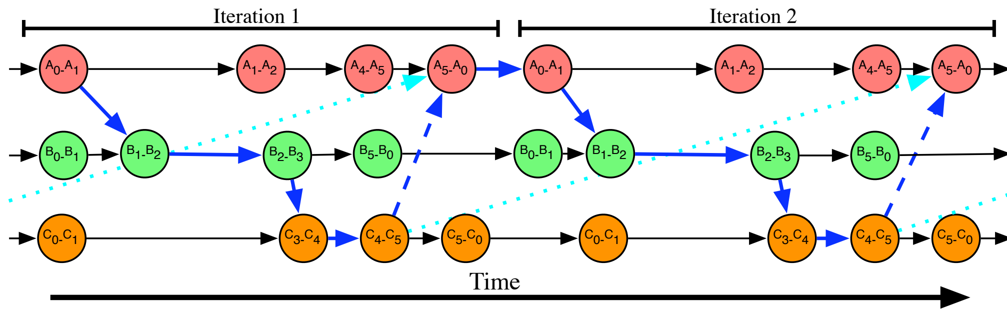


Figure 5.4: Illustration of a resource dependence loop.

is composed of by the `tcp_push_one()` function that is mostly waiting as that function instructs lower level TCP and IP state machines to transmit packets and `__copy_user()` that does the user-to-kernel copy of the payload. In this case this function is interrupted by the NIC interrupting the CPU to handle TCP acknowledgements.

5.4 Resource Dependence Loops

Although the most obvious way to eliminate a critical path is to reduce the latency of the operations it contains, in some situations the critical path reflects a resource constraint and can be removed without improving any operation latencies. This ability is particularly useful when a feature that is out of the designer's control, such as the network latency, is the dominant component of the critical path. I have developed an automated technique for identifying these situations in my analysis.

In resource-constrained environments, the critical path forms a repeating pattern, moving from one state machine to another and eventually returning to the initial state machine, thus creating a "loop." (Although the critical path is acyclic, it is often visualized by using a representation in which different iterations of the same state machine are folded onto the

```
Most critical edges (resource loops) on critical path 1:
----------------------------------------------------------------------
25 testsys:TcpRcvProcess_0xc1880:Queue->
   testsys:TcpSendMsg_0xc1880:Dequeue[sndmemQ0x4c1880]
```

Figure 5.5: Example of resource dependence loop analysis.

same graph nodes, causing these edges to form a loop.)

Figure 5.4, illustrates a portion of a hypothetical dependence graph composed of three interacting state machines: A, B, and C. A transfers data through B to a buffer managed by C, where it is consumed. C returns credits indicating the availability of buffer space to A. If only one buffer is available at C, A must wait for a credit before each transfer. The solid bold arrows in the figure indicate the true data dependence, while the dashed arrows from $C_4 \rightarrow C_5$ to $A_5 \rightarrow A_0$ indicate A's dependence on C's flow-control credit and complete the loop back to A. Assuming a unit weight on each edge, the path following the bold lines from $A_0 \rightarrow A_1$ of iteration 1 (in the upper left) to $A_5 \rightarrow A_0$ of iteration 2 (in the upper right) will be the critical path at a latency of 11 units.

If a second buffer is added to C, A can now send two units of data before requiring a credit from C. This change modifies the dependence graph; $A_5 \rightarrow A_0$ is now dependent on the instance of $C_4 \rightarrow C_5$ in the previous iteration rather than the same iteration. In the figure, the dotted arrows replace the bold dashed arrows. The weight of the path from $A_0 \rightarrow A_1$ of iteration 1 through C and back to $A_5 \rightarrow A_0$ of iteration 2 is now 5 units, and the critical path between these nodes has shifted to the 7-unit path internal to A.

My analysis program automatically identifies these resource loops and presents them to the user. The resource constraints I have identified with this technique include the NIC device's DMA descriptor rings, kernel socket buffers, and TCP window size parameters.

An example of the output provided by this analysis is shown in Figure 5.5. In this case the transmitting system is limiting bandwidth on the link because not enough memory is available to buffer transmitted packets in the socket until acknowledgements are received.

5.5 Predicting Speed-ups

All the state machine transitions and interactions that occurred—not just the ones that compose the critical path—are contained in the dependence graph. Once the critical path is identified, a user can reduce the weights on the edges in that path to identify the next most critical path, and rerun the extended analysis on the recorded trace to find a whole series of bottlenecks. The lengths of these additional critical paths relative to the original

```
drivesys:Device TX_0x16c80(270):          set([1, 2, 3, 4])
drivesys:TXQ_0x1(271):                    set([1, 2, 3, 4])
testsys:RXQ_0x2(272):                     set([1, 2, 3, 4])
testsys:RX Desc Writeback_0x2(273):       set([1, 2, 3, 4])
testsys:Ip Recv_0x38440(274):             set([1, 2, 3, 4])
drivesys:TcpRcvProcess_0xa3880(275):      set([2])
drivesys:TcpRcvProcess_0xa3280(297):      set([1])
testsys:TcpRcvProcess_0xe3280(276):       set([1])
testsys:TcpSendMsg_0xe3280(281):          set([1])
testsys:TcpOutput_0xe3280(282):           set([1])
```

Figure 5.6: Example of connection tracing analysis.

critical path quantify the potential speedup that can be achieved by eliminating an existing bottleneck. These additional critical paths can be extracted in a few minutes, and do not require the user to have any idea how a particular bottleneck could be eliminated in practice. In contrast, even with a simulator, measuring the speedup of bottleneck elimination directly would require determining a parameter adjustment or another modification that would eliminate the bottleneck, prototyping that modification, and finally multiple hours of running the simulation. Additionally, the resource dependence loops described above can be eliminated by disallowing the critical path to traverse the edge identified by the resource loop analysis. This, in turn, predicts the performance of the resource dependence being completely removed by providing additional resources so it is no longer the bottleneck.

5.6 Connection Trace

This analysis option prints a list of all the state machines seen in the system and the connection IDs that each state machine handled. It can be used to debug any connection tracking issues, and provides a quick way to see which resources are shared and which are allocated per connection.

An example of this analysis is shown in Figure 5.6. The first five state machines listed belong to the device, driver, and receive processing portions of the system. Since there is only one network interface in this experiment, all of the four present connections traverse these state machines. On the other hand, the last four state machines are connection specific. Two instances of the TcpRcvProcess state machine are shown, one for connection 1 and another for connection 2. Finally, three more state machines all belonging to connection 1 are shown.

```
testsys:TcpSendMsg_0xbd880                                          6031us (98%)
-- testsys:TcpSendMsg_0xbd880:__copy_user                           4314us
-- testsys:TcpSendMsg_0xbd880:__alloc_skb                            661us
-- testsys:TcpSendMsg_0xbd880:tcp_sendmsg                            391us
-- testsys:TcpSendMsg_0xbd880:__kmalloc                              224us
-- testsys:TcpSendMsg_0xbd880:kmem_cache_alloc                       180us
-- testsys:TcpSendMsg_0xbd880:loop                                    40us
-- testsys:TcpSendMsg_0xbd880:release_sock                            35us
-- testsys:TcpSendMsg_0xbd880:Dequeue__sndmemQ0x4bd880                34us
-- testsys:TcpSendMsg_0xbd880:Queue__skSendQ0x4bd880                  27us
-- testsys:TcpSendMsg_0xbd880:cache_alloc_refill                      23us
```

Figure 5.7: Example of state machine information.

5.7 State Machine Information

The information recorded for the analysis is a superset of the information that would be recorded by a profiler. As such, the analysis program is able to print out information about each software state machine, how long it was active, and what the top N functions are in that state machine. This analysis is particularly useful when a system is CPU bound. If it is not possible to easily address any of the most critical states on the critical path, providing an additional CPU may improve performance.

For example, in Figure 5.7 the `TcpSendMsg` state machine for a connection is shown. This one state machine occupied a little over 6ms of total CPU time during the detailed experiment that was used to generate it. Out of the 6ms this state machine was active for 98% of the time, which is captured by the top 10 functions shown below. The most CPU time was used by the `__copy_user` function which used 4.3ms.

5.8 Queue statistics

A great deal of information is collected about the queue while the analysis is running. For each queue in the system, a page of histograms can be printed that show the occupancy, the time state machines were waiting for space or data, communication latency, service time, and arrival rate.

The statistics for the receive descriptor queue of the NIC are shown in Figure 5.8.

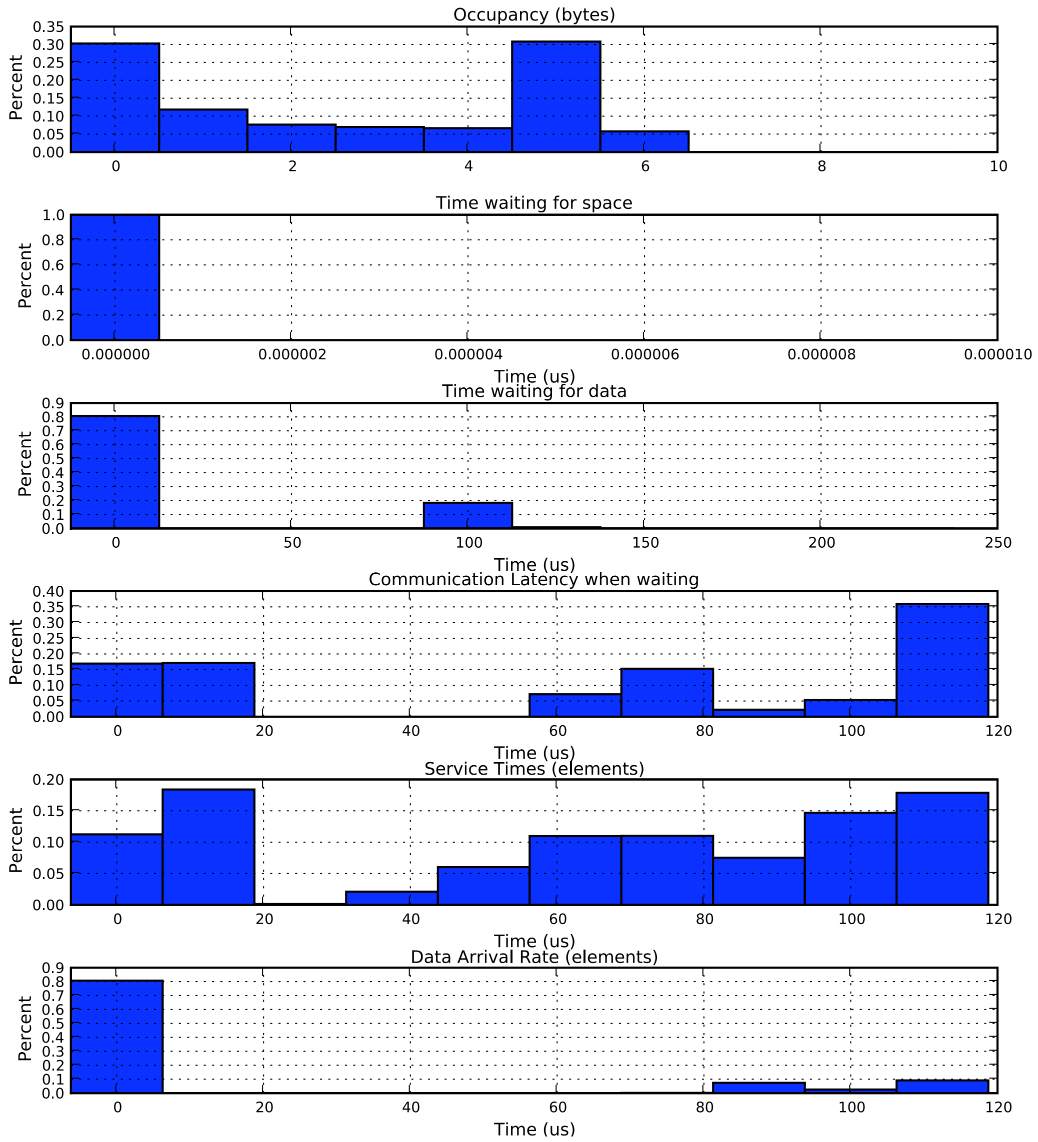


Figure 5.8: Example of queue statistics output.

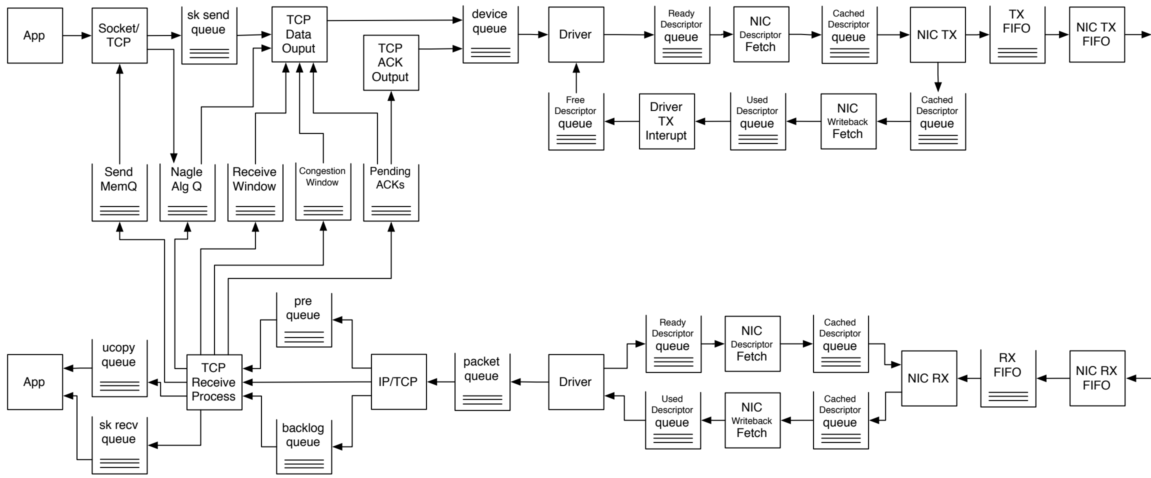


Figure 5.9: Graph of state machines and queues in Linux TCP/IP stack from NIC to application.

5.9 Compact Graph Output

Graphical output can be useful to visualize the state machines, states, and interactions. However, as mentioned earlier, the number of nodes present in a dependence graph can easily exceed a million. Over 100 sheets of paper would be required to draw only 10,000 nodes, making visualization impossible. I have devised a number of graph outputs that limit the nodes and edges while still containing a rich amount of information. These formats are described below.

5.9.1 High-level Interaction

Another visualization option is to graph all the state machines and queues in the system at a high level. This graph helps the user visualize which state machines are operating on which queues and the end-to-end flow of the system from a high level. For example, the graphic in Figure 5.9 contains automatically generated data.[2]

5.9.2 Global State Machine Graph

The graph analysis option prints a large graph of all the state machine and their interactions. While the above techniques can quickly provide considerable data on the graph structure for analysis, it's also useful to visualize the graph to see how and where various state machines interact. Unfortunately, as mentioned previously, the graph is far too large

[2]The images and links in the graph were reformatted for easier presentation as the analysis output simply creates circles for state machines and squares for queues, and uses graphviz to lay out the nodes and arcs.

to visualize when the number of nodes exceeds a few thousand.

To cope with this problem, I created a graphical model that is a hybrid between a canonical state machine graph and the dependence graph described in Chapter 4. Like the dependence graph, nodes in the graph represent transitions and edges represent states; however, there is only one node for each transition regardless of the number of times that transition occurs at runtime. These transformations result in a graph that is much easier to visualize and work with.

The nodes in this graph are labeled with the system they belong to and the transition that is taking place. Edges are labeled with a state name and three numbers corresponding to the number of entrances into that state, the time spent in that state, and the time on the critical path spent in that state. The graphs are generated per connection and all the nodes belonging to a single state machine have the same color. The edges are colored based on the critical path. The red and blue components of the edge's color are proportional to the most critical edge and the most traversed edge, respectively. Because of the limitations of the number of nodes that graphviz/dot can reasonably process at one time, the output is limited to approximately 1500 nodes. The number of nodes in the graph is further reduced by replacing straight line interactions with a single super-edge that represents multiple transitions and states. Additionally, rarely traversed paths can be pruned from the graph if it is still too large.

There are several graph invariants:

1. For every node in the graph, the sum of the entrance count of the outgoing edges must be equal to the sum of the entrance count of the incoming edges.[3]

2. The time on the critical path in the node can not exceed the time spent in that node.

3. For every system and state machine, there exists only one node for any state transition $A \rightarrow B$.

4. For any edge, there can be more than one edge corresponding to the same state. Multiple state transitions can end up in state B, so an edge corresponding to state B must be able to appear multiple times.

5. Inter-state-machine edges (dashed lines) do not represent any time spent executing in a state machine but instead describe interactions between state machines. When a state machine A reaches state X in which it is waiting for another state machine B to reach state Z before it can continue, an inter-state-machine edge is placed between

[3]This count can be off by one due to the window in which data was recorded.

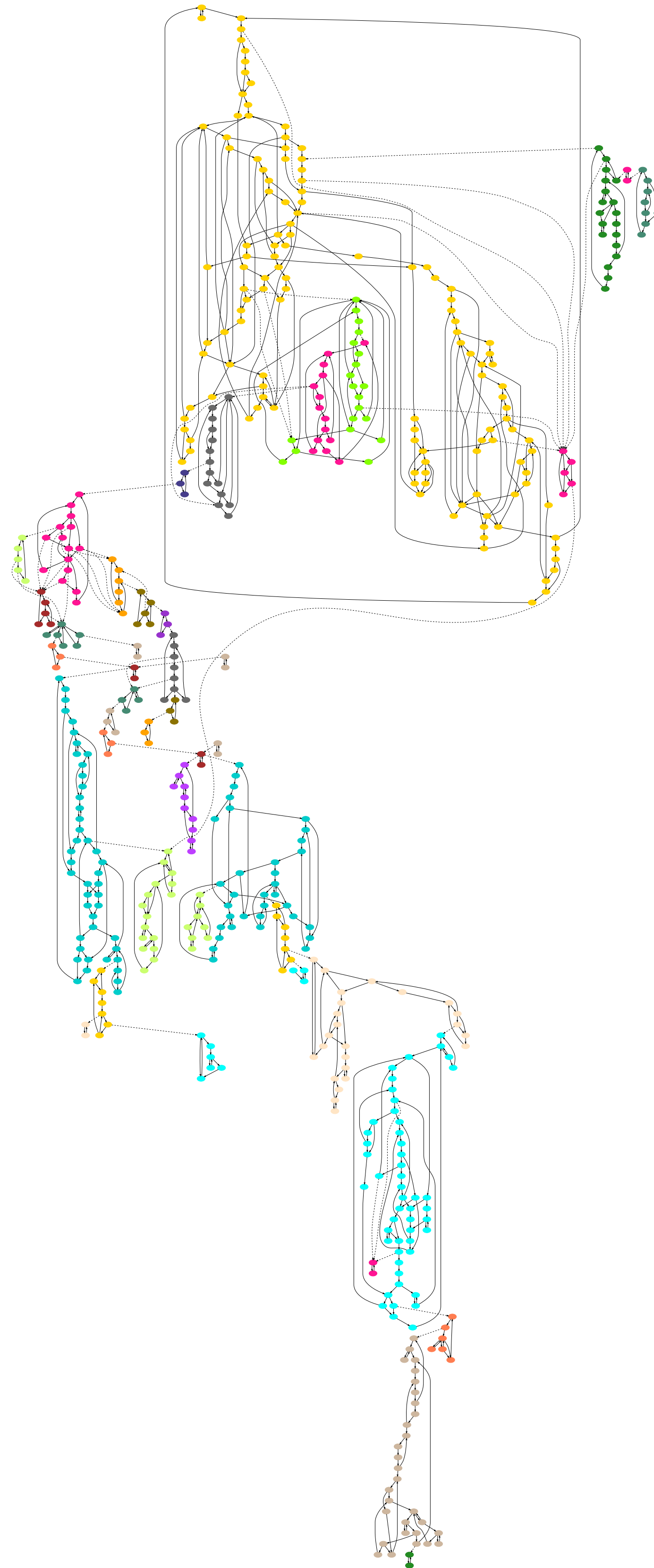


Figure 5.10: Example global state machine graph generated by the analysis program.

the successor of state X in state machine A and the node in state machine B that begins state Z.

An example of the graph is presented in Figure 5.10. All the text that would normally be present in the graph has been removed for clarity. When displayed on an interactive terminal the user would be expected to zoom into portions of the graph that are considered interesting.

5.9.3 Single State Machine Output

Instead of visualizing the entire graph at once the analysis program can output a single state machine and the interactions it had with other state machines. This graph is much smaller than the global graph described above, so it can be generated and manipulated in a fraction of the time. However, it provides the same information contained in the global state machine, albeit one state machine at a time. An example of this graph generated for the transmit portion of the Ethernet device is presented in Figure 5.11.

5.10 Conclusion

This chapter has described the methods used to process the data provided by the simulator. In doing so, a useful representation is created that can be used for analysis and can bypass directly creating the entire graph in memory, which would entail large space requirements. Ultimately, further analysis is required to extract the useful information in the graph and provide it to the user in an useable form. Several analyses are presented that accomplish this task, such as the most critical states and resource dependence loops analysis. After processing up to hundreds of millions of nodes, the analysis program provides a few hundred lines that describe the most critical portions of the system. It can then be used to predict the performance that may be attained if those critical portions are modified in some way.

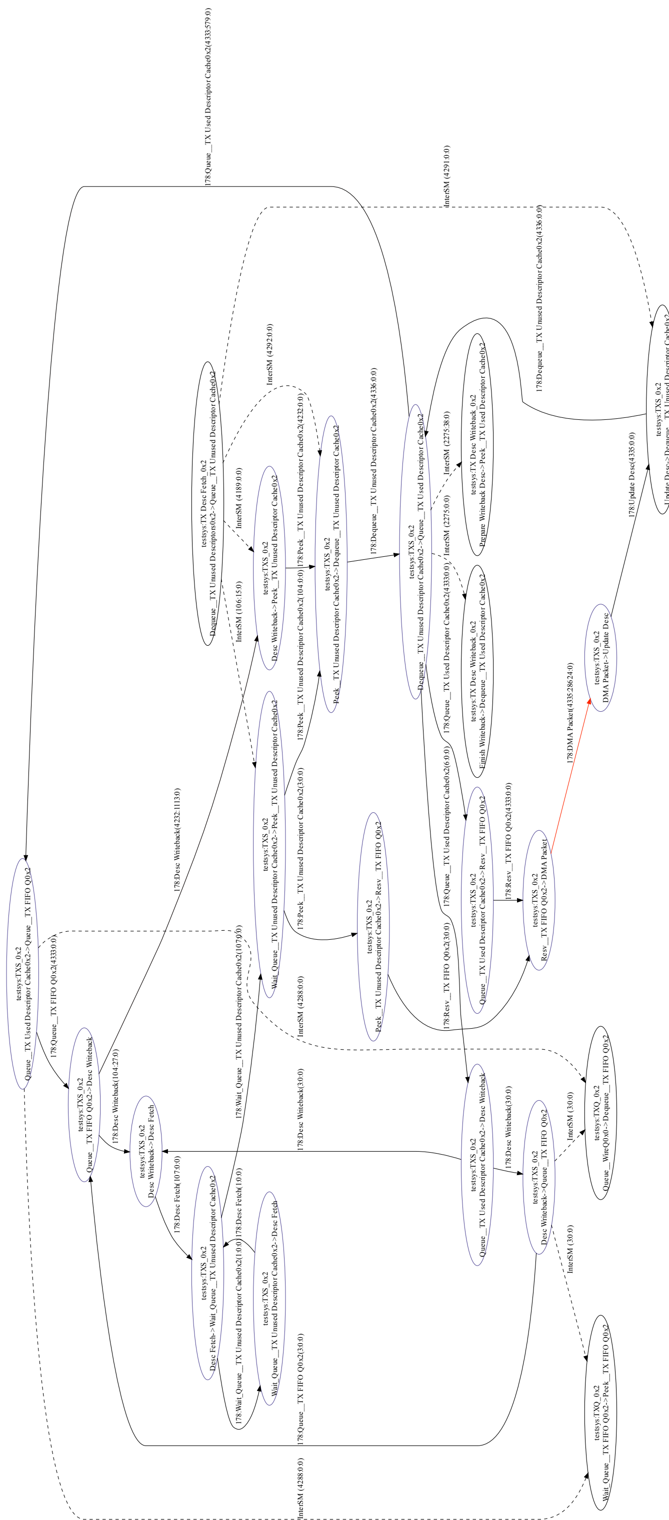


Figure 5.11: Example single state machine graph generated by the analysis program.

Chapter 6

Application of Analysis

In this chapter, the technique and tool from the previous chapter are applied to find bottlenecks in systems and predict performance after the bottlenecks are removed. With this methodology, bottlenecks can be detected at all levels of the system, spanning both hardware and software.

This chapter begins by looking at a simple UDP workload to show that the methodology described previously can identify a hardware bottleneck and locate a bug in the Linux 2.6.13 kernel. Later in the chapter, TCP based workloads are analyzed and, finally the overheads involved in a copy-engine are discussed. TCP, as described in Chapter 2, is more complex protocol that, unlike UDP, provides reliable, in-order delivery. This technique described in the previous two chapters is able to find a sequence of bottlenecks and to accurately predict the resulting performance after the bottlenecks are removed. From beginning to end, the tool predicts an over 300% performance improvement from the baseline system with only a single detailed experiment. These predictions are then validated with additional simulation, and are found to be very accurate.

After analyzing a TCP streaming workload, the analysis moves to a more complex web server workload that utilizes multiple connections. Bottlenecks in the web serving workload are located, the performance without the bottlenecks predicted, and the predictions are validated.

Finally, the analysis returns to the TCP streaming workload. This time, the receive side is analyzed and a the overheads involved in a DMA copy-engine are enumerated.

6.1 Simulation and Methodology

The M5 simulator [25], described in Chapter 3 is used to execute annotated software and hardware models. For this work, I used the default M5 parameters with a few exceptions.

All simulated CPUs operated at 4 GHz and the I/O bus bandwidth was originally set to that of a single (x1) PCIe lane, then changed (as described below) to the bandwidth of a PCIe x4 channel. The I/O bridge latency was set to 100ns. All experiments involve two systems connected with a single simulated Ethernet link, with a link delay of 350μs for the streaming experiments and 500μs for the web server experiments.

In each experiment, M5 simulates the systems' functionally as they boot a Linux 2.6.13 or 2.6.18 kernel and invoke the benchmark. Once the benchmark execution stabilizes, a checkpoint is created. The simulation is restored from that checkpoint into a configuration in which the server (the system under test) has a detailed CPU model and timing memory system, while the client continues to execute functionally. The client's effective performance is scaled so that it is not the bottleneck. The detailed simulation is run for a simulated 100ms, with annotations recorded after a 20ms warm-up period.

The critical path analysis is begun by choosing suitable starting and ending state machines. For the streaming workload, the path between the client and server applications is measured. Similarly, for the web server workload the path between the request and response components of the client application is measured. The critical path is determined from the first time the starting state machine is entered to the last time the ending state machine is entered within the simulation window. I use the term "most critical state" to indicate the state in which the critical path spends the largest fraction of its time. The critical path indicates only the latency bottleneck, but for large transfers (much larger than the network's bandwidth-delay product), latency is inversely proportional to bandwidth. Thus, by simulating sufficiently long intervals, the analysis program can predict the increase in bandwidth caused by a change in the dependence graph by calculating the reciprocal of the percent decrease in critical path length.

6.2 Streaming Benchmark Analysis

First, a streaming benchmark is analyzed. These benchmarks are frequently used for performance analysis on real hardware and in simulation because they quickly provide a uniform, steady-state measurement of the bandwidth between the two systems.

6.2.1 UDP Streaming Benchmark

The first protocol that is analyzed is UDP, which is described in Chapter 2. The analysis program is able to identify hardware and software bottlenecks in this workload, something that a profiler would be unable to do.

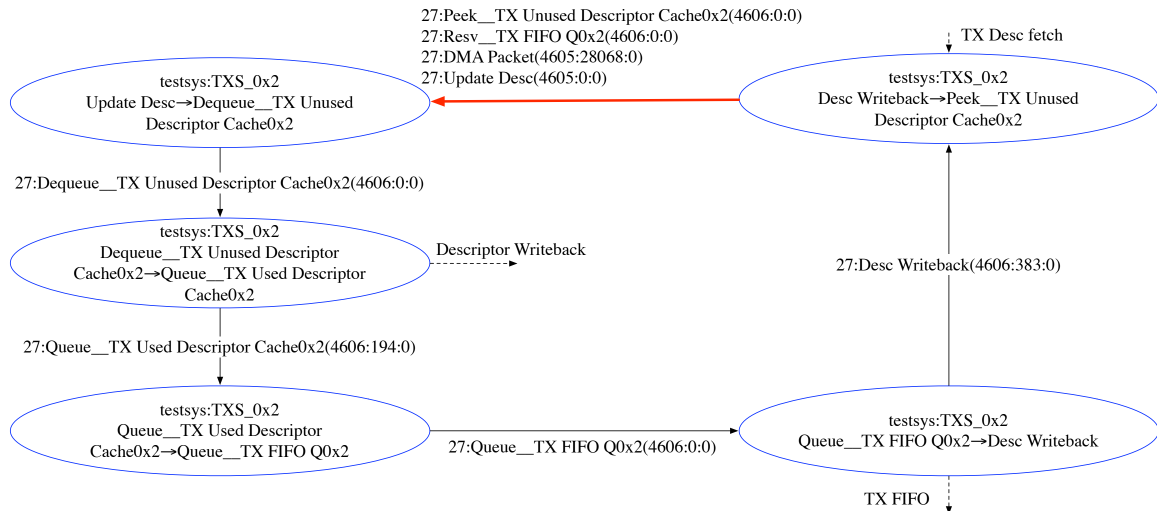


Figure 6.1: NIC Transmit State Machine – Global bottleneck graph

Hardware bottlenecks

In the first experiment, a Linux 2.6.13 kernel is utilized to send 16KiB UDP packets. The I/O bus is configured to be a standard PCI Express x1 bus. This bus provides 2 Gbps of peak I/O bandwidth. However, a fraction of this bandwidth is lost to bus arbitration. An additional portion of the bandwidth is consumed by the NIC to manage DMA descriptors: For each DMA transfer of actual network data, the NIC must read the descriptor corresponding to that buffer, then update and write back the descriptor to mark it as processed. In the case of an Intel e1000 compatible NIC these descriptors are normally 16 bytes long. Finally, data may not always be ready to transfer when the bus is free, introducing idle cycles that further reduce the effective bus bandwidth.

Simulating this configuration resulted in a bandwidth of 1.8 Gbps on the link. The critical path analysis tool identified the NIC transmit and the NIC TX Descriptor Fetch state machines as the primary bottleneck. The critical path spent 57% of the time in the "Wait" state in the TX Descriptor Fetch state machine—a state that waits for commands from the Transmit state machine—and 39% of the time was spent in the "DMA packet" state of the Transmit state machine.

The transmit state machine portion of the combined graph for this experiment is shown in Figure 6.1. All other state machines have been removed from the graph, and rarely traversed paths of the transmit state machine have also been omitted for legibility. As discussed in Section 5.9.2, each edge of the cgraph is labeled with the name of the corresponding state in the original state machine and a 3-tuple indicating the number of visits to that state, the total time spent in the state, and the time spent in that state on the critical

path.

The state machine operates by reading a locally cached descriptor, executing the action specified (DMAing a packet), queuing the packet for transmission, updating the descriptor, and repeating. The state machine interacts with three other state machines (dotted lines): Descriptor Fetch, Descriptor Writeback, and TX FIFO state machines. The analysis clearly indicates that the “DMA Packet” state is the most critical.

To verify this, the simulated system is modified, replacing the PCI Express 1x link with a 4x link and re-running the experiment. The new bus has four times the bandwidth of the original bus, so it should no longer be the bottleneck. Upon rerunning the experiment—only changing the I/O bus—3.3 Gbps of bandwidth is observed on the link. The most critical state was no longer in the NIC’s transmit state machine (and not in the NIC at all).

This I/O bus bottleneck would not be easily observed via software profiling, even given detailed visibility into the kernel code. The profiler would see various kernel components working steadily, since packets are still being sent at a fast rate. The only kernel indicator that the DMA is a bottleneck would be the number of free DMA descriptors the driver has available and the size of the device queue. However, an execution-time profiler would not provide any insight to the size or occupancy of these structures.

Software and configuration bottlenecks

The analysis of the UDP workloads continues by looking at the software bottleneck that is uncovered after replacing the PCI Express x1 bus with a x4 configuration. Here again, the simulated system is still using a Linux 2.6.13 kernel and transmitting a 16KiB payload. Since the 16KiB payload is larger than the MTU of 1518 bytes, it must be fragmented by the IP layer. This is generally preferable to sending smaller UDP packets that don’t need to be fragmented because the smaller packets require the application to perform more system calls to send the same amount of data. These differences result in different paths through the kernel and somewhat different critical paths.

The analysis program produced some unexpected results when analyzing the data from the UDP experiment. When compared to other protocols, the UDP workload is simple and one would expect the majority of the CPU time to be the user-to-kernel copy. However, the results presented in Table 6.1 do not show this to be the case. A great deal of time is spent waiting for the other state machines in the system to process the packet.

In an effort to resolve why so much time is spent waiting in the `ip_push_pending_frames` state, the analysis program is instructed to remove the time spent in the `ip_push_pending_frames` state and print the the next most critical path. The results of this are presented in Table 6.2. The most critical non-waiting state, `ip_defrag`, defrag-

State	Criticality
testsys:Sys Send:ip_push_pending_frames	53.73% (99.96% waiting)
testsys:Sys Send:do_csum_partial_copy_from_user	30.69% (09.10% waiting)
testsys:Sys Send:alloc_skb	4.77% (12.41% waiting)

Table 6.1: Most critical states Linux 2.6.13 w/netfilter

State	Criticality
testsys:Ip Send:qdisc_restart	32.57% (98.45% waiting)
testsys:Ip Send:ip_defrag	6.26% (09.98% waiting)
testsys:Ip Send:memcpy	5.97% (09.80% waiting)
testsys:Ip Send:_read_unlock_bh	04.68% (06.99% waiting)
testsys:Ip Send:do_softirq	3.94% (99.36% waiting)
testsys:Ip Send:ip_copy_metadata	3.74% (17.61% waiting)
testsys:Ip Send:ipt_do_table	3.60% (11.12% waiting)
testsys:Ip Send:ip_fragment	3.60% (07.88% waiting)
testsys:TXQ:Queue->drivesys:RXQ:Dequeue[WireQ]	3.17 (00.00% waiting)
testsys:Ip Send:nf_iterate	2.45% (17.05% waiting)
testsys:Ip Send:dev_queue_xmit	2.30% (03.43% waiting)

Table 6.2: Next most critical states Linux 2.6.13 w/netfilter

ments IP packets. However, the system under test is the transmit side, which is expected only to fragment packets. In fact, six rows below the `ip_defrag` state there is an state named `ip_fragment`, which (with the help of some of the states in between) fragments packets that are larger than the MTU. Looking at the global state machine graph or the critical path confirms that the kernel is fragmenting the packet to be sent down the wire and then almost immediately reassembling the fragments to pass them to the netfilter. This order results in a large overhead, and is an error in the code; the netfilter should be invoked before the packet is fragmented. This problem was fixed in Linux 2.6.16, something that I was unaware of when I began running experiments, but sought out after I found this surprising result.[1]

This bug can be verified by running the same experiment with a newer Linux kernel. Performance increases substantially. As such, all future experiments described in this Chapter use Linux 2.6.18. Unlike the PCI bottleneck described above, these netfilter bottlenecks are theoretically visible to a kernel profiler. However, the user-to-kernel copy function is still by far the dominant item on a function-based profile, consuming roughly five times more cycles than `ip_defrag`. To discover this bottleneck with a profiler, a user would have to realize that the UDP and IP stacks form a pipeline, then sum the time spent

[1] See http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;-h=1bd9bef6f9fe06dd0c628ac877c85b6b36aca062.

in UDP functions and in IP functions separately to determine that the aggregate IP processing dominates. This function categorization is not obvious, as it cannot be performed solely based on function names. In contrast, the implicit-state-machine approach automatically assigns functions to the appropriate state machine once the boundary between state machines is properly identified and can even cope with a single static function being called from multiple state machines. Thus, this technique provides a much more direct, reliable, and automatic identification of the bottleneck.

6.2.2 TCP Streaming Benchmark

This subsection analyzes a TCP streaming benchmark developed at CERN called GenSink [41]. The GenSink benchmark is composed of two applications: (a) a generator that sends data as quickly as possible over a single TCP/IP connection and (b) a sink that receives the data and discards it. Both applications use blocking system calls for network I/O. In these experiments, the generator system is the system under test and is simulated in detail, while the sink is modeled only functionally.

The analysis begins with a detailed simulation of the benchmark on a baseline configuration (configuration 1). The analysis tool described in Chapter 5 is used to process the trace generated by the annotations in this run to locate the bottleneck in the system. The tool is then re-run, modifying the critical path calculation to model a hypothetical optimization removing that bottleneck, to generate a new critical path. The length of this new critical path predicts the performance of the optimized platform (configuration 2), and the path itself predicts the bottleneck in that platform. The tool is then re-run a third time, including both the prior modifications and a new set of modifications that address configuration 2's predicted bottleneck, so as to generate a new critical path and performance prediction for the further optimized system (configuration 3). Finally, the tool is run a fourth time, producing critical-path and performance predictions for a further optimized configuration 4. To validate the tool's predictions, detailed simulations of configurations 2–4 are then run. The experimental results are presented in Figure 6.2.

The baseline configuration (configuration 1) sets the link bandwidth between the systems to 1 Gbps. The simulated systems achieve 1 Gbps, as shown by the dashed line in Figure 6.2. The analysis program indicates that the critical path is through the NIC's TX FIFO state machine, which removes packets from the outbound FIFO buffer and places them on the link. This result indicates that the link is indeed the bottleneck.

For configuration 2, I decide to address this bottleneck by increasing the link bandwidth from 1 Gbps to 10 Gbps. This optimization should reduce the time spent transmitting on

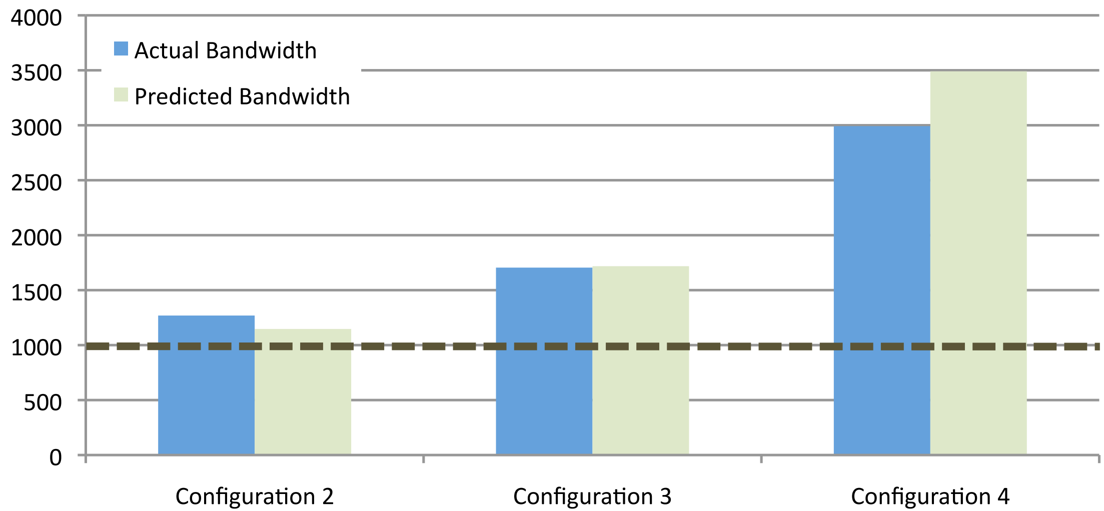


Figure 6.2: Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 1. Dotted line is bandwidth produced by initial configuration.

the Ethernet link by 90%, so I instruct my analysis tool to recompute the critical path based on the configuration 1 dependence graph but with this modification. The tool predicts that the critical path length would be reduced by 13%, which would increase the bandwidth of the system to 1147 Mbps. This is shown in the leftmost bar group in Figure 6.2.

Examining the predicted critical path itself, I see that it goes back and forth between state machines in the generator and sink systems, with the majority of time being the communication latency between the systems (the Ethernet link latency). In the real world, this network latency is typically out of the system designer's control. However, the resource dependence loop analysis, described in Section 5.4, identifies the queue that represents the TCP send window to be the bottleneck. The TCP send and receive windows indicate the buffer space on both sides of a TCP connection. Sender buffers are required to store copies of all packets in case a re-transmission is required. Receiver buffers provide enough space to buffer all the bytes in flight from the sender. When the buffers are smaller than the bandwidth-delay product of the connection, they can limit performance. The analysis tool predicts that for configuration 2, these buffer sizes are the bottleneck.

I then consider a configuration 3, which addresses this bottleneck by increasing these buffer sizes. I instruct the analysis program to break the TCP resource dependence loop that is limiting performance by ignoring the edge that completes the loop in the critical-path calculation. With this change, combined with the 10 Gbps link modification for configuration 2, my tool predicts the new bandwidth to be 1718 Mbps. This result is shown in the figure as the first bar in the "Configuration 3" group.

The critical path from this analysis run is dominated by the NIC state machine that DMAs packets from main memory, indicating the I/O bus bandwidth is the bottleneck. I propose a configuration 4, which removes this bottleneck by increasing the I/O bus bandwidth by a factor of 4, and instruct my analysis program to model this new system by reducing the time spent in the DMA states by 75% (in addition to the previous modifications). The length of the resulting predicted critical path indicates a bandwidth of 3491 Mbps (the first bar in the "Configuration 4" group). The critical path itself identifies the user-to-kernel buffer copy as the bottleneck. Because this copy takes place in software, this result indicates that the benchmark is now CPU bound.

Each of the proposed configurations was simulated in detail and used to verify the performance predictions. The detailed simulation of configuration 2, increasing the link bandwidth to 10 Gbps, produced a bandwidth of 1289 Gbps (the first bar group), within 9.6% of the predicted performance. The error in the performance projection is due to a change in the behavior of the transmit state machine.[2] When a change in the system produces a qualitative change in the way state machines behave or interact, the analysis tool's prediction cannot account for the impact of those changes since they are not captured in the original dependence graph. Nevertheless, the tool did identify the correct bottleneck, and the predicted bandwidth is still close enough to be useful even in the face of minor changes in system behavior.

To validate the prediction for configuration 3, the TCP stack parameters were modified and another detailed simulation was done. [3] The measured bandwidth increases, validating the prediction of the bottleneck. Furthermore, the new bandwidth is 1705 Mbps (middle bar group of Figure 6.2), within 1% of the predicted value.

Finally, configuration 4 is simulated, which increases the I/O bus bandwidth by a factor of four. The measured bandwidth increases to 2992 Mbps (right-most bar group), validating the identification of the I/O bus as the bottleneck, with a 17% error on the predicted bandwidth. The source of this error will be discussed shortly.

To further examine the accuracy of the analysis tools, I applied my analysis tool to the dependence graphs generated by my annotations during the detailed simulations of configurations 2–4. These results are presented in Figures 6.3 and 6.4. The tool verifies that the actual critical paths were qualitatively the same as predicted by the original run.[4] The

[2]The new TCP window bottleneck causes burstier behavior than seen in the original run as the sender fills the window, waits for ACKs, then fills the window again. As a result of this burstiness, there is slightly more overlap between the ACK delays and NIC processing than predicted.

[3]The values of the following Linux kernel parameters were increased: `tcp_rmem`, `tcp_wmem`, `tcp_mem`, `rmem_max`, `wmem_max`, `rmem_default`, and `wmem_default`.

[4]Of course, the quantitative difference in the actual and predicted critical paths matches the error in the predicted bandwidths.

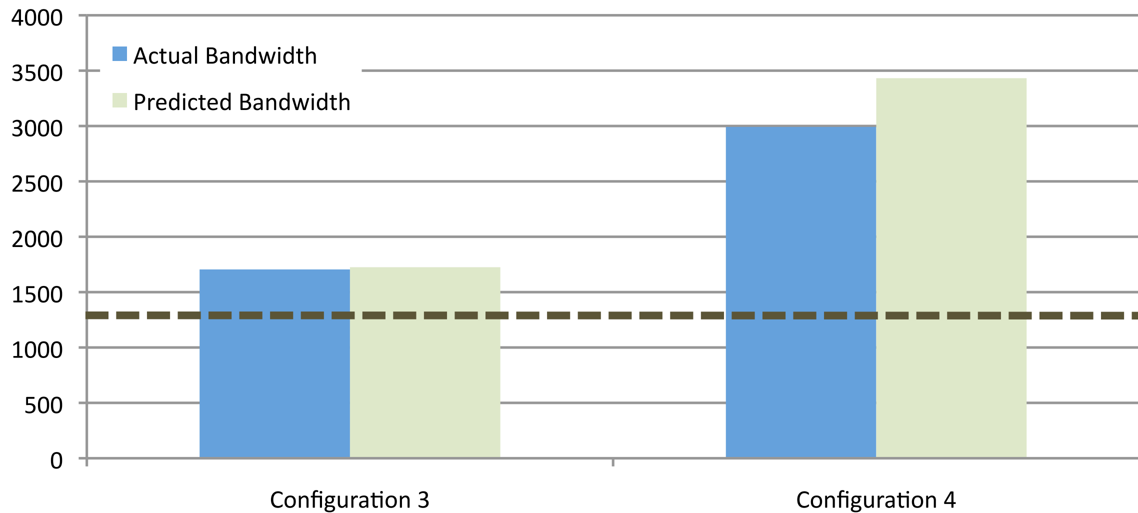


Figure 6.3: Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 2. Dotted line is bandwidth produced by initial configuration.

results of the configuration 2 simulation were used to predict the critical paths and performance for configurations 3 and 4, giving 1725 Mbps and 3432 Mbps (1% and 15% error), respectively. Using the result from configuration 3 to predict the critical path for configuration 4 projects a bandwidth of 3081 Mbps (3% error). Again, the critical paths were qualitatively in agreement with the original predictions. The bandwidth predictions based on these critical-path lengths are shown as the second and third bars in the corresponding groups in Figure 6.2.

My speedup predictions for configuration 4 overestimate the actual bandwidth by 3–17%. To identify the source of this error, I compared the critical paths produced by the analysis program for the three predictions and the measured result. I traced the error to a change in cache behavior at higher bandwidths. When the Linux `e1000` NIC device driver receives a small packet (such as the TCP ACKs the sink sends to the generator), it copies the packet for processing by the TCP stack. This copy moves the data from main memory (where the NIC DMA engine placed it) into the CPU's cache. In the 1 Gbps case, ACK packets remain in the cache until they are processed. However, as the data rate increases, the cache footprint of the benchmark increases as well: More packets are copied to kernel space for transmission (and buffered for potential retransmission), and ACK packets arrive at a faster rate. This increased memory pressure tends to flush the ACK packets out of the cache before they are processed. The resulting cache misses make ACK processing approximately 10x more expensive than in the original experiment. Thus, although the analysis program correctly identifies the state machines on the critical path, the

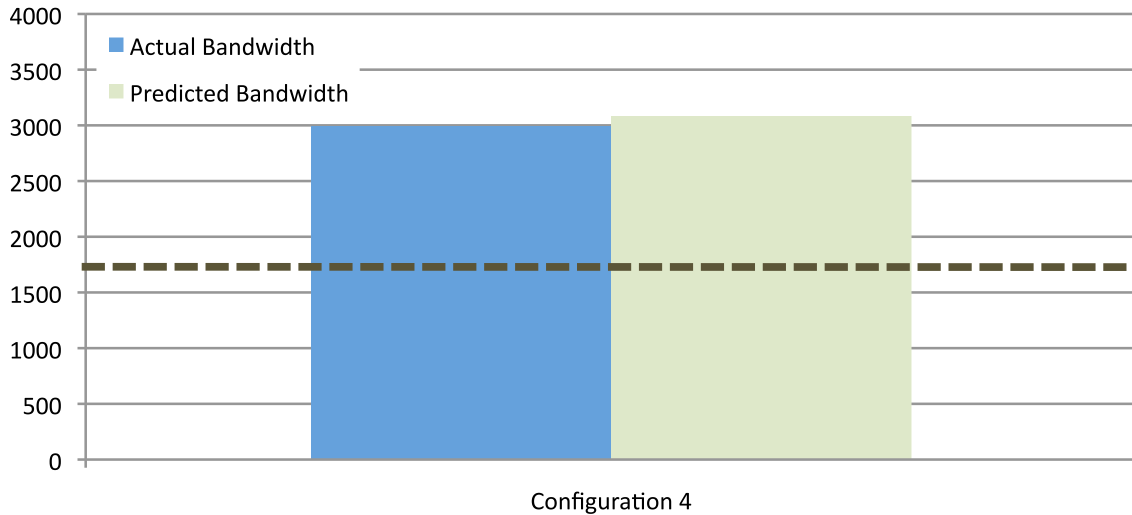


Figure 6.4: Actual bandwidth and predictions made with the TCP streaming benchmark beginning with configuration 3. Dotted line is bandwidth produced by initial configuration.

latencies it uses for the ACK processing states are optimistic, so its predictions overshoot the actual bandwidth. As successive experiments increase the bandwidth, this cache effect increases gradually; hence I achieve increasingly more accurate predictions from the later experiments.

Overall, I was able to correctly identify the top four performance bottlenecks using the initial 1 Gbps experiment, and predict the bandwidth increase achievable by removing the first three bottlenecks with accuracies ranging from 1% to 17%. Additionally, in all cases, all predictions and validations agreed on the bottlenecks and critical paths. The analysis program produced each of these predictions by reprocessing the annotation trace from the original simulation in under 5 minutes, which is orders of magnitude less CPU time than was required for all of the detailed simulation results. Furthermore, the workload was not CPU limited until the final bottleneck, so conventional software profiling tools would not have identified these bottlenecks.

6.3 Web Server Analysis

The techniques described are not limited to a single connection or a single core. To illustrate its applicability to multi-processor, multi-connection workloads, this section analyzes a web server. For this workload, the lighttpd web server is used, which powers several of the world's biggest web sites, including YouTube, Wikipedia, and Meebo [19]. The web server is stressed with a modified Surge [34] web client to request large files ranging from

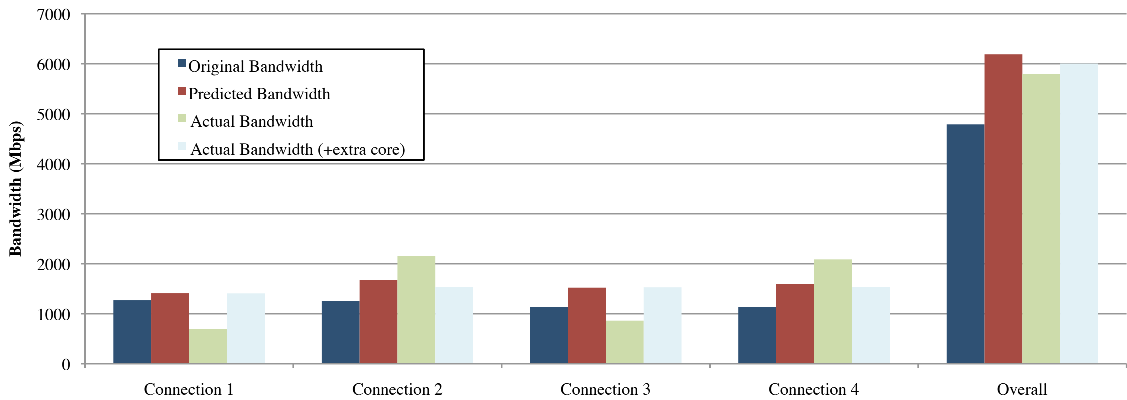


Figure 6.5: Actual bandwidth and predictions for four connections on web server.

500kB to 4MB. The URLs are requested in the same order in each experiment, and the initial checkpoint is created after the same number of HTTP responses have been received. The number of connections to the web server is limited to four to allow the results to be more easily analyzed and presented. However, the methodology and web server are both capable of handling more that four connections at a time. Since my primary interest is in the networking behavior, I reduced M5's simulated disk delay to a point at which it was no longer a bottleneck in this workload.

The web server was first simulated the lighttpd workload running on a single core. The resulting total bandwidth is 4.8 Gbps, shown in Figure 6.5. Using the data produced by the annotations during this run, the analysis program constructed four critical paths, one for each connection. The paths are nearly identical; the bandwidths of the individual connections range from 1130 Mbps to 1267 Mbps. The analysis program identified the primary latency in each of the connections to be communication delays inside the kernel on the web server machine. Furthermore, the resource-loop analysis points to the TCP stack parameters as the bottleneck. A software profiler would be unable to pinpoint the TCP stack parameters as the primary bottleneck, nor would it be able to identify the communication latency in the TCP stack as limiting performance.

In the real world, the server designer typically does not have control over the client's TCP stack parameters, so I chose not to address this bottleneck in that fashion. I instead attempted to reduce the communication latency in the server. This latency stems from the fact that the lighttpd web server is a single-threaded process that uses non-blocking I/O. The server handles a single connection until it is blocked (because it either has satisfied the request or is unable to continue due to a resource limitation, such as a full socket buffer) before moving on to the next connection. The latency observed is the time taken by the server thread to return to servicing a blocked connection after that connection becomes

ready. I hypothesize that adding an additional core running an additional lighttpd process and distributing the connections among the two processes should reduce the communication latency substantially.

This optimization is modeled by noting that in the baseline system, with four connections on one core, each connection must wait for three others before being serviced. With an additional core, the system will have two connections per core, so each connection will wait on only one other, a two-thirds reduction. Thus, I instruct my tool to predict the performance of the system with two-thirds of the relevant communication delay removed from each connection. The resulting critical paths predict per connections ranging from 1407 to 1670 Mbps, for a total bandwidth of 6.2 Gbps.

To validate these results, I added a second core to the simulated system and a second copy of the web server for that core. The resulting total bandwidth is 5.8 Gbps, 6% lower than the analysis tool's prediction. The analysis tool overestimates the bandwidth improvement for two reasons. First, decreasing the wait time by two-thirds does not account for the overheads of a multicore system, such as coherence effects and lock contention (particularly in the NIC driver). Second, because the system still contains only one network interface, NIC interrupt handling and TCP receive processing are isolated to just one of the two CPUs. This asymmetric loading causes two of the connections (those on the interrupt-handling CPU) to have much lower bandwidth than the tool predicted, with the other two having much higher bandwidth, as can be seen in Figure 6.5. While these asymmetries nearly cancel out, the inefficiency of this uneven load balancing further degrades overall performance.

To determine whether the tool's per-connection predictions are accurate in the absence of this asymmetric interrupt interference, I add yet another core solely to handle NIC interrupts and TCP receive processing. This arrangement allows for one core to be dedicated to each of the two lighttpd processes without interference. The results obtained from this experiment are also presented in Figure 6.5. The connections now behave more uniformly, and my per-connection bandwidth predictions are much more accurate, particularly for the individual connections (errors of 0%, 9%, 0%, and 3%) but also for the total bandwidth (error of 3%).

6.4 Analyzing a Copy Engine

In section 6.2.2, several bottlenecks were addressed, yet the last discovered bottleneck—the user-to-kernel copy—was left unaddressed. Here the system became CPU limited, and the simple modification of parameters was unable to improve performance. To improve

performance, more CPU resources (additional or faster CPUs) are required or some of the processing that the CPU is currently doing needs to be offloaded to specialized hardware.

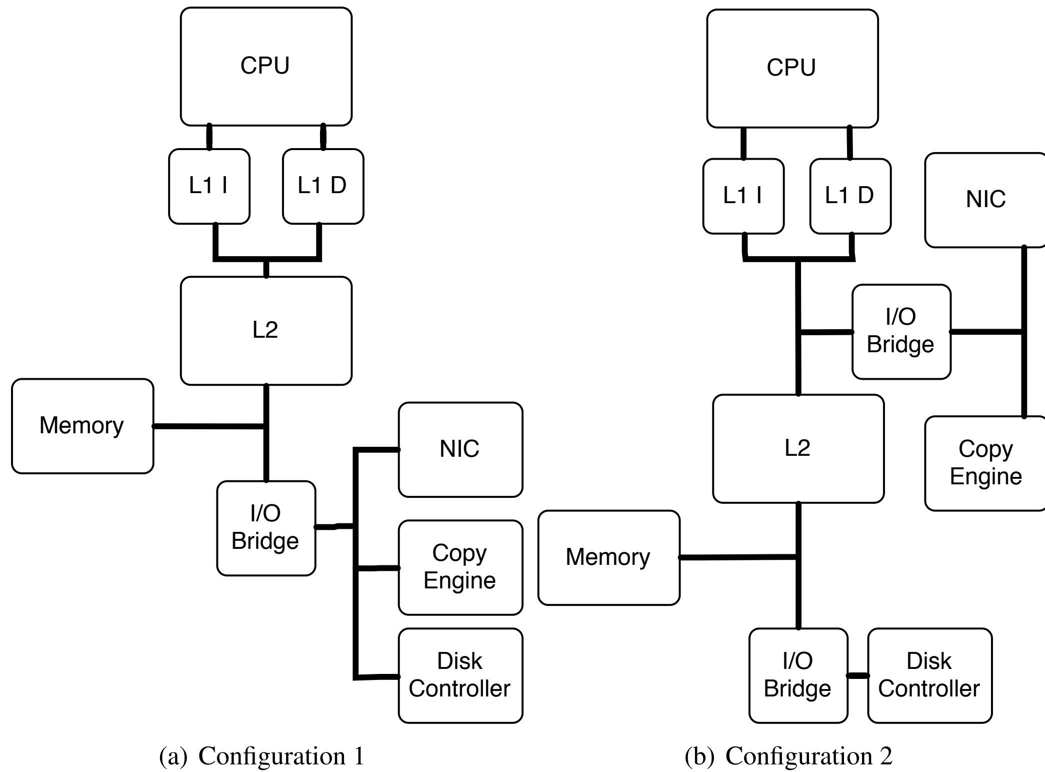
6.4.1 Copy Engine Overview

A DMA copy-engine has been proposed to offload the user-to-kernel and kernel-to-user copies in the network stack. However, while the copy-engine can easily parallelize the copy, interfacing with the engine is another question. To be most useful, the copy must be done asynchronously to the CPU execution. However, the asynchronous behavior is more difficult to interface with. Copies need to be queued and then the kernel needs to find other things to do while the copy is in progress.

Interfacing a copy engine with a kernel is not a simple task. Support for DMA copy engines made it into the Linux kernel in version 2.6.18, and even their support is limited. Due to the method that user applications employ to provide data to the kernel for transmission, it is difficult to make the user-to-kernel copy asynchronous. The data is provided to the kernel via a pointer argument in a system call, and as soon as the system call returns to the user program, the data pointer can be modified at will. Thus, the copy must occur before control returns to the user program. While the DMA engine could be utilized for this copy, there are not many tasks that the CPU could do while it waited for the copy to complete, so there would be very little performance gain from this approach.

The kernel-to-user copy provides a little more flexibility for offloading. A user-level application may frequently request enough data from a network socket so that a number of packets (up to approximately 45) can be needed to satisfy a single request. Thus, while the kernel is processing incoming packets, the previously received packets can be copied to the user application buffers asynchronously. However, as the buffer fills, the the pipeline of operations must drain.

Intel has recently provided such a copy engine on some of their server chipsets under the name Intel I/O Acceleration Technologies (IOAT) and the driver for this device was added to the kernel when copy-engine support was added to the Linux kernel. This copy engine is a simple DMA engine that processes descriptors provided by the kernel driver. It operates of physical addresses, so pages that are being copied to or from must be pinned in memory. The latency to the copy engine (and thus cost of uncached read/write operations to device registers) is reasonably high, so the driver that controls the device batches several copies together before communicating them to the device.



(a) Configuration 1 (b) Configuration 2

Figure 6.6: Evaluated copy engine configurations

6.4.2 Performance Evaluation

To evaluate the performance impacts and bottlenecks of a copy engine, I created a model of the IOAT DMA engine for the M5 simulator. It is capable of interfacing with an unmodified copy of the driver in Linux and is flexible enough to be connected to any location in the memory hierarchy. The various locations in the memory hierarchy principally change the latency for communicating with the device. These un-cached accesses can be quite expensive, so the closer it is to the CPU, the faster the accesses occur and the less requests the driver would need to batch together for a performance improvement.

Since the copy engine is only utilized for the receive side, the GenSink benchmark was used (as used above in Section 6.2.2). However, in this case, the receive system is evaluated. The methodology in this section is identical to the one presented in the previous section and is not repeated.

In this section, two system configurations are evaluated. The first is a traditional NIC attached to a standard I/O bus and a copy engine attached to that same bus. This configuration is a bit worse than the one that Intel provides, since the copy engine in the Intel case is embedded within the north bridge and is shown in Figure 6.6(a). The GenSink benchmark is run on the system with the copy engine removed to establish a baseline to

attempt to improve. The experiment results in a bandwidth received of 3.4 Gbps and the analysis program shows that the system is CPU limited, particularly 68% of the critical path is composed of the kernel-to-user copy[5].

This result shows that if a copy engine could increase the speed of the copy or allow the CPU to handle other tasks while the copy is progressing, a performance improvement may be realizable. Next, the system with the copy engine is evaluated. Surprisingly, this system produces a bandwidth of 2.6 Gbps, or almost 25% less than the system without the copy engine! The critical path is now dominated by the copy engine reading and writing data. This configuration increases the I/O bus bandwidth requirements by 3x over the system without the copy engine since the data is being written to main memory through the I/O bridge. Then, at some point later, it is being read from memory and re-written to a new location.

To remedy this situation, I switch to the configuration presented in Figure 6.6(b). The copy engine and NIC are now embedded within the CPU die and connect to the memory hierarchy between the L1 and L2 caches. This increases the bandwidth available to the copy engine and should improve performance.

The performance of the new configuration improves to 4.5 Gbps (a 33% improvement over the original). Analyzing the new critical path emphasizes what I alluded to in the overview sub-section: the overhead of communicating with the copy engine limits the performance that can be realized from the device. The analysis program shows the communication latency, queueing up several requests to send to the copy engine, and then waiting for copy engine to complete the copies and report back. Additionally, the analysis program reports a resource dependence loop around the descriptors used to communicate copies to the device. Unfortunately, in this case providing more resources is not an option. Each set of copies is of limited length (about 65KB), and while it is true that if the copy was much larger the overhead would shrink and the communication delay wouldn't be as large of an issue, this is not possible. If the copy engine is thought of as a pipeline, this means that the pipeline is almost always in a state of filling or draining, but the copies aren't long enough to keep the system at a efficient state for very long. A great deal of time is spent waiting for new copies to be received, and the CPU spins after issuing all the copies, waiting for them to complete. While it would be possible for the CPU to do something else, much like the transmission side, this is not a trivial optimization inside the kernel.

Recently, Large Receive Offload (LRO) has become popular. This technique limits the amount of work the network stack must do by coalescing several data packets together

[5] A little under half of this time is spent in other portions of code taking interrupts since the copy is such a long running operation

in the common case. In all implementations that I'm aware of, the data packets from multiple packets are combined in one through the use of pointers. Thus, while a single packet contains more data, limiting the work that the stack must do, the data is fragmented. This is unfortunate, because it still would require the same number of individual copies by the copy engine. A NIC that was able to reassemble the data fragments of incoming packets would have the advantage that kernel-to-user copies would be much more efficient. Implementations with a copy engine would require less communication for the copies to take place, and implementations that used a standard memory copy would be improved because a simple prefetcher could easily prefect data for the copy.

6.4.3 Kernel Bug

In doing the evaluation discussed above, several experiments produced bandwidths that were surprisingly low. I evaluated the etherenet traces that were produced by the low-performing experiments and saw that a duplicate acknowledgemetn was being sent by the receiving system. A duplicate acknowledgment normally means that a packet was lost in route from sender to receiver and instructs the sender to enter a fast retransmit mode which resends the packet immediately as well as limits the bandwidth on the connection to prevent future drops (assumed to be because of congestion). However, the trace data recorded from the simulations didn't show any state machines entering states in which they would drop or otherwise not deliver packets.

With the ethernet trace showing when the acknowledgment packet and duplicate acknowledgement were sent and the trace data available, I was able to trace the two acknowledgements back to where they were generated in `tcp_rcv_established()`. When TCP receive copy offload is enabled and `tcp_dma_early_copy()` is successful `copied_early` is set to true. This causes `tcp_cleanup_rbuf()` to be called early, which can send an acknowledgment. Further along in `tcp_rcv_established()`, `__tcp_ack_snd_check()` is called and will schedule a delayed acknowledgement. If no packets are processed before the delayed acknowledgment timer expires, the packet will be acknowledged twice. I submitted a patch to the Linux kernel that addressed this issue and it has been included in the kernel[6].

[6] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=53240c208776d557dba9d7afedbcdbf512774c16

Chapter 7

Related Work

There is a large body of work on performance analysis and critical path analysis that is related to this work. Generally, these related works fall into three categories: (1) work that uses a critical path graph to analyze a particular system; (2) work that uses profilers to locate software bottlenecks; and (3) work that uses other means to analyze a complex system.

7.1 Critical Path Analysis

A prerequisite for critical path analysis is a dependence graph representing timing constraints. This section presents a subset of the research to create dependence graphs for a specific domain. Creating a model in this manner requires a great deal of knowledge about the domain in question; it is both difficult and time consuming.

Fields et al. have developed a relatively simple model from scratch for an out-of-order execution pipeline [42]. The authors' principal concern is the granularity of instructions; their goal is to find which instructions are critical to better apply scheduling and speculation policies. They further show that with slack—a measurement of how much a particular event can be delayed without affecting performance—they can make decisions about which instructions to delay in a resource or technologically constrained system [43]. Finally, the authors seek to understand which instructions interact positively or negatively and ultimately lead to increased execution time [44]. Tune et al. [45] propose a more direct offline method for quantifying criticality and measuring slack as well as creating a new metric, tautness, which measures how critical an instruction is.

Nagarajan et al. extends Fields' work to the TRIPS architecture [46]. Here Fields' original model is expanded to encompass some of the unique features of the TRIPS system. The block execution model and network links are modeled, increasing the number of dependen-

cies and interactions at any given time. The focus in this study shifts from instructions to micro-architectural events as they examine how an application uses the resources of the TRIPS architecture.

Barford and Crovella have developed such a model for TCP [47]. They create a Packet Dependence Graph by observing traces collected at the endpoints of a given TCP connection. Their tool simulates the TCP stack on the two end points, allowing them to know what state the stack is in when packets are sent or ACKs are received. They then apply their system to HTTP transactions with a very coarse granularity (propagation, server, timeout, client, etc).

In the MPI domain, Yang and Miller [48] have developed a methodology to build a program activity graph based on message and synchronization calls. Their methodology is based on instrumenting well-defined interfaces to record events. Unfortunately, since many of the components in the systems I am interested in interact via custom interfaces (e.g., the DMA descriptors between the driver and NIC), this method isn't feasible in my work. This methodology is expanded upon by Hollingsworth and Miller, who have developed several metrics to guide users to performance problems including: Slack, True Zeroing, and Logical Zeroing [49, 50].

7.2 Profilers

Applications are frequently profiled with tools such as gprof [51] in an effort to improve the slowest parts of the code. This profiling is either timer based, in which the location of the program counter is recorded at some interval, or call/return based, in which additional code is added to the binary to record the time spent in each function and provide a call graph. Although invaluable to application developers, many profilers are limited to application code only. There are a handful of profilers that allow inspection of both applications and kernels such as OProfile [52] or er_kernel [53]. These kernel profilers are timer based, reporting a histogram of the location—functions in applications or the kernel—over some number of samples. Unlike application only profilers, they generally do not provide program flow information, and because of interrupt priorities they cannot observe critical section and other areas of the kernel that execute at the highest interrupt priority level.

7.3 Other Related Work

The Magpie [54] project at Microsoft Research sought to gather fine grained traces of a large number of software components across multiple systems, attribute the events in the trace to an initial request, and use machine learning techniques to find anomalies in the normal execution for real-time performance debugging. The goal was not to improve performance from the baseline, but instead to find problems in a real system as it ran and correct it to the baseline.

In a similar work, Tierney et al. [55] attempt to diagnose performance problems in complex high-performance systems. They use a combination of application modifications and kernel logging to gather end-to-end information about the system, time stamping important events as they happen. Visualization tools process the events to diagnose problems in a real system. This is done by presenting the user with "life lines" of important events moving through the system. It is up to the user to identify patterns or delays that are suspicious and investigate them further.

Aguilera et al. [56] attempt to find causal paths between messages by passively recording the traffic between systems and without any instrumentation in the machines themselves. The collected network traces are analyzed to find patterns and attempt to infer request patterns. They are able to identify the node in a distributed system (for example the database server for a website) that is the largest source of latency in responding to a request; however, their technique does not provide any information about the machine itself.

Azizi et al. created a simulator that symbolically simulates programs [57]. When executing a benchmark, rather than simply accumulating elapsed time due to various latencies in the architecture, it produces an expression containing variables (or symbols) that identify each latency. Their simulation engine can then algebraically compute the execution time and provide an equation that describes the runtime of a benchmark as a set of variables, each of which represents the performance of some aspect of the processor. This allows them to quickly estimate the performance of a huge range of parameters that otherwise would be computationally impossible. My approach to predicting performance is similar; however, I measure latencies of large groups of micro-architectural events. Instead of explicitly maintaining an algebraic expression that predicts performance, I maintain enough data about the observed latencies to alter them and observe the outcome.

In their Vertical Profiling work, Hauswirth et al. [58] find the root cause of performance problems that span several layers of software. They use software performance monitors coupled with hardware counters to understand system behavior. In their work, the user must identify particular software event(s) and hardware counters to monitor while being

careful not to significantly perturb the system. Additionally, because the information at various levels is gathered separately, the user must correlate the data by hand.

Chapter 8

Conclusion and Future work

Architects today are increasingly turning their attention to system-level issues for both performance and power. This trend is largely driven by the increasing dominance of I/O-intensive applications. Most tools today still focus on improving overall efficiency of processors; few tools exist to aid system designers in understanding the complex interactions of the various components in a complete system.

This dissertation addresses this need by developing a novel methodology for applying critical-path analysis to complete systems composed of concurrent components and spanning multiple layers of hardware and software. The demonstrated technique can capture dependence information from all levels of a system, and build a dependence graph that can be analyzed to locate bottlenecks.

The analysis tool that I have developed is capable of using this dependence graph to identify the bottlenecks in the system, locate critical resource dependence limitations, and predict the performance of a hypothetical system with the bottleneck removed or additional resources added. This predictive capability of the tools and techniques described within allows system architects to determine, in minutes, the results that can be expected for a particular optimization. The optimization being evaluated doesn't need to be realizable in any way—in fact, no way to achieve the optimization may be known—and crucially a prediction can be made without designing, implementing, and evaluating a system. This allows for a shortening of the design loop, as a prediction can be made in minutes. Finally, the analysis can be applied iteratively to identify the best course of action for fixing the top performance problems in a system.

The power of the techniques described has been shown by evaluating several networking workloads. I have shown that an end-to-end dependence graph can be constructed, bottlenecks identified, and performance predicted. Additionally, with little effort, the tool can be extended to analyze other sub-systems and applications.

I believe that the technique and tools described here can greatly assist architects by replacing ad hoc methods based on intuition with a rigorous methodology for both locating initial bottlenecks and rapidly predicting performance if a given bottleneck is removed.

8.1 Future Work

There are three directions in which this work could evolve. The first area is the introduction of additional automation for locating interactions and state machine boundaries. The second area is in the application of these techniques outside of a simulated system. Finally, this research could be applied to different systems and problems such as VM overheads and the composition of VM instances.

8.1.1 Automation

Although my current manual method of annotating state machines does a reasonable job—the TCP/IP stack in the Linux kernel can be annotated in a few days—I would like to automate the process. Research such as Mysore *et al.*'s Data Flow Tomography [59] deals with tracking the source of every memory location in the system. With this tracking system, it is possible to know the history of each byte. This information could be used to see which state machines interact, in what order they interact, and exactly where the interactions occur. This information could be used to quickly guide users to the locations that need annotation, however it may be possible for tools such as these to provide automated annotation in many cases.

8.1.2 Real Systems

The work described in this thesis has been carried out with the aid of a full-system simulator. While this is a reasonable choice, given that in its current form this technique will be most useful to architects as they explore platform designs, it may also be useful for real systems. This would allow larger workloads to be analyzed than is possible in a simulation and data to be collected for longer amounts of time. I believe that the techniques discussed in this study could be applied to real systems using instrumentation toolkits such as DTrace [1] or Linux Trace Toolkit [60]. In this case the observability of hardware devices would be significantly limited; however, as more systems continue to be built by layering on previous abstractions, interactions between software layers could be analyzed and improved. The major challenge to implementing this analysis on a real system is the overhead

introduced by the annotations and the amount of data recorded. Techniques would have to be developed that minimize the intrusiveness of the instrumentation and limit the amount of data that is recorded. Additionally, if the analysis spanned multiple machines a form of time synchronization would be required.

8.1.3 Different Systems

In this work I only look at the network aspects of workloads. It should be straightforward to analyze other aspects of interesting workloads such as storage or virtual memory. Other than annotating the additional parts of the relevant workloads, it is probable that new shared data structures would need to be developed. Many operations in the storage hierarchy are not done in a first-come first-serve order, but instead are grouped together by algorithms like elevator scheduling. The annotations would need to understand the other ordering schemes and it may be interesting to understand how the ordering affects the critical path. This would probably entail generalizing the enqueue/dequeue interface to support other orders, probably utilizing the flags that are currently ignored. With virtual machines becoming ever more popular, using critical path analysis to better understand the overheads involved in virtualization, especially of I/O devices, is also a promising direction.

There is also potential for wholly different applications of this work. It's possible that this research could be the foundation to estimate performance in a system when more resources are added or workloads composed. For example, the critical path could be analyzed for areas of computation that could—in principle—be run simultaneously, possibly predicting the performance of an optimized application running on multiple processors. Additionally, it may be possible to predict the performance of composing multiple VMs in a single machine or other similar analysis.

Bibliography

Bibliography

[1] B. Cantrill, "Hidden in plain sight," *Queue*, vol. 4, no. 1, pp. 26–36, 2006.

[2] W. Feng *et al.*, "Optimizing 10-Gigabit Ethernet for networks of workstations, clusters, and grids: A case study," in *Proc. Supercomputing 2003*, Nov. 2003.

[3] D. Kegel, "Mindcraft redux." `http://www.kegel.com/mindcraft_redux.html`, Jan. 2003.

[4] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, pp. 28–35, July 2000.

[5] Transaction Processing Performance Council, "Tpc benchmarks." `http://www.tpc.org`.

[6] Standard Performance Evaluation Corporation, "SPECweb99 benchmark." `http://www.spec.org/web99`.

[7] K. G. Lockyer, *An Introduction to Critical Path Analysis*. Pitman Publishing Co., 1964.

[8] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated network interfaces for high-bandwidth TCP/IP," in *Proc. Twelfth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pp. 315–324, Oct. 2006.

[9] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt, "Performance analysis of system overheads in TCP/IP workloads," in *Proc. 14th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 218–228, Sept. 2005.

[10] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *Proc. 9th Workshop on Hot Topics in Operating Systems*, pp. 25–30, May 2003.

[11] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong, "TCP onloading for data center servers," *IEEE Computer*, vol. 37, pp. 48–58, Nov. 2004.

[12] A. P. Foong, T. R. Huff, H. H. Hum, J. Patwardhan, and G. J. Regnier, "TCP performance re-visited," in *Proc. 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pp. 70–79, Mar. 2003.

[13] J. Chase, *High Performance TCP/IP Networking*, ch. 13, "Software Implementation of TCP". Prentice-Hall, 2003.

[14] J. Postel, "RFC 791: Internet protocol," Sept. 1981. `http://www.ietf.org/rfc/rfc791.txt`.

[15] J. Postel, "RFC 768: User datagram protocol." `http://www.ietf.org/rfc/rfc768.txt`, August 1980.

[16] J. Postel, "RFC 793: Transmission control protocol." `http://www.ietf.org/rfc/rfc793.txt`, Sept. 1981.

[17] J. Nagle, "RFC 896: Congestion control in IP/TCP internetworks." `http://www.ietf.org/rfc/rfc896.txt`, January 1984.

[18] V. Jacobson, "Re: query about tcp header on tcp-ip." ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt, September 1993.

[19] J. Kneschke, "lighttpd." `http://www.lighttpd.net`.

[20] Apache Software Foundation, "Apache HTTP server." `http://httpd.apache.org`.

[21] R. C. Bedichek, "Talisman: Fast and accurate multicomputer simulation," in *Proc. 1995 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 14–24, 1995.

[22] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (simulated) FLASH: Closing the simulation loop," in *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pp. 49–58, Oct. 2000.

[23] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, pp. 59–65, May 1998.

[24] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. 28th Ann. Int'l Symp. on Computer Architecture*, pp. 266–277, 2001.

[25] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, Jul/Aug 2006.

[26] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel & Distributed Technology*, vol. 3, pp. 34–43, Winter 1995.

[27] L. Schaelicke and M. Parker, "ML-RSIM reference manual." http://www.cse.nd.edu/~lambert/pdf/ml-rsim.pdf.

[28] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Halberg, J. Hogberg, F. Larsson, A. Moestedt, , and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, pp. 50–58, Feb. 2002.

[29] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: the SimpleScalar tool set," Tech. Rep. 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[30] R. E. Kessler, "The Alpha 21264 microprocesor," *IEEE Micro*, vol. 19, pp. 24–36, March/April 1999.

[31] High Performance Technical Computing Group, "Exploring Alpha Power for Technical Computing," April 2002. http://h18002.www1.hp.com/alphaserver/download/wp_alpha_tech_apr00.pdf.

[32] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, pp. 279–294, 1996.

[33] Hewlett-Packard Company, "Netperf: A network performance benchmark." http://www.netperf.org.

[34] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proc. 1998 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 151–160, 1998.

[35] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt, "Sampling and stability in TCP/IP workloads," in *Proc. First Annual Workshop on Modeling, Benchmarking, and Simulation*, pp. 68–77, June 2005.

[36] P. S. Panchamukhi, "Smashing performance with OProfile," Oct. 2003. http://www-106.ibm.com/developerworks/linux/library/l-oprof.html.

[37] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pp. 7–18, Feb. 2003.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Ann. Int'l Symp. on Computer Architecture*, pp. 24–36, June 1995.

[39] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[40] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools (SPDT'96)*, pp. 11–20, 1996.

[41] J. Sørensen, "gensink." http://jes.home.cern.ch/jes/gensink/.

[42] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proc. 28th Ann. Int'l Symp. on Computer Architecture*, pp. 74–85, May 2001.

[43] B. Fields, R. Bodík, and M. D. Hill, "Slack: maximizing performance under technological constraints," in *Proc. 29th Ann. Int'l Symp. on Computer Architecture*, pp. 47–58, 2002.

[44] B. A. Fields, R. Bodík, M. D. Hill, and C. J. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *Proc. 36th Ann. Int'l Symp. on Microarchitecture*, pp. 228–239, Dec. 2003.

[45] E. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *Proc. 11th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, p. 104, 2002.

[46] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, and S. W. Keckler, "Critical path analysis of the TRIPS architecture," in *Proc. 2006 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pp. 37–47, Mar. 2006.

[47] P. Barford and M. Crovella, "Critical path analysis of TCP transactions," in *Proc. SIGCOMM '00*, pp. 127–138, 2000.

[48] C.-Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int'l Conf. on Distributed Computing Systems*, pp. 366–373, June 1988.

[49] J. K. Hollingsworth and B. P. Miller, "Slack: A performance metric for parallel programs," Tech. Rep. 1260, Computer Sciences Department, University of Wisconsin-Madison, Dec. 1994.

[50] J. K. Hollingsworth and B. P. Miller, "Parallel program performance metrics: a comprison and validation," in *Proc. 1992 Int'l Conf. on Supercomputing*, pp. 4–13, Nov. 1992.

[51] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proc. ACM SIGPLAN 1982 Symp. on Compiler Construction*, (New York, NY, USA), pp. 120–126, ACM Press, 1982.

[52] J. Levon, "OProfile." http://oprofile.sourceforge.net.

[53] Sun Microsystems, "Performance analyzer: er_kernel." http://docs.sun.com/app/docs/doc/819-5264/afaht?l=sv&a=view.

[54] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: on-line modelling and performance-aware systems," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pp. 85–90, May 2003.

[55] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, "The net-logger methodology for high performance distributed systems performance analysis," in *Proc. 7th Int'l Symp. on High Performance Distributed Computing*, (Washington, DC, USA), p. 260, IEEE Computer Society, 1998.

[56] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. Nineteenth ACM Symp. on Operating System Principles (SOSP)*, pp. 74–89, 2003.

[57] O. Azizi, J. Collins, D. Patil, H. Wang, and M. Horowitz, "Processor performance modeling using symbolic simulation," in *Proc. 2008 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pp. 127–138, Apr. 2008.

[58] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: understanding the behavior of object-oriented applications," in *Proc. 19th Ann. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '04)*, pp. 251–269, 2004.

[59] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and visualizing full systems with data flow tomography," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 211–221, 2008.

[60] K. Yaghmour and M. Dagenais, "System administration: The Linux trace toolkit," *Linux J.*, vol. 2000, no. 73es, p. 22, 2000.