End-To-End Performance Forecasting: Finding Bottlenecks Before They Happen

Ali G. Saidi
The University of Michigan
Department of EECS
Ann Arbor, Mich.
saidi@eecs.umich.edu

Nathan L. Binkert
Hewlett-Packard Labs
Palo Alto, Calif.
binkert@hp.com

Steven K. Reinhardt
AMD RAD Lab
Bellevue, Wash.
steve.reinhardt@amd.com

Trevor Mudge
The University of Michigan
Department of EECS
Ann Arbor, Mich.
tnm@eecs.umich.edu

ABSTRACT

Many important workloads today, such as web-hosted services, are limited not by processor core performance but by interactions among the cores, the memory system, I/O devices, and the complex software layers that tie these components together. Architects designing future systems for these workloads are challenged to identify performance bottlenecks because, as in any concurrent system, overheads in one component may be hidden due to overlap with other operations. These overlaps span the user/kernel and software/hardware boundaries, making traditional performance analysis techniques inadequate.

We present a methodology for identifying end-to-end critical paths across software and simulated hardware in complex networked systems. By modeling systems as collections of state machines interacting via queues, we can trace critical paths through multiplexed processing engines, identify when resources create bottlenecks (including abstract resources such as flow-control credits), and predict the benefit of eliminating bottlenecks by increasing hardware speeds or expanding available resources.

We implement our technique in a full-system simulator and analyze a TCP microbenchmark, a web server, the Linux TCP/IP stack, and an Ethernet controller. From a single run of the microbenchmark, our tool—within minutes—correctly identifies a series of bottlenecks, and predicts the performance of hypothetical systems in which these bottlenecks are successively eliminated, culminating in a total speedup of 3X. We then validate these predictions through hours of additional simulation, and find them to be accurate within 1–17%. We also analyze the web server, find it to be CPU-bound, and predict the performance of a system with an additional core within 6%.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: [Modeling technique, Measurement techniques]

General Terms

Measurement, Performance

Keywords

Performance Analysis, Critical Path Analysis

1. INTRODUCTION

System optimization requires knowledge of the current design'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 [6, 7, 15]. When dealing with interacting, 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 contributions of hardware, operating system, and application layers to a simulated system's performance is very challenging. When results do not match expectations, identifying the root cause is extraordinarily difficult.

This paper proposes an end-to-end critical-path analysis methodology that is capable of identifying performance bottlenecks across the user/kernel and software/hardware boundaries. We leverage the observability of a full-system simulation environment to annotate both software and hardware components of a pair of networked systems. We use measurements recorded from annotated runs to construct a complete dependence graph of component interactions. By identifying the waiting states in each component, we can generate the end-to-end critical path from a transmitting application on one system, through its operating system and network interface, across the network link, and back up to the receiving application on another system.

Because we record and analyze the full end-to-end dependence graph, we can extend our technique beyond identification of the critical path to performance prediction. By instructing our tool to reduce or eliminate the latencies of certain edges while processing the dependence graph, users can generate a predicted dependence graph. 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, our tool can—in a matter of minutes—identify the end-to-end performance bottleneck and, in successive runs of similar duration, predict the quantitative impact of removing that bottleneck and expose the subsequent bottleneck. In contrast, even with a fast, 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, hours or 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.

We also introduce a novel analysis technique that identifies critical paths constrained by resource limitations. We observe that resource constraints can induce "loops" in which the critical path oscillates between two state machines. For example, the critical path normally flows unidirectionally from a producer to a consumer, but an undersized finite buffer between the two causes it to also loop back as the producer waits for the consumer to free up buffer space. In many cases, the individual edges in the loop are not among the dominant edges in the critical path, so recognizing the loop is necessary to reveal the resource dependence. In these situations, the critical path can be reduced without speeding up any of its components by increasing the supply of the resource.

We demonstrate our techniques by first analyzing a pair of simulated Linux systems communicating over a 1 Gbps Ethernet link. The information that we extract from this single base run accurately identifies the top four bottlenecks, ranging from hardware constraints to TCP stack parameters to software (CPU) performance. This initial information also predicts the speedup achieved by eliminating each of the top three bottlenecks, indicating the potential for more than 3X performance improvement. Running additional simulations with these bottlenecks removed shows that these performance predictions are within 9%, 1%, and 17%, respectively. For predictions with non-negligible error, we identify the sources of those errors as situations in which a change in the execution environment causes the system to follow a code path slightly different from the one measured in the initial run.

Next, we apply our technique to a more complex application (a web server) servicing multiple concurrent connections. Our analysis indicates that performance is limited by latency in the application, due to different connections contending for the same processor resource. We hypothesize that it can be reduced by adding cores to the system, and predict that adding two cores will increase bandwidth from 4.8 Gbps to 6.2 Gbps. Simulations of a two-core system show a throughput of 5.8 Gbps, within 6% of our prediction. We show that half of the error (3%) is due to the asymmetry of interrupt handling and TCP receive processing in Linux.

The key contributions of this paper are:

- We propose modeling systems as state machines interacting via queues to capture all the information necessary for critical-path analysis. Specifically, queue-based interactions allow:
  1. tracking of independent contexts through multiplexed state machines;
  2. recording of buffered state-machine interactions not on the critical path (necessary for identifying alternate critical paths when predicting speedups); and
  3. modeling of finite resource constraints, including both real resources such as memory buffers and abstract resources such as TCP congestion window space.

  These features are sufficient to enable the analysis of multiprocessor, multi-connection TCP workloads.
- We identify critical-path "loops" as the characteristic signature of a finite-resource bottleneck, and automate the detection of these loops.
- We demonstrate that our critical-path methodology is capable of correctly identifying multiple bottlenecks from a single run and accurately predicting the quantitative benefit of removing those bottlenecks. This methodology eliminates the tedious and imprecise process of deducing bottlenecks from raw statistics, and enables designers to predict in minutes results that may take hours or days to prototype and simulate.

The following section summarizes the prior work on which we build. We then describe our technique in detail. Section 4 discusses our application of the technique to a TCP/IP microbenchmark and a web server, illustrating its use in a variety of scenarios for both bottleneck identification and performance prediction. Section 5 presents related work and Section 6 discusses our future work and conclusions.

2. BACKGROUND

Critical-path analysis (CPA) has been applied in many domains to find bottlenecks in complex, concurrent systems. The key input to CPA is a dependence graph that represents timing constraints between important events. Construction of this dependence graph is typically the most challenging aspect of applying CPA to a system. In most earlier work applying CPA to computer systems, dependence graphs were constructed either from scratch by domain experts or by restricting the analysis to well-defined interfaces such as message-passing between processes. (See Section 5 for further details.)

Generating a dependence graph spanning multiple layers of complex software and hardware communicating through wide, often non-standard interfaces precludes either of these approaches. Instead, we build on our prior work [20], which algorithmically maps the execution of the state machines that govern software and hardware component behavior to a global dependence graph. This section recaps that prior work to provide background for our enhanced technique to be discussed in Section 3. We refer readers to our previous paper for further details.

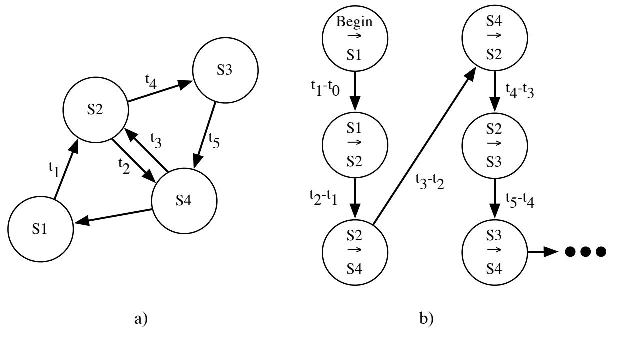


Figure 1: Conversion of a state machine to a dependence graph. a) The state machine being converted: nodes are labeled with the state name; transitions are labeled with the time they are traversed. b) The resulting dependence graph: nodes are labeled as the transition between two states; edges are labeled with the time spent in the corresponding state.

2.1 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 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 $S2$. In the right-hand part of the figure, we show the resulting dependence graph. Note that the dependence graph is created based only on observed transitions in a specific execution; a different execution of the same state machine would result in a different dependence graph.

Hardware devices are often designed as state machines, so this technique can apply directly to extracting a dependence graph from a hardware model. For components that are not modeled explicitly as state machines, such as most software, an initial automatic state partitioning can be generated based on function boundaries extracted from symbol tables. If the resulting dependence graph indicates that this function-based decomposition is too coarse, users can iteratively refine the decomposition using source-code annotations until the dependence graph captures adequate detail.

2.2 State Machine Interactions

Given a system decomposed into multiple state machines, a global dependence graph 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. These interactions are indicated through annotations in the source code (in the case of software) or device model (in the case of simulated hardware). As with the detailed decomposition of software state machines, annotations can be inserted fairly quickly by non-experts using iterative refinement: a dependence graph with incongruous states or spurious edges connecting unrelated components indicates missing annotations in the vicinity of the error.

2.3 Critical-Path Analysis

Once the state machines and their interactions have been identified, the annotated software is executed on a simulator that includes the annotated device models. The simulator records observed state-machine transitions and interactions in a trace. This trace is post-processed to create a global dependence graph using observed latencies as edge weights. The weights on edges representing waiting states are set to zero, so 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 then be found with standard graph analysis techniques.

A key limitation of this previous work is that state machine interactions are modeled as simple producer-consumer edges, and only appear in the dependence graph when one state machine actually waits on another during the observed execution. This simplification makes their technique unable to deal with several common characteristics of real-world systems, such as producers that block due to output-buffer-full conditions and state machines that are shared among logically independent processes or connections. Furthermore, this technique does not allow for performance prediction: because buffered dependences in which no waiting occurs are not tracked, there is no way to know whether an earlier arrival by the consumer would result in a stall. While our original work only looked at UDP/IP (a connectionless protocol with no flow control), our desire to look at more complex systems requires us to address these limitations.

3. QUEUE-BASED INTERACTIONS

The central contribution of this paper is the development of *queue-based* state-machine interactions as the key feature required for critical-path modeling of complex networked systems. Although a queue-based model is natural for situations involving actual data queues, we find that this model is more widely applicable than we anticipated, and is sufficient to address all the challenges we faced in modeling complex systems such a web server processing multiple concurrent TCP connections. In particular, the queue-based communication model is applicable to abstract resources such as TCP protocol constraints, allows tracking multiple independent contexts such as TCP connections through shared state machines (Section 3.3), and provides sufficient information to predict speedups (Section 3.4).

Our queue-based technique also enables a novel analysis we developed for identifying critical paths induced by finite resource constraints (Section 3.5). This technique allows us to detect automatically when the critical path is constrained by the availability of a particular resource, such as TCP buffer space. In these situations, performance can be improved without speeding up any system components simply by increasing the supply of the critical resource.

3.1 Annotating Queue-based Interactions

Consider the case where a producing state machine provides data to a consuming state machine through a queue. There are three possibilities: (1) the queue is empty, so the consumer must wait on the producer; (2) the queue is bounded and full, so the producer must wait on the consumer; or (3) the queue is at equilibrium, so neither state machine waits. While our prior work modeled the first type of interaction, it ignored the other two. Capturing all these forms of interaction is required to handle the full complexity of real-world systems and to predict performance.

We describe these interactions in software and in hardware models using annotations representing operations on abstract named

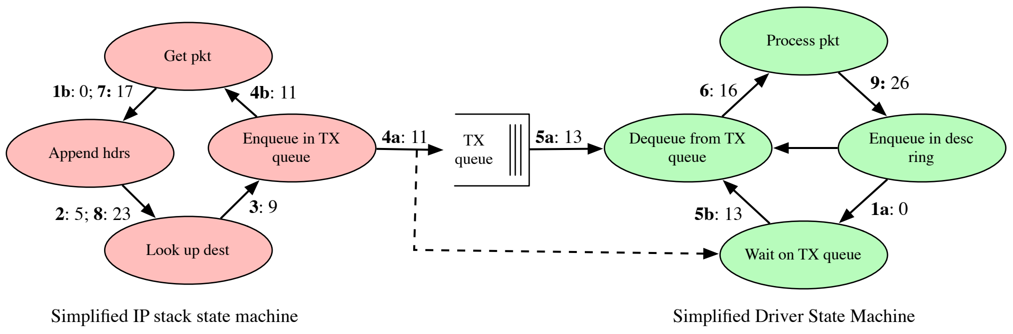
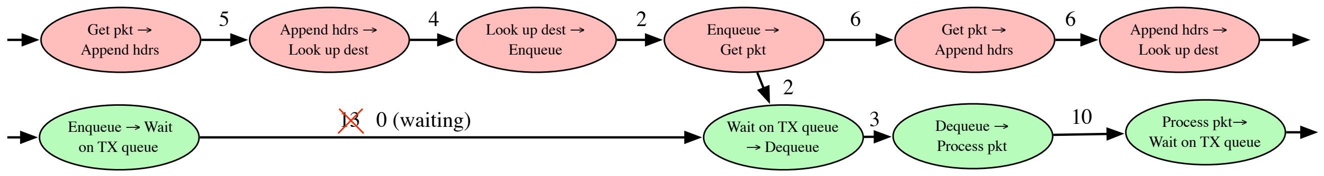


Figure 2: Simplified example state machines interacting via a queue. Edges are labeled with the sequence number and timestamp of observed transitions. A semicolon indicates that two transitions were observed on that edge.



Get pkt → Append hdrs
5
Append hdrs → Look up dest
4
Look up dest → Enqueue
2
Enqueue → Get pkt
6
Get pkt → Append hdrs
6
Append hdrs → Look up dest
2
Enqueue → Wait on TX queue
13 0 (waiting)
Wait on TX queue → Dequeue
3
Dequeue → Process pkt
10
Process pkt→ Wait on TX queue

Figure 3: Example dependence graph showing the interaction of the two state machines of Figure 2.

queues. The *enqueue* annotation indicates when a state machine produces an item for consumption by another state machine, while *dequeue* indicates when a state machine consumes an item produced externally. As with other annotations, occurrences of these events are merely recorded in a trace during execution. When the trace is processed off-line to construct the dependence graph, corresponding *enqueue* and *dequeue* events are associated to create inter-state-machine edges. The weight on this edge corresponds to the delay between the *enqueue* and *dequeue*, which represents the communication latency between the producer and the consumer when one is waiting for the other.

We use a *wait_empty* annotation to mark where a consuming state machine stalls because no item is available (typically just prior to a *dequeue*), and *wait_full* where a producer stalls because an output queue is full (typically prior to an *enqueue*). The weights on the intra-state-machine dependence-graph edges generated from these events are set to zero so that the critical path will properly follow the inter-state-machine edge.

To illustrate this process, consider the state machines shown in Figure 2. These are simplified versions of state machines that might exist in a kernel protocol stack and a network device driver. In this example, the stack state machine places packets in a queue and the driver state machine removes those packets and places them in a DMA descriptor that is given to the NIC.[1] We add *enqueue* and *dequeue* annotations to the corresponding states, a *wait_full* annotation to the stack's "Enqueue in TX queue" state, and a *wait_empty* annotation to the driver's "Wait on TX queue" state.

In the execution represented by the edge labels in the diagram, the driver reaches the "Wait on TX queue" state before the stack enqueues any packets, causing the driver to execute the code path with the *wait_empty* annotation and thus recording a *wait_empty* event. When execution of the stack code reaches the "Enqueue in TX queue" state, the *enqueue* annotation is triggered (but not the *wait_full* annotation, because the queue is not full). The dashed arrow from the edge labeled 4a to the "Wait on TX queue" state indicates that the *wait_empty* event in that state is dependent on the *enqueue* operation (as is the *dequeue* operation in the subsequent driver state).

Figure 3 illustrates a dependence graph created from the observed transitions in Figure 2. Though 13 time units elapse in the "Wait on TX queue" state, the edge representing that state is given a weight of zero because the state machine was stalled. A value of two time units is placed on the edge between the stack and the driver state machine, because the driver state machine waited two time units between the data becoming available and being consumed. Note that the critical path from a hypothetical common start state at time 0 to the transition into the driver's "Dequeue" state correctly flows through the stack state machine.

Though in this example the queue elements are atomic units (packets), some queues represent bulk quantities such as bytes. Our queue annotations take an optional integer parameter to allow enqueuing, dequeuing, and waiting for a particular number of abstract queue elements. For example, an application may issue a `read()` to a socket buffer that consumes the payloads of multiple TCP packets or only part of one TCP packet.

Also in contrast to our example, some abstract queues do not represent actual system queues. For example, we use queues to represent the space available in TCP flow control and congestion control windows. 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.

3.2 Refining Annotations

The ability to simulate and analyze an incompletely annotated system enables an iterative refinement approach that is key to giving non-experts the ability to annotate unfamiliar systems. Queue annotations are checked for consistency at runtime, and can pro-

[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.

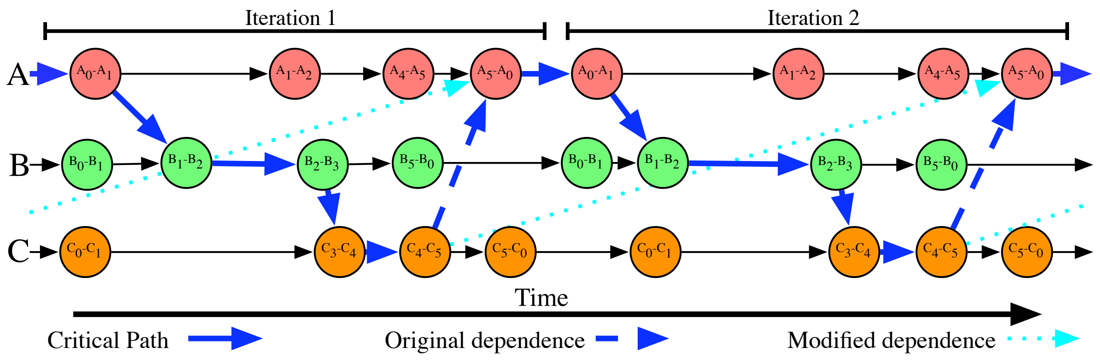


Figure 4: Illustration of a resource dependence loop.

duce a variety of warnings that inform the user where annotations may be missing or need refinement. These include notification of a dequeue operation on an empty queue, a state machine waiting to dequeue from a non-empty queue, etc. The off-line analysis program provides the most complete warnings as it constructs the dependence graph, but some basic sanity tests are incorporated directly into the simulator's support for annotations to enable earlier error detection when possible.

Another technique that helps in debugging annotations is to analyze a pair of applications communicating at low bandwidth. Analyzing a pair of applications communicating at low bandwidth should produce a critical path that is in the application state machine that is 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.

Although our 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—we would like to automate the process. Work such as Mysore *et al.*'s Data Flow Tomography [18] deals with tracking the source of every memory location in the system. With the tracking system, it is possible to know the history of each byte. This information could be used to see which state machines interact, where they interact, and in what order.

3.3 Tracking Multiple Contexts

A state-machine implementation often supports multiple concurrent contexts (logical instances). For example, kernel TCP stacks and web servers process multiple connections concurrently. To analyze modern systems, we must identify which instance of a state machine is transitioning when an annotation is encountered. We use a key, unique to the particular context, to distinguish instances for each state machine. The key is treated by our analysis program as an opaque identifier. For example, in the TCP stack, the pointer to the socket buffer 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.

Our analysis associates a state machine's identity (name and context key) with the queue elements it produces, along with the state-machine identities associated with any queue element(s) consumed by the state machine in the process of producing that output. As information flows through the system, queue elements build up a set of state-machine identities that have contributed to that flow. With this information, the analysis program can automatically deduce which state machines operated on a common connection, for example, which file descriptor on a client communicated with which file descriptor of a server process.

This connection-tracking information is required by the analysis program to properly track a single connection's critical path through the system. Without this information, dependence graph edges associated with different connections would be indistinguishable, and the analysis could result in nonsensical dependencies such as a critical path that traverses two unrelated applications that happen to use the same NIC.

3.4 Predicting Speedups

Because we record all state-machine interactions, not just the ones that caused state machines to stall in the observed execution, we can predict critical paths for hypothetical alternative executions. Specifically, we can rerun our analysis on the recorded event trace but artificially reduce the weights on selected edges in the observed critical path, similar to Logical Zeroing [13]. The analysis will identify the next most critical path (*i.e.*, the new bottleneck). Furthermore, the length of this next-most critical path relative to the original critical path quantifies the potential speedup that can be achieved by eliminating the original bottleneck. This process can be repeated to identify tertiary and subsequent bottlenecks and predict the speedups from removing them as well. 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 requires determining a parameter adjustment or other modification that would eliminate the bottleneck, prototyping that modification, and, finally, multiple hours of running the simulation.

3.5 Identifying 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 sped up without improving any operation latencies. This option 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. We have developed an automated technique for identifying these situations in our analysis. Our technique is based on the observation that, 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".[2]

[2]Although the critical path is acyclic, we often visualize results using a representation in which different iterations of the same state machine are folded onto the same graph nodes, causing these edges to form a loop.

In Figure 4, we illustrate 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-C_5 to A_5-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-A_1 of iteration 1 (in the upper left) to A_5-A_0 of iteration 2 (in the upper right) will be the critical path at a latency of 11 units.

If we add a second buffer to C, A can now send two units of data before requiring a credit from C. This change modifies the dependence graph; A_5-A_0 is now dependent on the instance of C_4-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-A_1 of iteration 1 through C and back to A_5-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.

Our analysis tool automatically identifies these resource loops and presents them to the user. The resource constraints we have identified with this technique include the NIC's DMA descriptor rings, kernel socket buffers, and TCP window size parameters. In many cases, the dominant component of the critical path is not part of the resource loop, so the resource dependence would not be apparent without this analysis.

4. APPLICATION

In this section, we apply our tools and techniques to finding bottlenecks in a system and predicting the performance after each bottleneck is removed. With our methodology, we are able to detect bottlenecks at all levels of the system or network of systems, spanning both hardware and software.

4.1 Implementation

We implemented the technique described in the previous section using the M5 full-system simulator [5], which provides deterministic results and complete visibility into all software execution and hardware models. M5 boots a Linux 2.6 kernel on a simulated Compaq Tsunami Alpha system. The simulator includes a range of hardware components including multi-threaded out-of-order processors, multi-level memory hierarchies, and network and disk interfaces. The simulator's detailed performance models of memory and I/O devices are very important for this work since they provide timing information about the network interface state machines.

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. The simulated CPU must fetch and execute these pseudo-instructions, so they do perturb the execution. However, they are reasonably rare and execute instantaneously, so their effect on simulated performance is minimal (about 3% on average).

We perform each analysis in two phases. First, we simulate the system(s) of interest and generate a file containing the annotations. This step typically generates a few hundred megabytes of data per second of simulated time. We then process this file with a separate analysis application. To minimize memory requirements, we exploit the structure of the graph to avoid building the entire graph in memory, using an algorithm similar to that of Hollingsworth [12]. For the experiments in this paper, we process the annotations in approximately five minutes.

4.2 Methodology

For this work, we used the default M5 parameters with a few exceptions. All simulated CPUs operated at 4 GHz. The I/O bus bandwidth was originally set to be 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 100 ns. All experiments involve two systems connected with a single simulated Ethernet link, with a link delay of 350 μs for the streaming experiments and 500 μs for the web server experiments.

In each experiment, M5 simulates the systems functionally as they boot a Linux 2.6.18 kernel and invoke the benchmark. Once the benchmark execution stabilizes, we create a checkpoint. We restore 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 100 ms, with annotations recorded after a 20 ms warm-up period.

We begin our critical path analysis by choosing suitable starting and ending state machines; for the streaming workload, we measure the path between the client and server applications, while for the web server workload we measure between the request and response components of the client application. We determine the critical path from the first time we enter the starting state machine to the last time we enter the ending state machine within the simulation window. We 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, we 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.

4.3 Streaming Benchmark Analysis

In this subsection, we analyze a simple streaming benchmark developed at CERN called GenSink [21]. The GenSink benchmark is composed of two applications: (1) a generator, which sends data as quickly as possible over a single TCP/IP connection; and, (2) a sink, which 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.

We begin with a detailed simulation of the benchmark on a baseline configuration (Configuration 1). For the remainder of this section we refer to Configuration 1 as C1, Configuration 2 as C2, etc. We use our analysis tool to process the trace generated by the annotations in this run to locate the bottleneck in the system. We then re-run our tool, 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 (C2), and the path itself predicts the bottleneck in that platform. We then run our tool again on the same trace data, including both the prior modifications and a new set of modifications that address C2's predicted bottleneck, to generate a new critical path and performance prediction for the further optimized system (C3). We repeat this process a

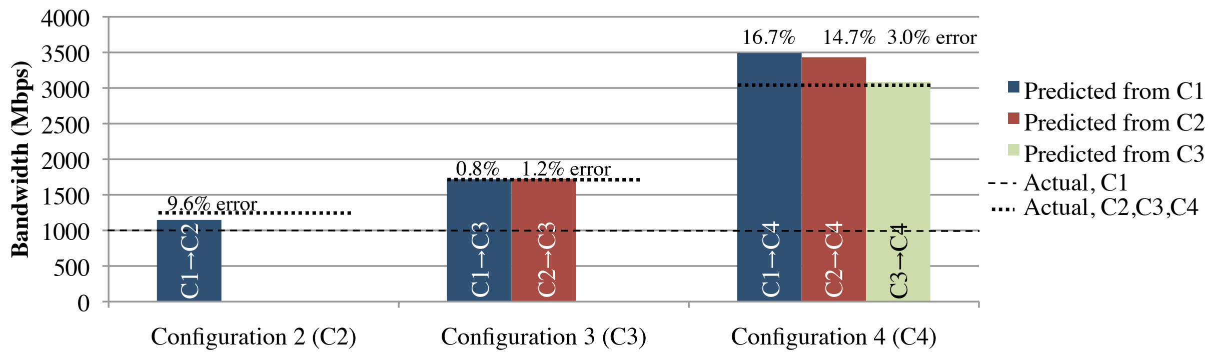


Figure 5: Actual bandwidth and predictions made with the TCP streaming benchmark. The number of bars increases from left to right as more experiments are available to use for making predictions.

third time, producing critical-path and performance predictions for a further optimized C4. To validate our predictions, we then do detailed simulations of Configurations 2–4. The experimental results are presented in Figure 5.

Our baseline configuration (C1) sets the link bandwidth between the systems to 1 Gbps. Our simulated systems achieve 1 Gbps, as shown by the dashed line in Figure 5. 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 C2, we 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 the Ethernet link by 90%, so we instruct our analysis tool to recompute the critical path based on the C1 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. We show this result as the bar labeled $C1 \rightarrow C2$ in Figure 5.

Examining the predicted critical path itself, we 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, our resource dependence loop analysis, described in Section 3.5, 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. For C2, our use of the Linux default sizes for these buffers is a bottleneck.

We then consider C3, which removes this bottleneck by increasing these buffer sizes. We 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. This change models the impact of eliminating this bottleneck without specifying any specific buffer sizes. With this change, combined with the 10 Gbps link modification for C2, our tool predicts the new bandwidth will be 1718 Mbps. This result is shown in the figure as the bar labeled $C1 \rightarrow C3$.

The critical path from this analysis run is dominated by the NIC state machine that DMAs packets from main memory, indicating that I/O bus bandwidth is the bottleneck. We propose C4, which removes this bottleneck by increasing the I/O bus bandwidth by a factor of four, and instruct our 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 (shown by the bar labeled $C1 \rightarrow C4$). 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.

To verify our predictions, we simulated each of our proposed configurations in detail. Our simulated C2, increasing the link bandwidth to 10 Gbps, produces a bandwidth of 1289 Gbps (the dotted line over the "Configuration 2" 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.[3] When a change in the system produces a qualitative change in the way state machines behave or interact, our prediction cannot account for the impact of those changes since they are not captured in the original dependence graph. Nevertheless, our tool identifies 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 our prediction for C3, we re-ran the detailed simulation with larger TCP stack parameters.[4] The measured bandwidth increases, verifying our prediction of the bottleneck. Furthermore, the new bandwidth is 1705 Mbps (again shown by a dotted line in the figure), within 1% of the predicted value.

We then simulated C4, which increases the I/O bus bandwidth by a factor of four. The measured bandwidth increases to 2992 Mbps (again, shown by a dotted line), validating our identification of the I/O bus as the bottleneck, with a 17% error on the predicted bandwidth. We will discuss the source of this error shortly.

To verify our critical path predictions, we applied our analysis tool to the dependence graphs generated during the detailed simulations of Configurations 2–4. The directly observed critical paths are qualitatively the same as predicted by our original run. Of course, the quantitative difference in the actual and predicted critical paths matches the error in our predicted bandwidths.

[3]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.

[4]We increased the values of the following Linux kernel parameters by roughly 10X: `tcp_rmem`, `tcp_wmem`, `tcp_mem`, `rmem_max`, `wmem_max`, `rmem_default`, and `wmem_default`.

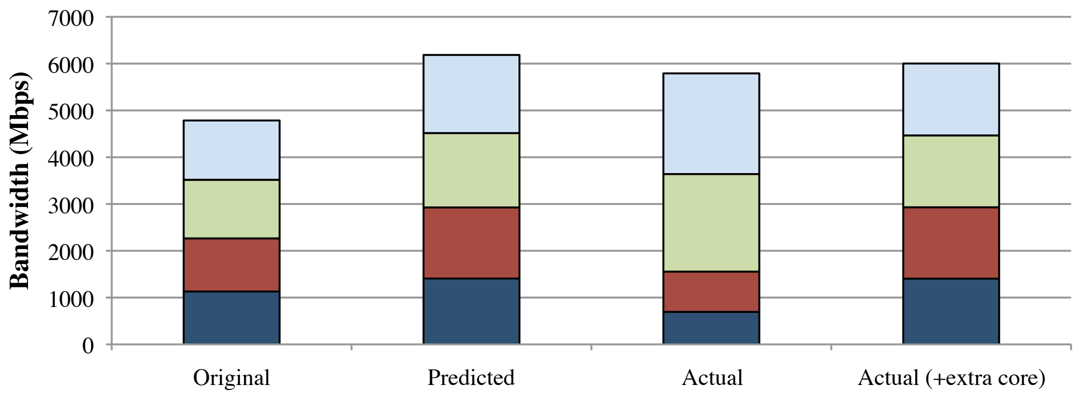


Figure 6: Actual bandwidth and predictions for four connections on web server.

All the predictions discussed thus far have been based on the dependence graph collected during a single simulation of C1. To better understand the accuracy characteristics of our technique, we also generated additional predictions for C3 and C4 using the dependence graphs generated during the simulations of the intermediate configurations C2 and C3. In all cases, the critical paths were qualitatively in agreement with our original predictions. The bandwidth predictions based on these critical paths are shown as the second and third bars in the corresponding groups in Figure 5. Comparing the C2-based predictions to the C1-based predictions, results for C3 differed negligibly, while the C4 results showed a small improvement, with error decreasing from 17% to 15%. The C3-based prediction of C4's bandwidth was significantly more accurate, reducing error to only 3%.

To understand the source of the prediction error for C4, and the reason why accuracy improves from 17% to 3% as we base our predictions on subsequent experiments, we compared the critical paths produced by the analysis program for the three predictions and the measured result. We 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 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, we achieve increasingly more accurate predictions from the later experiments.

Overall, we correctly identified 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 less than 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.

4.4 Web Server Analysis

Our technique is not limited to a single connection or a single core. To illustrate its applicability to multi-processor, multi-connection workloads, we turn our analysis to a web server. For this workload, we used the lighttpd web server, which powers several of the worlds biggest web sites including YouTube, Wikipedia, and Meebo [16]. We use a modified Surge [3] web client to request large files ranging from 500kB to 4MB. The URLs are requested in the same order in each experiment, and we create the initial checkpoint after the same number of HTTP responses have been received. We limited the number of connections to the web server to four, to allow the results to be more easily analyzed and presented, although our methodology and the web server are both able to handle more than four connections at a time. Since our primary interest is in the networking behavior, we reduced M5's simulated disk delay to a point at which it was no longer a bottleneck in this workload.

We first simulated the web server workload running on a single core. The resulting total bandwidth is 4.8 Gbps, shown in Figure 6. Using the data produced by the annotations during this run, we construct 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. Our analysis identifies the primary latency in each of the connections as communication delay 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, and it would be unable 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 we choose not to address this bottleneck in that fashion. We instead attempt 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 onto the next connection. The latency we observe is the time taken by the server thread to return to servicing a blocked connection after that connection becomes ready. We 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.

We model this optimization 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, we instruct our 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-connection bandwidths ranging from 1407 to 1670 Mbps, for a total bandwidth of 6.2 Gbps.

To validate these results, we 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 our prediction. Our tool overestimates the bandwidth improvement for two reasons. First, our model of decreasing 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 our system still contains only one network interface, NIC interrupt handling and TCP receive processing is 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 we predicted, with the other two having much higher bandwidth, as can be seen in Figure 6. While these asymmetries nearly cancel out, the inefficiency of this uneven load balancing further degrades overall performance.

To determine whether our per-connection predictions are accurate in the absence of this asymmetric interrupt interference, we 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. The connections now behave more uniformly, and our 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%).

5. RELATED WORK

Prior work has used dependence graphs to analyze current performance, locate bottlenecks, and predict possible performance in specific systems. In all these cases, the models were either developed from scratch with detailed knowledge of the subject domain or based on well-defined interfaces. This work seeks to simplify the model creation to allow analysis of much larger and more complex systems without the need for domain experts.

Yang and Miller [23] developed a methodology to build a parallel program activity graph based on message and synchronization calls. Their methodology is based on instrumenting well-defined MPI calls to record events. Unfortunately, since many of the components in the systems we are interested in interact via custom interfaces (*e.g.*, the DMA descriptors between the driver and NIC), this method isn't feasible in our work. The work is extended by Hollingsworth and Miller, who developed several metrics to guide users to performance problems, including Slack, True Zeroing, and Logical Zeroing [14, 13].

Barford and Crovella applied critical path analysis to TCP [4]. They developed a custom TCP stack simulator that analyzes traces collected at the endpoints of a TCP connection to create a Packet Dependence Graph. Their tool identifies the bottleneck as one of several coarse-grained causes (server, client, propagation delay, protocol, etc.). While their tool analyzes TCP protocol bottlenecks, it doesn't provide insight into the bottlenecks within the systems being analyzed.

Fields *et al.* developed a model of an out-of-order execution pipeline from scratch [9]. Although simple, the model captures the major interactions and was used to find critical instructions and steer or schedule them to shorten overall execution time or save power [8, 10]. Tune *et al.* [22] proposed 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. Li *et al.* [17] extended Fields' work to multi-processor systems. Nagarajan *et al.* used similar techniques to analyze the TRIPS architecture [19]. In addition to avoiding the need to develop models from scratch, our work increases the granularity of analysis from micro-architectural to system-level events.

Aguilera *et al.* [1] treat systems as black boxes and attempt to locate causal relationships between messages using only passively captured network traces. They analyze the collected network traces to find patterns and attempt to surmise request chains. Their analysis is able to identify a particular node in a distributed system as the source of the bulk of latency for a request; however, the root cause of the latency within that node is not ascertained.

Other approaches have been proposed that seek to find and explain performance anomalies without the use of dependence graphs and critical-path analysis. Azizi *et al.* created a simulator that symbolically simulates programs [2]. When executing a benchmark, rather than simply accumulating elapsed time due to various latencies in the architecture, it produces an expression containing variables (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. Our approach to predicting performance is similar; however, we measure latencies of large groups of micro-architectural events. Instead of explicitly maintaining an algebraic expression that predicts performance, we maintain enough data about the observed latencies to alter them and observe the outcome.

In their Vertical Profiling work, Hauswirth *et al.* [11] 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 cognizant not to significantly perturb the system. Additionally, because the information at various levels is gathered separately, the user must correlate the data by hand.

6. CONCLUSION AND FUTURE WORK

The increasing importance of I/O in today's network-centric world, combined with the platform-wide integration enabled by continued technology scaling, indicates that architects must consider performance issues at the system level. However, few tools exist to aid system designers in understanding the complex interactions of the various hardware and software components that make up a complete system.

We have addressed 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. Our technique can build a dependence graph of the interactions within the system and is able to identify end-to-end bottlenecks in networked applications consisting of single or multiple connections. We present a tool that can use this dependence graph in several ways: 1) to identify the bottlenecks in the system; 2) to identify critical resource dependence loops that degrade system performance; and, 3) to predict the performance of a hypothetical system with the bottleneck removed or resources added to fix the dependence loop. The predictive capability of our system enables architects to determine, in minutes, the results that can be expected for a particular optimization. Finally, the analysis can be applied iteratively to identify the best course of action for fixing the top performance problems in a system.

We believe that the techniques 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 the performance of removing specific bottlenecks. While the workloads in this paper focused on networking, we believe that the techniques we outlined could be applied to other application domains (*e.g.*, database servers). Much of our analysis involved the Linux kernel, demonstrating our tools' ability to handle large, complex software systems. To enable other researchers to build on our work, the software described in this paper is available on the M5 web site, `http://www.m5sim.org`.

Our future work is segmented into two areas. The first area is continued development of the annotation framework. We would like to further automate the annotation process, as discussed in Section 3.2. A more automated process would simplify the study of other large workloads, including those involving virtual machines. The larger portion of our future work involves extending the analysis of the dependence graphs we create and increasing the sophistication of the manipulations we can perform to predict future systems. Work in this area includes improving prediction techniques for multiple processors and dealing with the impact of software scheduling. For example, we would like to extend our methodology to predict the performance impact of moving workloads run on independent machines to a shared VM by building a composite of their dependence graphs.

Acknowledgments

This work was supported by the National Science Foundation under grant CNS-0834435. Srilatha (Bobbie) Manne and Mark Hill provided comments on early drafts of this paper that helped us improve the presentation considerably.

7. 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, 2003.

[2] 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*, pages 127–138, Apr. 2008.

[3] 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*, pages 151–160, 1998.

[4] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *Proc. SIGCOMM '00*, pages 127–138, 2000.

[5] 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.

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

[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, 2002.

[9] B. Fields, S. Rubin, and R. Bodík. 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. Bodík, 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] 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, 2004.

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

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

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

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

[16] J. Kneschke. lighttpd. `http://www.lighttpd.net`.

[17] T. Li, A. R. Lebeck, and D. J. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *Proc. fifteenth ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 128–137, New York, NY, USA, 2003. ACM.

[18] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. *SIGARCH Comput. Archit. News*, 36(1):211–221, 2008.

[19] 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. 2006.

[20] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge. Full-system critical path analysis. In *Proc. 2008 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, Apr. 2008.

[21] J. Sørensen. gensink. `http://jes.home.cern.ch/jes/gensink/`.

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

[23] 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.