Full-System Critical Path Analysis

Ali G. Saidi[†] saidi@eecs.umich.edu
Nathan L. Binkert[‡] binkert@hp.com
Steven K. Reinhardt[†⋆] stever@reservoir.com
Trevor Mudge[†] tnm@eecs.umich.edu

[†]The University of Michigan Department of EECS
[‡]Hewlett-Packard Labs Palo Alto, California
[⋆]Reservoir Labs Portland, Oregon

Abstract

Many interesting workloads today are limited not by CPU processing power but by the interactions between the CPU, memory system, I/O devices, and the complex software that ties all the components together. Optimizing these workloads requires identifying performance bottlenecks across concurrent hardware components and across multiple layers of software. Common software profiling techniques cannot account for hardware bottlenecks or situations where software overheads are hidden due to overlap with hardware operations. Critical-path analysis is a powerful approach for identifying bottlenecks in highly concurrent systems, but typically requires detailed domain knowledge to construct the required event dependence graphs. As a result, to date it has been applied only to isolated system layers (e.g., processor microarchitectures or message-passing applications).

In this paper we present a novel technique for applying critical-path analysis to complex systems composed of numerous interacting state machines. We avoid tedious up-front modeling by using control-flow tracing to expose implicit software state machines automatically, and iterative refinement to add necessary manual annotations with minimal effort. By applying our technique within a full-system simulator, we achieve an integrated trace of hardware and software events with minimal perturbation. As a result, we can perform this analysis across the user/kernel and hardware/software boundaries and even across multiple systems. We apply this technique to analyzing network performance, and show that we are able to find performance bottlenecks in both hardware and software, including some surprising bottlenecks in the Linux 2.6.13 kernel.

1 Introduction

A key challenge in evaluating the performance of a modern computer system is that end-to-end behavior is the result of often subtle interactions between many complex, concurrently operating hardware and software components. When results do not match expectations, locating the bottleneck component is difficult. As architects and developers look to improve system performance, the areas in which they should focus their effort are not necessarily clear [15]. They can be left with no choice but to rely on ad-hoc methods or intuition, which can be faulty, or to explore a huge state space, which is costly both in terms of time and resources.

This situation is particularly acute in high-bandwidth network processing [7], where there is no single bottleneck that can be easily addressed. Instead, performance losses come from the combination of numerous overheads in interactions between the network protocol, software running on the CPU, the memory system, and the network interface controller [11, 18]. For example, consider the case where the end-to-end bandwidth between some sender and receiver is lower than expected. There are a number of reasons that this could be the case:

- Transmit data is queued for the network interface controller (NIC) in its 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 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, etc. We 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.

In this paper we describe and apply a rigorous methodology to identify bottlenecks quantitatively using critical-path analysis. A prerequisite for critical-path analysis is an event dependence graph representing timing constraints (PERT chart). 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. We make the following contributions in this paper:

1. We describe an automated technique for converting systems of multiple interacting state machines into a global dependence graph suitable for performance analysis, including bottleneck identification.

2. We 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:

   (a) state machines implicit in complex software systems, such as the Linux kernel's UDP and IP stacks, can be traced with limited manual effort by relying primarily on control-flow tracing and symbol table lookup

   (b) the necessary inter-state-machine dependence annotations can be uncovered using iterative refinement

3. We demonstrate an implementation of our technique that uses a full-system multi-machine simulator to identify bottlenecks across hardware/software boundaries.

Implementing our analysis using a simulator provides deterministic results and easy, low-overhead observability of both hardware and software components, along with the ability to find performance bottlenecks in hardware designs before they are built. We believe the techniques presented here could be applied to real systems using instrumentation toolkits such as DTrace [5] or the Linux Trace Toolkit [20], though the observability of hardware devices would likely be more limited.

We discuss our techniques in Section 2 and our particular implementation in Section 3. In Section 4 we describe an experimental analysis we performed using our tools, presenting results in Section 5. We discuss related work in Section 6, and we conclude and present future work in Section 7.

2 Technique

Critical-path analysis is a powerful technique for identifying the key bottlenecks in a complex system with multiple concurrent operations. It can also provide other useful metrics such as slack (the amount of additional time an operation could take without impacting the system) and speedup potential (the difference between the critical path and the next-most-critical path).

The prerequisite for critical-path analysis is a graph representing the dependences and timing between events in a system. Building this graph for a simple system with a few events is relatively simple. However, as the scope of the analysis increases, the difficulty of building the graph increases at a faster rate: in the extreme, the number of possible event interactions increases as the square of the number of events. In the networking domain, the task of developing a global dependence graph between the application, kernel stack code, device driver, and NIC rapidly becomes intractable. The task requires substantial effort, and detailed knowledge of all the systems involved.

We have identified a key technique that enables us to extend critical-path analysis to a much wider scope by algorithmically mapping the state machines that govern the behavior of individual components (software or hardware) to a global dependence graph representing the overall execution of this collection of machines. Our technique involves two steps. First, the execution of each individual state machine is converted into a dependence graph. Second, the dependencies between the individual state machines (or equivalently their dependence graphs) must be explicitly marked.

We begin 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 we have the additional challenge of generating a meaningful implicit state machine from program execution. We automate this process substantially by using program flow control information to identify states. The final step of identifying dependences between state machines must be performed manually. We describe how this step can be done in an incremental, iterative fashion, eliminating the need for substantial up-front effort.

2.1 Explicit State Machines

We convert a state machine into a dependence graph by a simple transformation. The fundamental insight is that events of interest, which are the nodes in the dependence graph, correspond to transitions (edges) in the executing state machine. Similarly the time spent waiting between events, which are edge weights in the dependence graph, correspond to the time spent in a state of

the state machine. There is thus a correspondence between the states (nodes) of the state machine and the edges of the dependence graph. These two equivalences are sufficient to generate a dependence graph from the execution of a single state machine.

We illustrate the conversion of one explicit state machine in Figure 1. An execution path through the state machine is transformed into a DAG by turning each edge traversal into a node and each node visit into an edge, with the edge weight corresponding to how long the state machine remained in the corresponding state.

When multiple state machines interact, they do so because a transition in one state machine creates an output that induces a transition in a different state machine. In this case, an edge must be inserted in the dependence graph between the node representing the first state machine's transition and the node representing the induced transition. The weight on these inter-state-machine edges corresponds to the communication latency between two interacting state machines and is normally assumed to be zero. Additionally, the weight on the other (intra-state-machine) edge coming into the induced transition node is set to zero. This edge corresponds to the waiting state in the consuming state machine, which ideally has no inherent latency of its own. Setting this weight to zero enables the correct calculation of the critical path through the node.

This technique reduces the generation of a global dependence graph to the description of several local state machines and their interactions. These local state machines are often well understood—in fact, hardware devices are often specified in terms of these state machines—or can be derived by inspection.

To further illustrate this process, consider the state machines presented in Figure 2. These state machines are simplified versions of three state machines in a NIC that interact to send packets out onto the network. The Descriptor Fetch state machine DMAs buffer descriptors from main memory where the device driver has placed them. The Transmit state machine reads these descriptors, DMAs any packets that are referenced in them, and places the packet in the outbound FIFO. Finally, the TX FIFO state machine takes packets from the TX FIFO and sends them out onto the wire.

Figure 3 shows one set of possible paths through the state machines in Figure 2. Note that the nodes are the transitions between states and not the states themselves in this graph. In the scenario shown, the TX FIFO state machine waits on the Transmit State machine, and the Transmit state machine waits on the Descriptor Fetch state machine. Once the Descriptor Fetch state machine has fetched a descriptor, the Transmit state machine can DMA the associated packet and hand it off to the TX FIFO state machine to be sent down the wire.

For our analysis, we must record two events: when the machine enters a new state (including the identity of the state being entered) and when the machine is blocked waiting on an external state machine (including the identity of the state machine and state being waited on). The mechanism for recording these events will vary, depending on both the nature of the state machine itself (e.g., software vs. hardware) and the instrumentation environment (e.g., run-time tracing on real hardware vs. simulation). In Section 3, we discuss how we annotate state machines to record these events in our simulator-based implementation.

The resulting dependence graph (which is very similar to a PERT chart [16]) is a fully connected DAG. (Any unconnected component is not on the critical path and can be ignored.) Every node in the graph corresponds to some state machine making a state transition, and the edges between nodes are weighted with the amount of time spent in that state. Every interaction edge represents a causal interaction and thus must point forward in time. We call this graph a bottleneck graph (bgraph) for the remainder of this paper.

The critical path between two nodes in the bgraph can be found using standard graph analysis techniques. Generally, the critical path of interest is one that starts where a request or data is produced and ends where the result is consumed. In our future discussion we use the term *starting state machine* to be any node in the producing state machine and *destination state machine* to mean any node in the consuming state machine.

2.2 Implicit state machines

It is not always the case that a component design is based on an explicit state-machine model, particularly for software components. Even when a software design is based on a state machine, that structure may not be evident in the source code. Nevertheless, for the purposes of our analysis, each software component must be decomposed into states where the component is busy and states where it is waiting on an external component for input. Given this decomposition, along with the transitions in the software component on which other state machines wait (i.e., outgoing dependence edges), we can determine whether the component is a bottleneck. If the component is a bottleneck, it may be desirable to decompose its behavior further into finer-grained states to better understand the nature of that bottleneck.

To enable non-experts to identify the state machines—even in such substantial pieces of unfamiliar software as the UDP and IP stacks in the Linux kernel—we can automatically generate 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 three tasks, all of which can be performed incrementally as described in the following section. All of these tasks are handled by manual identification of points of interest using source-code annotations. We discuss our annotation implementation further in Section 3.2.

The first task is identification of inter-state-machine dependences, which is largely unchanged from the explicit state machine case. Second, the automatic decomposition may require refining if there are places where the function-level decomposition is too coarse, e.g., if there is a waiting state in the middle of a function. The third task arises when a single executable binary

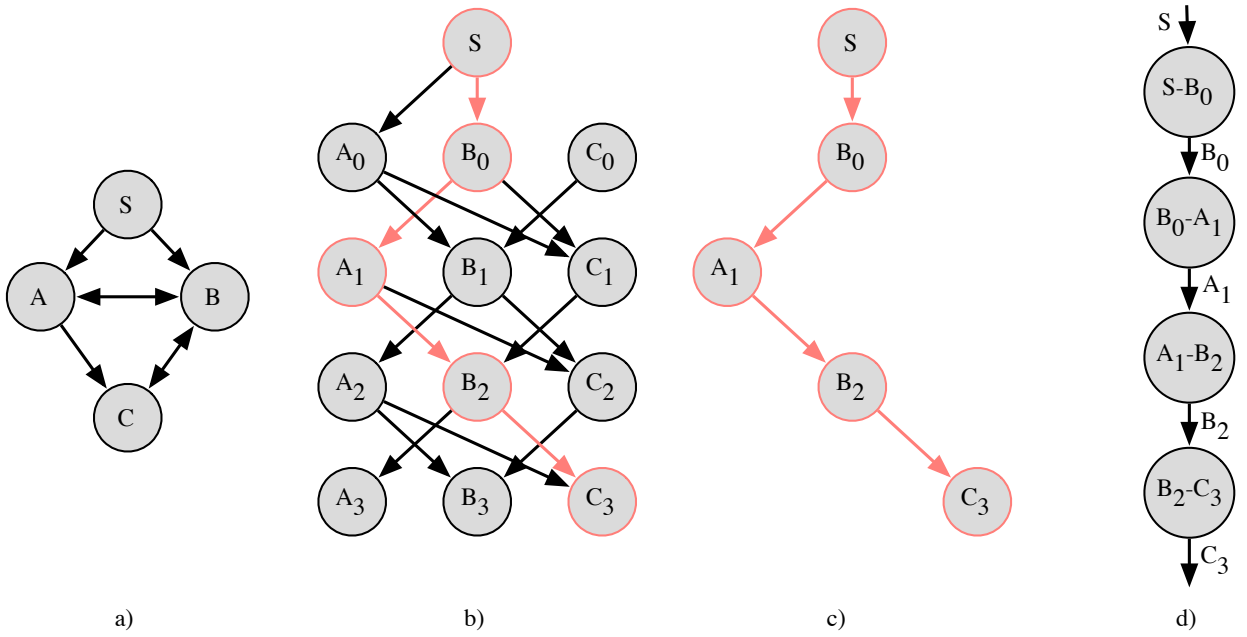

Figure 1: Conversion of a state machine. a) The original state machine; b) the trellis of that state machine (a DAG of all possible state transitions in a state machine); c) the execution path through the trellis; d) the conversion of the execution path to a set of nodes in the global dependence graph.

(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 into another, e.g., from the protocol stack into the device driver, and is thus 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. For software state machines sharing a single CPU, the currently executing state machine stops running when a new state machine begins because of an interrupt or function call.

In our experience so far, the difficulty in instrumenting software state machines has been minimal, requiring not more than a few hours of effort. For example, fewer than 100 lines of annotations were required to instrument the entire UDP code path—from application to driver—in a 2.6 series Linux kernel. We have gone through the process twice now, once manually instrumenting all states of interest and a second time relying on a tool to automatically generate software states as described above.

2.3 Iterative Model Construction and Verification

A key feature of our approach is that a complete and detailed state-machine decomposition is not required to perform analysis. As a result, users can make some minimal initial effort towards labeling states and inter-state-machine dependences, or use automatic tools as described above; use this information to perform an initial analysis; then use the results of that analysis to identify where critical modeling information is missing. This technique allows users to construct the model through iterative refinement, focusing their effort on the interesting and relevant portions, and using some trial-and-error where necessary to tease out key model attributes.

There are several tell-tale signs that a model is incomplete. States that are occupied for extremely long periods are typically waiting on another state machine, indicating a missing inter-machine dependence edge. Incorrect dependences can be detected by observing when a state machine that is labeled as waiting advances before the waited-on transition occurs. Our implementation prints a detailed warning message in this situation. Spurious edges connecting otherwise disjoint components often indicate that a transition between separate software state machines was not labeled. The function-based state names produced by our automatic decomposition do a reasonably good job of indicating the software state machine to which they belong, aiding in identifying these situations.

If there is no path from the starting state machine to the destination state machine, there are two possibilities. One is that the graph has disjoint components, in which case the user can direct their attention to the state machines on either side of the disconnect, where a dependence has likely been overlooked. The other possibility is that a resource along the way is so underprovisioned that a producing state machine is always waiting on its consumer, e.g., a queue is always full causing the producer to stall. In this situation, only the dependence edge from the consumer to the producer is created; because the system never observes the consumer waiting on the producer, the corresponding

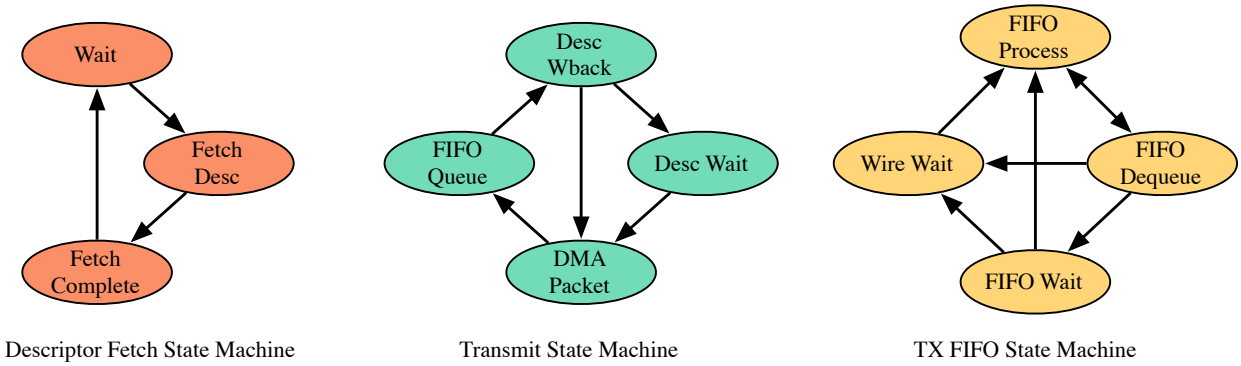

Figure 2: Simplified NIC State Machines

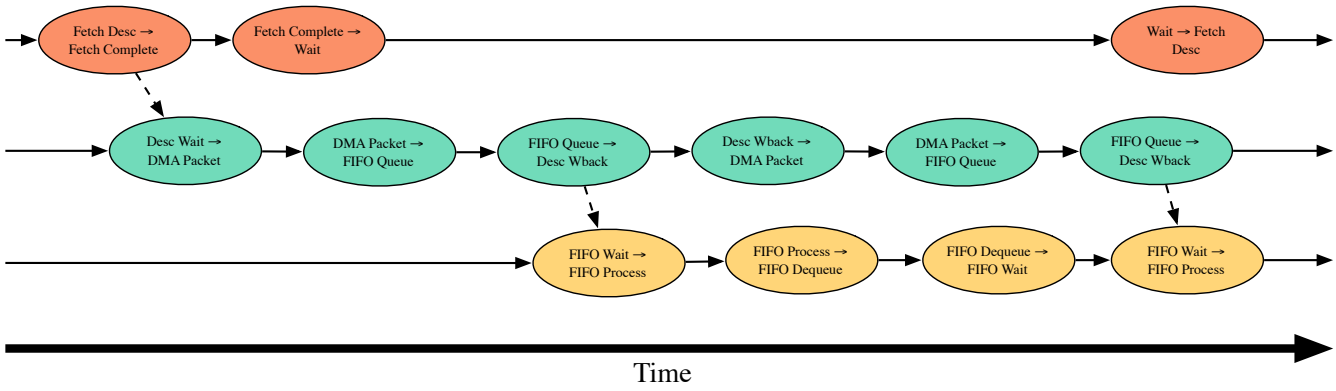

Figure 3: One possible bottleneck graph from the state machines illustrated in Figure 2. Note that time flows from left to right.

forward dependence edge does not exist. As a result, the graph is not strongly connected. We have only seen this happen when we intentionally under-provisioned a resource. The situation is easily identified by starting at the destination state machine and traversing inter-state-machine edges in reverse until a state machine is found that has no incoming inter-state-machine edges.

3 Implementation

In this section we describe our particular implementation of the analysis technique described above. First we discuss the simulator that we used, followed by the changes required to enable annotation of state machines. Next we discuss the tool that processes the recorded annotations, and finally we discuss the analysis and visualization techniques we have found useful.

3.1 M5 Simulator

Though our approach is not restricted to simulation, a simulator provides benefits not easily obtainable otherwise, specifically deterministic results and complete visibility into all aspects of both hardware and software. For this work we use the M5 simulator [4], which models Compaq Alpha systems with enough fidelity to boot unmodified Linux 2.6 kernels and includes a range of hardware components including multithreaded out-of-order processors, multi-level memory hierarchies, and I/O devices. M5's full-system simulation capability and detailed performance models of memory and I/O devices are particularly important, as they provide state-machine timing information from not only the application but also the kernel and hardware devices to enable meaningful end-to-end critical path analysis.

3.2 Supporting Annotations

We perform an analysis experiment in two steps: first M5 simulates the system(s) of interest and outputs the annotations to a file, then an analysis program reads the file and processes it. Decoupling the data generation and data analysis allows interactive analysis of the data. This ability comes at the cost of storage space for the data; however, in practice, this requirement has been relatively modest. One million annotation records require approximately 15MB of storage space (4MB compressed). The amount of simulated time these annotations represent is dependent on the workload and the number of states in each state machine. Our automatic function-based approach to decomposing software leads to software state machines have hundreds of

states. As a result, our current system generates a million annotations for every hundredth of a second of simulated time. Due to the cost of detailed performance simulation, a typical run may represent only a second of real time, producing about 1.5GB of uncompressed annotation data.

Since M5 uses software models to simulate all hardware components, adding state-machine annotations to the hardware is as simple as calling a function when the hardware model changes states. We added two functions to M5, *begin(state_machine, state)* and *wait(state_machine, state, wait_state_machine, wait_state)*, which record their parameters in a buffer. Periodically this buffer is flushed to disk.

To annotate software state machines, we leverage the simulator's ability to monitor execution with minimal system perturbation. Given some processor state (PC, privilege level, etc.) the simulator can find the nearest symbol (function name or label) and in that way we can automatically create states from symbol names in the kernel and applications as described in Section 2. When the simulator observes a branch-and-link instruction, it automatically finds the target symbol and records the beginning of a new state. However, using this technique, it is unclear what state machine that particular state belongs to. We solve this problem by annotating the entry and exit of state machines with pseudo-instructions. During simulation, we maintain a stack of active state machines, pushing the old machine each time a new state machine is entered and popping the old machine when the currently running machine is exited. Whenever a context switch occurs, we also change our state machine stack to the one that matches the newly running thread. Finally, we check that state machines are popped in the same order they are pushed, verifying that all entries and exists of the state machines are properly annotated.

In this way every function in the kernel and application can become its own state with no overhead and the user need only annotate the beginning and end of state machines as well as where waiting occurs. Explicit annotations in source code are converted (using gcc *asm* directives) into "pseudo-instructions" in the application and kernel binaries. These pseudo-instructions are encoded as unused opcodes and instruct the simulator to take a specific action. We have four pseudo instructions: *beginSM(state_machine)*, *endSM(state_machine)*, *begin(state)*, and *wait(state, wait_state_machine, wait_state)*. These instructions mark the beginning of a state machine, the end a state machine, entry into a state (in case the user wants to split up a large function), and waiting for a state machine, respectively. In each case the simulator simply records the event type, time of occurrence, and parameters involved.

3.3 Analysis

The bottleneck graph described above can be used for both data visualization and to find the critical path. While it's useful to think about the graph as a whole, building the entire graph is impractical. The graph quickly consumes memory and becomes difficult to work with, let alone visualize. While the algorithm to find the longest path of a DAG is straightforward [6], the time to do these operations on millions of nodes is impractical.

We solve this problem by avoiding the need to construct the full graph in memory. Instead we utilize the structure of the graph to find the critical path while reading in the data produced by our simulator. Because of the structure of the graph if the annotations are processed in the order that they occurred—which is trivial to do—the critical path from a given starting state machine to the current node is either from the previous node in that state machine or via an interaction with a different state machine. Thus to find the critical path we only need to store the current longest path from the start point to the newest state transition in each state machine. Any time one state machine interacts with another one, that state machine chooses the longer of its longest path or the longest path from state machine it's interacting with. Hollingsworth et al. used a similar method for their online computation of critical paths in the MPI domain [14]. With this method our analysis program can process 1 million annotations in approximately 80 seconds.

While the above technique can quickly find the critical path and can 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 to visualize when the number of nodes exceeds a few thousand.

To cope with this problem, we created a graphical model that is a hybrid between a canonical state machine graph and the previously described bgraph. We call this graph a combined graph (cgraph). Like the bgraph, cgraph nodes are state transitions; however, there is only one cgraph 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.

An example of this combined graph (based on the NIC model of Figures 2 and 3) is shown in Figure 4. Nodes are labeled with the system (computer) 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. We define the criticality of a particular state as the ratio of the time spent on the critical path in that state to the total time on the critical path. This metric gives a quick summary of the most important states on the critical path that is easy to compare with other experiments. This criticality is available by itself and is also coded on the graph by varying the edge color from black (non-critical) to red (most critical). 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.[1]

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

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

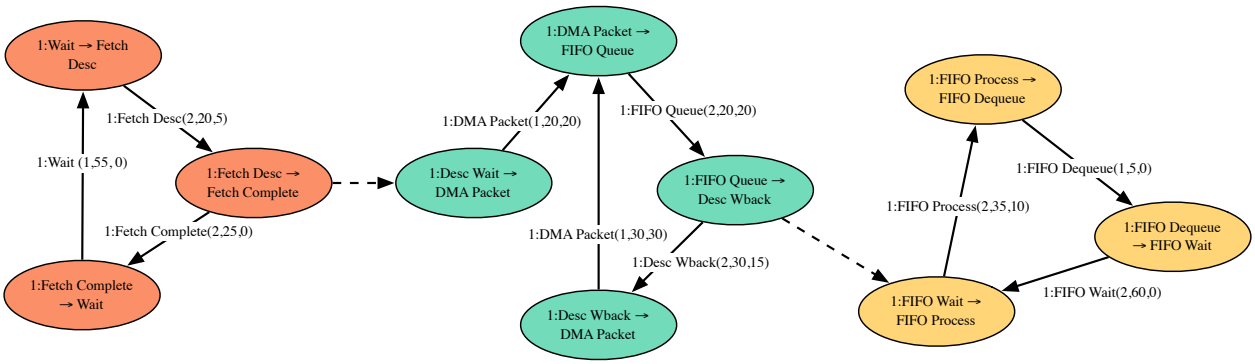

Figure 4: Combined bottleneck graph

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

In Figure 5, we show a complete cgraph based on the data presented in the results section. All the text normally on the graph has been removed as it would be too small to read. Though the graph's details are illegible, we include it to illustrate its size and complexity. Even though the cgraph is quite large, it is many orders of magnitude smaller than the corresponding bgraph, and is not too difficult to navigate online using panning and zooming. We present and discuss the most interesting portions of this graph in the results section.

4 Methodology

We have tested our methodology by running micro-benchmarks under Linux 2.6 on an appropriately modified copy of the M5 simulator. In the following sections we describe the benchmarks and simulator parameters in turn.

4.1 Benchmarks

Netperf [13] is a network microbenchmark suite developed at Hewlett-Packard. We used the UDP stream benchmark with the system under test sending UDP packets to its peer as fast as possible. In general, the time spent executing the user-space code is minimal; most of the processing time is spent in the networking stack processing packets or in driver code managing the NIC.

Netperf produces a bandwidth measurement indicating the maximum achievable communication rate between two systems. Though the critical-path analysis identifies the bottleneck in terms of latency, this latency bottleneck will also be the bandwidth bottleneck for the pipelined transmission of multiple packets. Put another way, if a state machine is still processing a block of data when the next block for it to process arrives it is both a source of additional latency and limits the rate in which the blocks of data can flow out of the system.

For the results shown in the next section we used a selection of kernels, parameters and packet sizes, each illustrating a different bottleneck. We started with Linux 2.6.13. After doing some initial experiments we found some interesting and unexpected performance problems in the kernel's "netfilter" code, which provides packet-filtering hooks used to implement firewalls and network address translation in Linux. To further investigate these we also used Linux 2.6.16 which fixed a number of issues with the netfilter code. In addition we ran each kernel with netfilter enabled and disabled.

In our simulations we used a fixed 1500 byte maximum transmission unit (MTU) as is standard on the internet today. We ran Netperf with two different UDP packet sizes, 1480 bytes (equal to the MTU after a header is added) and 16KiB (greater than the MTU, thus requiring fragmentation), to see how the bottlenecks are affected when fragmentation is introduced.

4.2 Simulator Parameters

As mentioned before, we modified a copy of the M5 simulator to support the annotating of states in hardware state machines and in software. We configured M5 to model an Alpha 21264 system based on the Compaq Tsunami chipset and used the default parameters for such a system with few exceptions. The I/O latencies were set to values similar to that of real hardware and the L2 cache size was set to 8MB. Additionally, the bandwidth of the I/O bus was set to that of a PCI-X bus (64bit, 133MHz) except when otherwise mentioned in the results section. The I/O bus is modeled as a generic bus, and does not model the particulars of

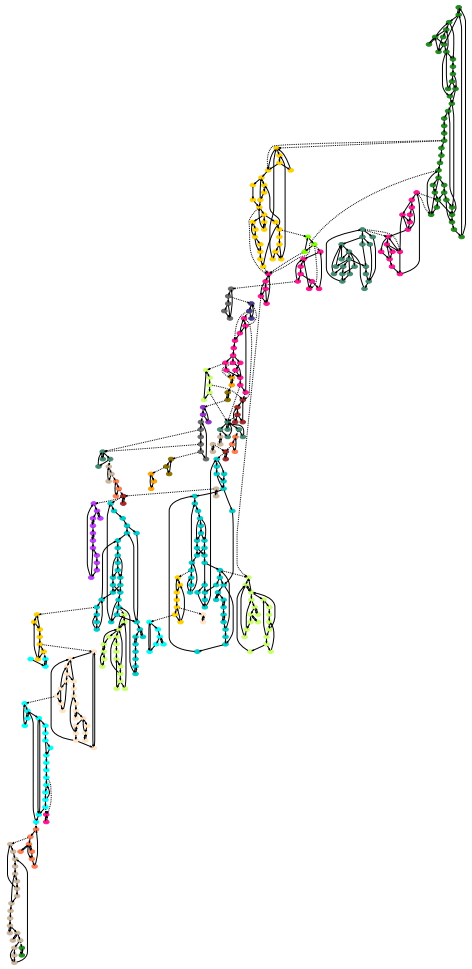

Figure 5: The combined graph generated by the analysis program.

the PCI bus specification. As such the generic bus tends to have a slightly higher effective bandwidth than a real PCI bus.

For the network interface we model an Intel 8254xGB chipset and scaled the link performance to not be the bottleneck. The model is accurate enough to support the standard Linux driver for this device. We use a 10 Gbps physical link to prevent the link itself from being the bottleneck.

All our experiments used two systems. In all cases one of the two systems is of interest (the system under test), while the other serves to stress the system under test. The system under test is configured using detailed CPU and memory-system models, as described above. The stressor is modeled with a simple 1 CPI CPU and a perfect memory system to prevent it from being the bottleneck and to reduce simulation time relative to modeling detailed CPUs on both systems.

5 Results

In this section, we begin by quantifying the perturbation that our annotations introduce. We then demonstrate the flexibility of our critical-path analysis by showing how our technique can detect bottlenecks at various levels of the system, including both hardware and software bottlenecks.

5.1 Annotation Overhead

We analyze the perturbation that our annotations introduce into the system by measuring the change in bandwidth between an annotated system and an unannotated system. To verify that our annotations are not introducing a large overhead, we compared the performance of each configuration we present in this section using the original kernel and application with the same experiment using our annotated kernel and application.

The perturbations we saw are a result of several things: 1. a larger binary due to the inserted pseudo-instructions; 2. different code emitted by the compiler (since functions are of a slightly different size the compiler could choose different optimizations for the function); and 3. interrupts and scheduling occurring at different times in the simulator due to the difference in the code size and instruction counts.

We found the annotations had very little effect on the bandwidth, resulting in a 3.8% change on average. This variation included both slight increases and slight decreases relative to the unannotated system, depending on the specific configuration.

5.2 Hardware bottlenecks

In our first experiment, we look at a Linux 2.6.13 kernel with netfilter disabled sending 1480-byte UDP packets. We configured the I/O bus to be a conventional 32-bit PCI bus running at 66MHz. 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 the Intel NIC we model, these descriptors are normally 16 bytes long. Finally, data may not always be ready to transfer when the bus is free, introducing idle cycles which further reduce the effective bus bandwidth.

Simulating this configuration resulted in a bandwidth of 1813 Mbps on the link. Our critical-path analysis identified the NIC transmit state machine as the primary bottleneck, specifically spending 96% of the time in the “DMA packet” state.

The transmit state machine portion of the combined graph (cgraph) for this experiment is shown in Figure 6. All other state machines have been removed from the graph, and rarely traversed paths of the transmit state machine have been omitted for legibility as well. As discussed in Section 3.3, each edge of the cgraph is labeled with the name of the corresponding state in the original state machine and the triple of values indicating the

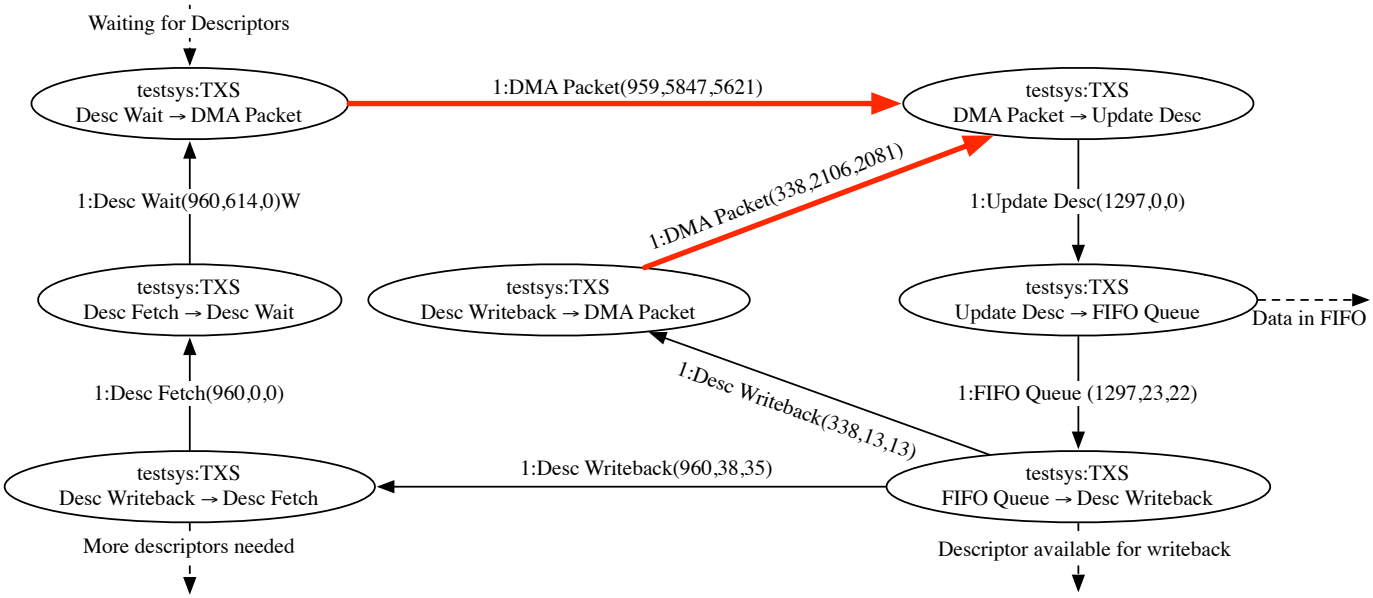

Figure 6: Transmit State Machine – Combined bottleneck graph

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 presented here is similar to the simplified one we mention in Section 2. 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 only waiting observed in this state machine during its execution was waiting for descriptors (the dotted line at the top left of the graph). The other intra-state-machine edges present in this graph are because of some state machine waiting on transmit state machine.

The graph shows that the edges consuming by far the most total time and most time on the critical path (the second and third numbers in the triple), marked in bold, both correspond to the "DMA packet" state. The next most critical path has a critical state in the Transmit Descriptor Fetch state machine (not shown). Both of these clearly indicate that the I/O bandwidth is insufficient to support the rate at which the application and kernel can generate data.

We verify this by replacing the PCI-like I/O bus (66MHz, 32-bit) with a PCI-X-like bus (133MHz, 64bit) and rerunning 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—we observed 2173Mbps of bandwidth on the link. The most critical state was no longer in the NIC's transmit state machine (or in the NIC at all), but instead was the more typical bottleneck of the (software) user-to-kernel data copy.

Our original 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 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.

5.3 Software and configuration bottlenecks

We now shift from looking at bottlenecks in hardware to bottlenecks in software. Here again we use a Linux 2.6.13 kernel, now with netfilter enabled, and a PCI-X-like bus. We look at the difference in bottlenecks between sending a 1480-byte UDP payload, the maximum size that can be sent without fragmentation, and a 16 KiB payload, which must be fragmented. The smaller payload size requires the application to perform more system calls to send the same amount of data. On the other hand, the larger payload must be fragmented by the transmitter and checksummed without the help of the NIC. These differences result in different paths through the kernel and more importantly different critical paths.

Upon running the two experiments, we noticed again that the bottlenecks were not in the expected location of the user-to-kernel copy, but instead being in the netfilter code. In Table 1 we show the states in which most of the time on the critical path is spent. The most critical state in the small UDP payload experiment was the netfilter function `ipt_do_table` which iterates through the table of netfilter rules—of which there were none in our experiment—checking the packet against each one. The experiment with larger payloads showed an even more surprising result: the most critical state was `ip_defrag`. By showing two different critical paths—and critical states—when only the payload size is changed, our technique demonstrates its sensitivity

1480 byte payload		16KiB payload	
Ip Send:ipt_do_table	11.36%	Ip Send:ip_defrag	12.78%
Ip Send:csum_partial	10.23%	Ip Send:memcpy	9.59%
Ip Send:nf_iterate	7.38%	Ip Send:ipt_do_table	6.94%
Ip Send:_read_unlock_bh	3.61%	Ip Send:ip_copy_metadata	6.90%
Ip Send:_read_lock_bh	3.56%	Ip Send:ip_fragment	6.51%
Ip Send:memcpy	3.47%	Ip Send:ip_fast_csum	4.87%
Ip Send:nf_hook_slow	3.42%	Ip Send:nf_iterate	3.84%
Ip Send:Call_Pal_Swpipl	3.26%	Ip Send:_read_unlock_bh	3.33%
Ip Send:ip_finish_output	2.95%	Ip Send:_read_lock_bh	3.12%
Ip Send:neigh_resolve_output	2.40%	Ip Send:sock_wfree	2.66%
Ip Send:__remlu	2.31%	Ip Send:ip_finish_output2	2.29%
Ip Send:skb_checksum	2.26%	Ip Send:neigh_resolve_output	2.29%

Table 1: Most critical states Linux 2.6.13 w/netfilter

to changes in workload parameters.

Additionally, we reran the analysis program to find the next most critical path and the resulting critical states on that path. The next most critical states are shown in Table 2. The small payload critical path is 36% shorter than the original one and the big payload critical path is 9% shorter. In both cases the path returns to the user-to-kernel copy. The significantly smaller next critical paths suggest that a large performance gain could be achieved if the current critical path through the netfilter code could be eliminated.

The critical paths we showed in Table 1 are a bit peculiar, especially for the larger payload. The most critical state, `ip_defrag`, defragments IP packets. However, the system under test is the transmit side, which is expected only to fragment packets. In fact, 5 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 combined 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 netfilter. This order results in a large overhead, and is an error in the code; netfilter should be invoked before the packet is fragmented. This problem was fixed in Linux 2.6.16, something that we were unaware of when we began running experiments, but sought out after we found our surprising result.[2]

Turning our attention to the results for the small payload we again see an unexpected function call, `ip_csum_partial`. While we do expect the outgoing packet to be checksummed, we expect the checksumming to be done by the hardware since the NIC is capable of checksumming packets smaller than the MTU. In this case the netfilter code erroneously believes it must calculate the checksum manually. Again, once we investigated this issue, we found that it had been fixed in 2.6.16.

An interesting observation from the most critical states tables above is that netfilter states appear quite often as critical even though no netfilter rules are present. Disabling netfilter and rerunning our experiments results in a 44% and 75% increase in performance for the small and large payload workloads, further validating the bottlenecks we have identified.

To verify the fixes mentioned above, we reran our experiments using the 2.6.16 kernel. The critical path in all the 2.6.16 experiments is the user-to-kernel copy, as might be expected. Enabling netfilter still affects performance, although not as significantly as it did in the 2.6.13 kernels. Netfilter is on the second most critical path which is only 5% shorter than the critical path including the user-to-kernel copy.

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` in the 16 KiB payload experiment. 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 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, our 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 our technique provides a much more direct, reliable, and automatic identification of the bottleneck.

While many of the problems highlighted above have been fixed in the 2.6.16 Linux kernel, our technique found these problems without any knowledge of the code or potential places that needed improvement. This technique can be useful to people in engaged in large or distributed development—where two programmers might not understand the interaction between their components—or to drive future hardware development giving engineers the ability locate and fix bottlenecks before building a product.

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

1480 byte payload		16KiB payload	
UDP Send:__copy_user	48.40%	UDP Send:do_csum_partial_copy_from_user	52.78%
UDP Send:ip_append_data	9.54%	UDP Send:Call_Pal_Swpipl	5.86%
UDP Send:Call_Pal_Swpipl	6.08%	UDP Send:alloc_skb	4.85%
UDP Send:udp_push_pending_frames	5.50%	UDP Send:ip_append_data	3.43%
UDP Send:alloc_skb	3.63%	Ip Send:ip_defrag	2.62%
UDP Send:sock_alloc_send_skb	3.36%	Ip Send:memcpy	1.93%
UDP Send:udp_sendmsg	3.28%	UDP Send:__kmalloc	1.79%
UDP Send:ip_generic_getfrag	2.77%	UDP Send:do_csum_partial_copy_fromiovecend	1.43%
UDP Send:_spin_lock_bh	1.51%	Ip Send:ipt_do_table	1.42%
UDP Send:__kmalloc	1.49%	Ip Send:ip_copy_metadata	1.37%
UDP Send:kmem_cache_alloc	1.41%	Ip Send:ip_fragment	1.31%
UDP Send:release_sock	1.29%	UDP Send:sock_wmalloc	1.22%

Table 2: Next most critical states Linux 2.6.13 w/netfilter

6 Related Work

There is a large body of relevant work on performance analysis and critical-path analysis.

A prerequisite for critical-path analysis is a dependence graph representing timing constraints. Fields et al. developed a relatively simple model from scratch for an out-of-order execution pipeline [9]. In this paper and follow-on work [8, 10] the principal concern was at the granularity of instructions; they seek to find critical instructions and minimize the delay that each cause through steering and scheduling. Nagarajan et al. extended this work to the TRIPS architecture modeling their block execution model and network links [17]. In that work, the focus shifts from instructions to micro-architectural events. Our work shifts the focus again upwards to system-level events.

In the networking domain, Barford and Crovella developed a critical-path model for TCP [2]. They create a Packet Dependence Graph by observing traces collected at the endpoints of a given TCP connection, then analyze the traces using a simulation of the TCP stack to attribute delay to one of several coarse-grain causes (server, client, propagation delay, protocol timeout, etc.). Unlike our work, their technique does not provide visibility into the specific bottlenecks within the client or server systems.

Work on critical-path analysis has also been done in the MPI domain. Yang and Miller developed a methodology to extract critical path information from message and synchronization calls and build a program activity graph based on these calls [21]. In this case the authors were interested in the network's usage, but not its implementation. Their methodology is based on instrumenting well defined interfaces to record events, which is not feasible for our work because many of the components in the systems we are interested in interact via custom interfaces (e.g., the descriptor ring between the driver and the NIC).

In all the work above, the models were developed essentially from scratch based on familiarity with the subject domain. One of our primary goals is to simplify model creation, allowing much larger systems to be analyzed.

The Magpie [3] work at Microsoft Research gathered fine grained traces of a large number of software components across multiple systems, attributed the events in the trace to an initial request and used machine learning techniques to find execution anomalies for real-time performance debugging. The goal was not to improve baseline performance, but instead find problems in a real system relative to the expected baseline as they ran. In similar work, Tierney et. al [19] modify applications and use kernel logging to timestamp interesting events, which are then visualized to help diagnose performance issues in real-time in a large distributed system.

Aguilera et. al [1] attempt to find causal paths between messages by passively recording the traffic between systems without any instrumentation in the machines themselves. They analyze the collected network traces to find patterns and attempt to infer request chains. 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 performance bottlenecks within individual machines.

Hauswirth et. al [12] explain performance phenomena in systems composed of multiple layers of software in their Vertical Profiling work. They use a combination of hardware performance counters and software performance monitors to understand system behavior. The information at various levels is gathered separately and must be correlated manually. Additionally, the user must know what particular software event(s) to monitor that will provide useful information but not significantly perturb the system.

7 Conclusion and Future Work

In this paper we have shown to how find bottlenecks in complex hardware and software systems through critical-path analysis. In doing so we have described a method for creating a dependence graph without complete familiarity of all the systems involved and proposed a method to visualize the interactions. We have then applied these techniques, with the aid of a simulator, to the Linux kernel, and shown that we can find both hardware and software bottlenecks. In the process, we identified two problems that existed in the Linux 2.6.13 kernel that we did not know about.

We believe that the techniques described here can greatly assist architects by replacing ad-hoc methods based on intuition with a rigorous method. These methods not only can help find performance bottlenecks, but also quantify the possible performance gain achievable. Furthermore, using these techniques can minimize the number of experiments an architect would normally run by focusing attention to the bottlenecks at hand.

We have a great deal of future work planned. In the near term we hope to analyze the performance bottlenecks inside the Linux TCP stack and extend this methodology to multi-processor systems. While it would be very difficult to analyze hardware on a real system, we would like to apply our technique to the analyze the software on a real system using a tracing toolkit. Finally, we hope apply these techniques to completely different systems, for example the cache hierarchy in a multiprocessor.

Acknowledgments

We would like to thank Kevin Lim and Lisa Hsu for their suggestions on early drafts of this paper and the anonymous reviewers for their helpful comments. This work was supported by a gift from Sun Microsystems.

References

[1] 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)*, pages 74–89, New York, NY, USA, 2003. ACM Press.

[2] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *Proc. SIGCOMM '05*, pages 127–138, 2000.

[3] 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)*, pages 85–90, May 2003.

[4] 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*, 26(4):52–60, Jul/Aug 2006.

[5] B. Cantrill. Hidden in plain sight. *Queue*, 4(1):26–36, 2006.

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

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

[8] B. Fields, R. Bodík, and M. D. Hill. Slack: maximizing performance under technological constraints. In *Proc. 29th Ann. Int'l Symp. on Computer Architecture*, pages 47–58, Washington, DC, USA, 2002. IEEE Computer Society.

[9] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proc. 28th Ann. Int'l Symp. on Computer Architecture*, pages 74–85, May 2001.

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

[11] 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*, pages 70–79, Mar. 2003.

[12] 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)*, pages 251–269, New York, NY, USA, 2004. ACM.

[13] Hewlett-Packard Company. Netperf: A network performance benchmark. `http://www.netperf.org`.

[14] J. K. Hollingsworth. An online computation of critical path profiling. In *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools (SPDT'96)*, pages 11–20, New York, NY, USA, 1996. ACM Press.

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

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

[17] 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*, pages 37–47, Mar. 19–21, 2006.

[18] 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*, 37(11):48–58, Nov. 2004.

[19] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proc. 7th Int'l Symp. on High Performance Distributed Computing*, page 260, Washington, DC, USA, 1998. IEEE Computer Society.

[20] K. Yaghmour and M. Dagenais. System administration: The Linux trace toolkit. *Linux J.*, 2000(73es):22, 2000.

[21] 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*, pages 366–373, June 1988.