Sampling and Stability in TCP/IP Workloads

Lisa R. Hsu, Ali G. Saidi, Nathan L. Binkert, and Steven K. Reinhardt
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{hsul, saidi, binkertn, stever}@eecs.umich.edu

Abstract

To overcome the performance cost of simulating detailed timing models, computer architects often augment these models with fast functional simulation for fast-forwarding, checkpointing, warm-up, or sampling. Our experience with simulating network-intensive workloads has shown us that researchers must exercise caution when using such techniques in this environment. Because the TCP/IP protocol is self-tuning, the difference between the effective performance of the functional and detailed models can lead to artificial results.

In this study, we examine the effects of applying conventional simulation speedup techniques to two networking benchmarks on the M5 simulator. We find that short samples taken immediately after switching to detailed models may capture only the period of time where TCP is tuning itself to the new model. In some situations, the transition is so disruptive that no meaningful results can be obtained. However, stable simulation results can be attained with detailed simulations of moderate length in most cases.

1. Introduction

Detailed timing-accurate computer system simulators typically run thousands of times more slowly than the hardware they model. As a result, simulation of complete runs of realistic workloads is impractical. In practice, architects simulate one or more small samples of each workload with the assumption that the data collected during these samples will be representative of the workload as a whole.

A key requirement for this sampling approach is the ability to create the initial system state for a sample interval relatively cheaply. These initial states are typically generated using functional (non-timing) simulation.[1] Even the most basic functional simulator is typically orders of magnitude faster than detailed timing simulation. Using advanced techniques such as direct execution and dynamic code generation, functional simulators can execute code only a few times slower than hardware [11, 4].

A fundamental assumption underlying this technique is that the system's state does not have any significant dependence on the system's timing. Otherwise, the lack of accurate timing in the functional simulation would result in a system state that is unrealistic for the configuration being studied. While this assumption often holds true, it is not axiomatic. In particular, this assumption does not hold for self-tuning workloads, which adapt their behavior to observed system performance.

In this paper, we examine an important class of self-tuning workloads—those based on the TCP/IP network protocol—to understand the impact of this timing dependence on common program sampling techniques. TCP adapts dynamically to the available end-to-end bandwidth by varying the number of outstanding packets (known as the “window size”) at the sender. When the system under test is a bottleneck, as is not uncommon, its performance will determine end-to-end bandwidth. During functional simulation, TCP will set its window size according to the largely meaningless effective system performance of the functional model, affecting the architectural state of the system (including TCP's self-tuned connection parameters and the occupancy of network buffers). Because of these effects, this state is not a realistic state for the system being modeled. After switching to the detailed timing model, TCP must adjust its window size accordingly before an accurate network bandwidth reflecting actual system performance is reached. Note that this effect is orthogonal to the issue of behavioral variation along the time axis of a program. This tuning period exists regardless of the point at which the switch from functional to detailed simulation occurs.

1. Initial states can also be extracted from actual hardware systems, but this process is complex, expensive, and rarely used outside of industry.

We use functional simulation to generate architectural checkpoints of TCP/IP-based workloads as well as to warm up cache state in preparation for detailed simulation and measurement. In both of these cases, the effective performance of the simulated system differs significantly from the performance seen using detailed timing simulation. We observe that the behavior of networking workloads depends on whether the connection bandwidth is limited by sender performance or by the network or receiver. We will refer to these scenarios as sender limited and receiver limited, respectively.

- In sender-limited situations, instantaneous performance changes are unlikely to cause packet loss. Instantaneous changes to the sender mean immediate changes to send rate, since there is no feedback required. Instantaneous changes to the receiver may yield different results. If the receiver increases in performance, there would be no change since sender determines overall performance anyway. If the receiver decreases in performance, tuning would only be required if the receiver's performance decreased to the point that it became the bottleneck.
- In receiver-limited situations, instantaneous increases in sender performance or decreases in receiver performance can lead to packet loss that may induce unstable behavior. Though TCP will eventually recover from packet losses, the recovery may not occur within a window of time that is feasible to simulate.

The primary contribution of this paper is that it is the first, to our knowledge, to identify and describe the issues that arise due to the interaction of self-tuning workloads and functional simulation. We also provide a detailed analysis of the impact of functional simulation on TCP-based networking workloads. While TCP itself is of great practical importance, these issues may apply to an increasing number of future workloads, as run-time profile-based software optimization becomes more widespread. Furthermore, many of the recent advances in simulation sampling have relied implicitly on workload timing independence; these techniques must be revisited and perhaps revised or even discarded when analyzing self-tuning workloads.

The next section presents a more detailed discussion of workload timing dependence and its impact on simulation techniques, including coverage of related work. The following sections describe the TCP protocol and our simulation environment, respectively. We then present the results of our experiments, and finally discuss our conclusions and directions for future work.

2. Discussion and Related Work

This section elaborates on the notion of workload timing dependence discussed in the introduction and relates this paper to prior work in the area.

Single-threaded applications were the first and are still the most prevalent type of workload used in architecture studies. Because individual threads in isolation have deterministic architectural behavior (including architectural register and memory contents and committed instruction order), and because application-only simulation does not model potentially non-deterministic interactions with the operating system (such as preemptive scheduling), the architectural behavior of these workloads is completely unaffected by execution timing. Only these workloads are truly timing independent. In this case, the architectural state generated by a fast functional simulator for a given program point is guaranteed to be identical to the architectural state of any (correct) detailed timing simulator at the same program point.

In contrast, multithreaded applications running on multiprocessors or dynamically scheduled onto uniprocessors (e.g., using SMT) are not fully timing independent. System timing effects, such as cache hits and misses, will lead to different rates of progress for different threads, causing variations in the interleaving of thread execution. These variations can have subtle architectural effects even in deterministically scheduled programs, such as the number of iterations a thread sits in a spin loop. More commonly, inter-thread synchronizations are not fully deterministic, and timing variations that, for example, change the order in which threads acquire locks may have significant effects. Goldschmidt and Hennessy [7] showed that in some situations this timing dependence could lead to incorrect results from trace-driven simulations, and recommended the (now widespread) use of execution-driven simulation. Alameldeen and Wood [1] showed that this dependence can be particularly severe for complex multiprocessor server workloads on full-system simulation (i.e., including the operating system), where the combination of different processor interleavings can lead to different OS scheduling decisions and different service orders among client transactions. They recommend combining multiple runs randomized using small perturbations to gain statistical confidence in the simulation results.

Although the timing dependence of these multiprocessor workloads is a significant issue, it is qualitatively distinct from the timing dependence we explore in this paper. In the former case, timing variations can lead the system down different but equally valid archi-

tectural execution paths. Specifically, each of the potential paths followed under one timing scenario is also a legitimate potential path under other timing scenarios. Alameldeen and Wood's randomization technique samples the space of valid paths so that statistically significant conclusions can be drawn about architecturally invisible changes in a system's configuration (e.g., in cache associativity or coherence protocol) without interference from the perturbations in architectural execution they cause. We call these workloads *weakly timing dependent*.

In contrast, we are concerned with workloads that explicitly tune themselves to the performance of the underlying system. These workloads take performance feedback from the underlying system (directly or indirectly) and incorporate this feedback into the control flow of the workload. For example, if TCP packets arrive at a host faster than they can be processed, kernel buffers will eventually fill and packets will be dropped. These packet drops will be detected by TCP, causing not only retransmissions but also a geometric decrease in the sender's window size, fundamentally restricting the connection's bandwidth. Thus timing variations lead to changes in the architectural execution in such a way that the path taken under one set of timing conditions may be substantially different from any execution path that the system would take under another set of timing conditions. As such, the architectural states generated along the former path may be states that could never be reached under other timing conditions. We call such workloads *strongly timing dependent*. Lim and Agarwal's reactive synchronization algorithms [9] are another example of a strong timing dependence.

The distinction between strongly and weakly timing-dependent workloads is not strict. A multiprocessor server running a TCP-intensive workload would exhibit both strong and weak timing dependencies. This paper focuses solely on the strong timing dependence aspects of self-tuning workloads. The interaction of these types of dependences is a topic for future research.

The key point of this paper is that strongly timing-dependent workloads may not be compatible with both traditional and recently proposed techniques for accelerating architectural simulations. Even a purely functional simulation of a timing-dependent workload must have some notion of time. For example, the rate of TCP packet arrivals must be specified outside of the architectural behavior of the system, leading to some relative timing relationship between network bandwidth and CPU execution rate. During functional simulation, a strongly timing dependent workload will adapt its behavior to the effective observed performance of the system. If a fast functional simulator is used to generate initial checkpoints for detailed modeling or to fast-forward between detailed sampling points for such a workload, the architectural state so generated may not be valid for the detailed system timing that the simulator is modeling.

As we will show, functional simulation is still useful as long as the detailed timing model can be run long enough for the workload to re-tune itself to the actual performance of the simulated system. However, this requirement directly conflicts with the approach of the SMARTS system of Wunderlich et al. [12], which advocates performing detailed modeling on numerous but extremely brief samples separated by fast functional execution. Another possible approach to strongly timing dependent workloads which would be compatible with SMARTS is to attempt to match the effective performance of the functional simulation with the detailed model. However, it is not clear how accurately this goal could be achieved, and we do not explore it further in this paper.

Sherwood et al.'s SimPoint technique [10] is another recent advance in simulation methodology. Their approach identifies a priori a set of representative portions of program execution. These portions are simulated using a detailed model and combined with appropriate weights to generate an estimate of the full program's performance. The benefit of the technique comes from the fact that the representative portions are identified once for all timing models using a fast functional execution of the entire program. This approach is clearly useful for timing-independent workloads. However, the applicability of SimPoint to strongly timing-dependent workloads depends on whether the portions of execution that are representative under one timing scenario (e.g., the implicit timing of the functional simulator) are also representative of other executions under other timing configurations. While this condition may hold true, further investigation is required. SimPoint has been applied to simulation of weakly timing-dependent workloads consisting of multiple single-threaded applications on an SMT processor [5]. This work exploits the timing-independent nature of the individual threads to simplify the analysis of potential dynamic interactions between them.

3. TCP/IP Overview

In this section we discuss the TCP/IP protocol and the way in which it tunes itself to the available end-to-end bandwidth capability of a connection.

3.1 Basic TCP/IP Operation

TCP/IP is a ubiquitous network protocol used to transfer data across the Internet. The sender transmits data in packets to the receiver. As the receiver processes data, it sends back acknowledgement packets (ACKs) to the sender to indicate the data it has received. Each packet contains a TCP header that identifies the connection and includes other information about the connection's status.

3.2 TCP Flow Control

TCP flow control is the attempt at the sender to match its sending rate to the processing rate of the receiver. In the header of each TCP packet, the receiver indicates the amount of kernel buffer space it has available for the connection. The sender's network stack maintains this value internally in a parameter called the send window, and ensures that the number of outstanding (unACKed) packets for that connection does not exceed this send window.

3.3 TCP Congestion Control

The terms congestion control and flow control are often used interchangeably, but they are two distinct algorithms in TCP. Congestion control refers to the sender's attempt to match its rate to the capability of the network through which packets are transmitted. If packets from the sender are being lost en route to the receiver (dropped by routers, etc.), the sender will reduce its sending rate regardless of the amount of buffer space available at the receiver. Because IP packets may be silently dropped at any point in the route, packet losses are inferred from receiver ACKs and timeout events. Loss events cause adjustments to a sender-internal parameter called the congestion window. The sender may not have more unACKed data than specified by the congestion window.

The congestion control algorithm has three main phases: (i) slow start, (ii) additive increase/multiplicative decrease (AIMD), and (iii) timeout event recovery. Connections always begin with the slow start phase, where the congestion window is initialized to one packet and grows exponentially with every ACK received. When the congestion window reaches a certain threshold, the protocol moves into the AIMD phase, where every successful ACK received results in an additive increase in the congestion window. If the sender observes from the receiver's ACK information that a packet appears to have been dropped, the congestion window is cut in half (i.e., a multiplicative decrease), but the connection stays in the AIMD phase. If the sender does not receive any ACKs from the receiver for a sufficient period, a timeout occurs, and the connection attempts to recover by returning to the slow start phase.

All common TCP implementations adhere to this general framework. There are many additional details that vary across specific protocol implementations (e.g., Tahoe, Reno, and Vegas), but these details are not critical to this paper. The important thing to note is that packet losses can result in severe throttling of the sender's transmission rate.

3.4 Tuning Latency

Except for timeouts, TCP flow and congestion control at the sender is based on information contained in the ACK packets from the receiver. As a result, the time required for a TCP connection to tune itself is primarily a function of the round-trip time (RTT) between a sender and receiver. In this paper, most of our simulations involve two systems directly connected by a single low-latency link. This setup minimizes the TCP self-tuning interval. We also do a few experiments with a higher latency link as well, in order to demonstrate the effects of RTT on tuning delay.

4. Methodology

We performed our experiments with M5 [6], a full-system simulator designed to support network-oriented workloads. It implements models for CPUs, caches, memory, I/O devices, and bus-based interconnects. M5 has both a detailed out-of-order CPU model and a simple in-order functional CPU model which can operate with or without caches. For the network interface, M5 models the National Semiconductor DP83820 network interface controller (NIC). We boot a Linux 2.6.8.1 kernel on the simulated hardware.[1]

4.1 System configurations

In this paper, we model three different CPU/cache combinations:

- The *pure functional* (PF) model uses the simple CPU model with no caches and a single-cycle memory latency. This model is used primarily for generating checkpoints.

1. Our simulated DP83820 fixes a bug in the actual device that prevents unaligned DMA transfers; we also patched the device driver to eliminate its workaround, allowing DMA transfers that automatically word-align payloads. We also patched a bug in the Linux kernel that does not enable checksum offloading correctly in some situations.

Table 1: System Under Test Parameters

	Parameter	Value
Detailed CPU	CPU width	4 insts/cycle
	Branch Predictor	hybrid local/global
	BTB	4k entries, 4-way assoc
	Instruction Queue	64 entries
	ROB	128 entries
Memory System	DL1/IL1	128kB, 2-way assoc, 64 byte blocks IL1: latency 1 cycle, 8 MSHRs DL1: latency 3 cycles, 32 MSHRs
	L1/L2 Bandwidth	64 bytes/CPU cycle
	L2/Memory Bandwidth	16 bytes/25 CPU cycles
	HT Controller Latency	750 cycles
	Memory/HT Bandwidth	16 bytes/25 CPU cycles

- The *functional with caches* (FC) model uses the same simple CPU model but adds a two-level cache hierarchy. Because the CPU issues instructions in order, the caches are effectively blocking. This model is intended for cache warm-up.
- The *detailed* (D) model uses our detailed out-of-order CPU with non-blocking caches. This model is intended for gathering detailed performance statistics.

See Table 1 for detailed parameter values. To avoid I/O bus bottlenecks, we model a NIC attached to a HyperTransport-like bus which is attached to the memory controller.

The simple functional CPU has a multiplier option that allows the user to specify a maximum number of instructions to execute per cycle. We vary this parameter, setting it at either 1 or 8 to get lower or higher performance. We label the PF models as PF1 and PF8 to indicate the multiplier setting. In this paper, the FC model always uses a multiplier of 1.

In terms of effective simulated system performance, the PF8 and PF1 models are the fastest due to their idealized single-cycle memory latency. The FC model is slowest, because it incurs cache miss penalties and cannot overlap them with its simple 1-CPI blocking CPU. The D model has intermediate performance; although it pays cache miss penalties, its non-blocking caches and out-of-order superscalar execution mitigate their impact relative to FC.

Our network simulations all involve multiple systems (at least a sender and a receiver, and in some cases a gateway system). We model all systems within a single simulator process to guarantee accurate, realistic, and repeatable timing for the network protocol. However, in any experiment only one system is of interest, the system under test. To both reduce simulation time and drive the system under test at maximum load, we always use the PF8 model for all systems other than the system under test.

4.2 Benchmarks

Netperf [8] is a collection of network micro-benchmarks developed by Hewlett-Packard. Included are several benchmarks for evaluating the bandwidth and latency characteristics of various network protocols. We selected TCP stream, a transmit benchmark; and TCP maerts, a receive benchmark. In both of these benchmarks, the client (the system under test) connects to a server. The appropriate system then attempts to send data as fast as possible over the connection.

SPEC WEB99 (specweb) is a popular benchmark for evaluating the performance of webservers. It simulates multiple users accessing a mix of both static and dynamic content over HTTP 1.1 connections. It includes CGI scripts to do dynamic ad-rotation and other services a production webserver would normally handle. For our simulations, we use the Apache webserver [2], version 2.0.52. We also use the mod_specweb99 module, available on the SPEC and Apache websites, which replaces the reference Perl CGI scripts with a more optimized C implementation. We also use a custom client based the Surge [3] traffic generator. This modified client uses the same statistical distributions as the standard SPEC WEB99 clients, but is lighter weight (to reduce simulation time) and attempts to saturate the webserver rather than maintain a fixed bandwidth as the standard client does.

In addition to the back-to-back configuration of the two previous benchmarks, we also created a network address translation (NAT) configuration for the netperf maerts benchmark, where the client communicates to the server through a NAT gateway. The NAT gateway masks the IP address of the client system from the server. NAT gateways are commonly used to connect an entire private network through a single public IP address. In this configuration, the NAT gateway machine is the system under test.

4.3 Checkpointing

We use checkpointing to avoid the overhead of booting multiple simulated machines and initializing the benchmarks for each experiment. Special instructions embedded in the benchmark scripts cause M5 to save

off all architectural state. Later simulations can then begin from the specified checkpoint, using whatever operating mode is desired by the user. For all configurations, checkpoints are generated after connections are established and network traffic has ramped up.

We generated four checkpoints for each of our benchmarks. The differences varied on two axes: whether the system under test was checkpointed in PF1 or PF8 mode, and whether the wire latency was 0 or 400 µs. All systems not under test used PF8 mode. All CPUs were clocked at 4 GHz.

4.4 Experiments

This paper focuses on the effects of traditional simulation acceleration techniques on networking workloads. To that end, we copied two common techniques, both starting from a purely functional (PF) checkpoint: running a detailed (D) timing simulation directly, and using a functional-with-caches (FC) interval to warm up cache state before the detailed timing simulation (FC→D). We varied the length of the FC cache warmup to be 125 million, 250 million, 500 million, or 1 billion cycles. Prior experimentation with showed that netperf performance is extremely stable, so we ran the detailed portion for 100 million cycles. We ran specweb for 1 billion detailed cycles because of its lack of steady behavior. Statistics were sampled every 10 million cycles in both cases.

5. Results

The results for the netperf stream benchmark on a zero-latency network are shown in Figure 1. For this and most remaining figures, the graphs plot the total wire bandwidth of the system under test for each 10 million cycle interval from the point immediately after loading the checkpoint. The legend indicates whether the checkpoint was taken with a PF1 or PF8 CPU. For space reasons, we show only the 500 million cycle cache warmup results for FC→D.

In looking at the stream graphs, there is no significant bandwidth settling period after any CPU model transition. There is also no apparent difference as to whether the checkpoint was taken with a PF1 or PF8 CPU. Recall that netperf stream exercises the client's ability to send data. Because the client (the system under test) is the bottleneck, this scenario is sender limited (as discussed in Section 1). The client's send rate is not constrained by the state of the network or receiver's free buffer count, so it can adapt quickly when its performance suddenly changes. Thus for netperf stream (and likely other sender-limited work-

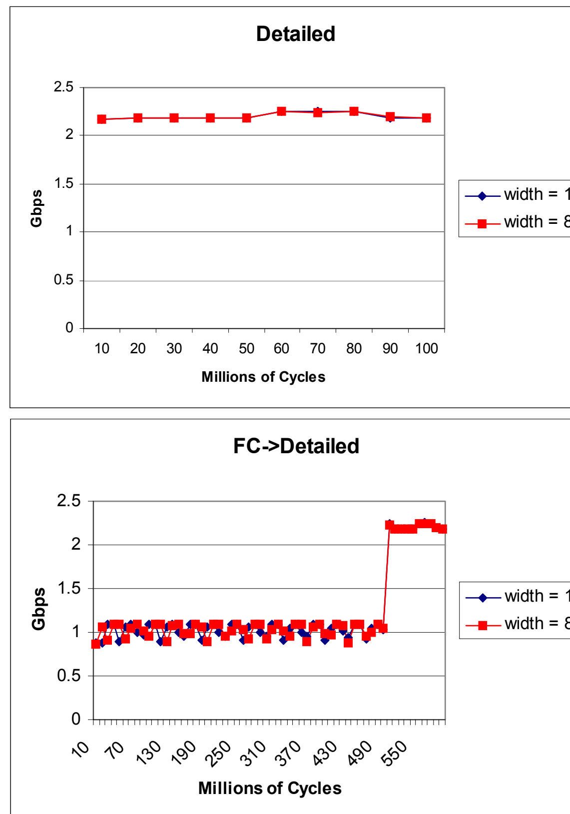


Figure 1: Netperf stream bandwidths. The legend indicates the width of the system under test CPU during the checkpoint. Since stream is a sender-limited benchmark, the transitions are smooth and stable.

loads), there is no need for a significant TCP tuning interval to reach the new steady-state performance.

The netperf maerts benchmark (Figure 2) tests the client's receive performance. When the client system changes instantaneously, it takes time for the effects of this change to propagate to the server sending the data. In the detailed case, the slowdown from the PF8 checkpointing CPU to the detailed CPU is drastic enough to require some tuning. Longer runs show that the true steady state for this benchmark under our configuration is about 5.7 Gbps. The PF8→D transition requires 70 million cycles to stabilize to this level. The bandwidth gets noticeably greater while it is tuning due to retransmission traffic. The PF1→D transition also requires a bit of time, but is harder to see in the graph. With some statistical analysis, we find the coefficient of variation (CoV) of Gbps for the 10 samples of the detailed run is 1.66%. However, considering the first sample as the tuning period, and only measuring the latter 9 samples drops the CoV to 0.5%. Further lengthening the tuning period does not reduce the CoV any more. Thus we can

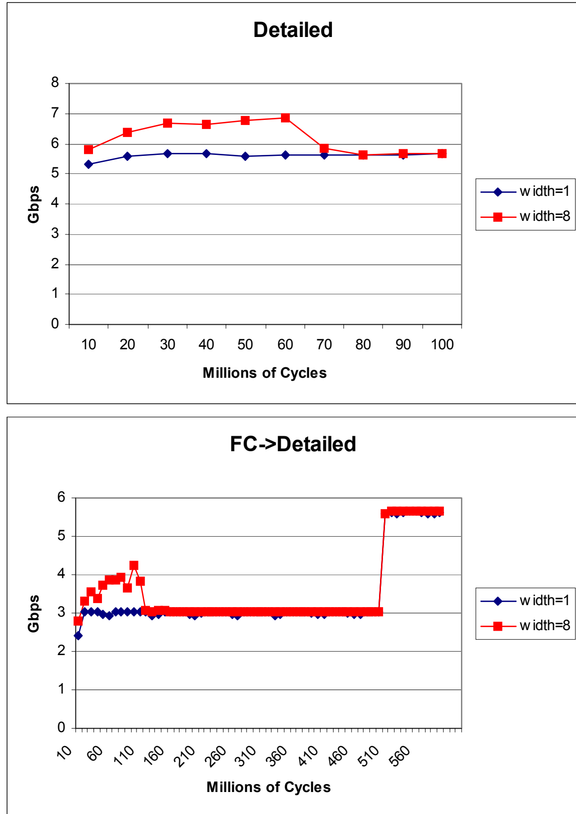


Figure 2: Netperf maerts bandwidths. The legend indicates the width of the system under test CPU during the checkpoint run. Since maerts is a receive-limited benchmark, CPU transitions require some tuning before becoming stable.

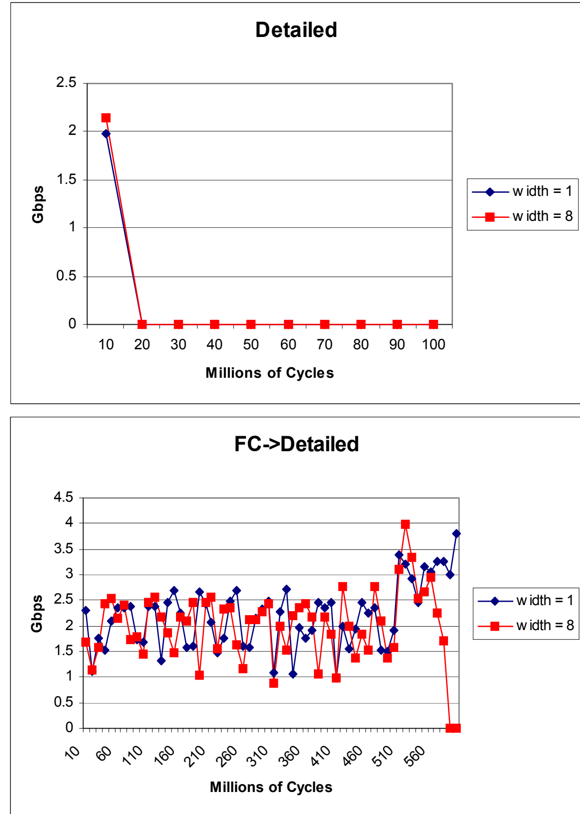


Figure 3: Bandwidth for netperf NAT configuration. In the top graph, the drastic transition from PF to D is exacerbated by the long network latency caused the by NAT gateway, causing the simulation to become unstable and effectively come to an end.

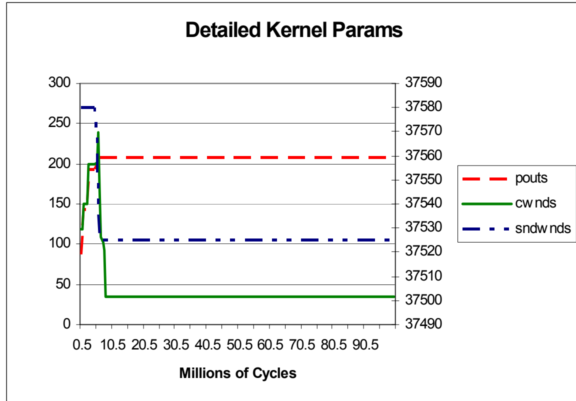
conclude that this tuning period requires no more than 10 million cycles.

In the FC→D case, there is a much lengthier stabilization period going from the PF8 checkpoint into the FC mode than from the PF1 checkpoint. The PF8 CPU is faster than PF1, so the performance mismatch is greater and requires greater adjustment. However, both cases stabilize well before the change from FC to D. The FC to detailed transition is quite smooth, likely since it involves a speedup of the CPU rather than a slowdown. The transitions in the graph are nearly instantaneous; the first sample is not measurably different from all the remaining samples.

We conclude that, for receiver-limited benchmarks such as netperf maerts, some significant transition time is necessary for TCP to retune itself whether it is transitioning directly from a pure functional checkpoint (or fast forward) to detailed simulation or to a functional cache warm-up period. The length of this transition time is a function of the severity of the CPU mismatch between the two phases: from 10 to 70 million cycles for PF→D and 10 to 125 million cycles for PF→FC, with the larger values corresponding to the PF8 checkpoint. TCP tuning time after the FC→D transition is negligible because the effective receiver performance is increasing rather than decreasing. With M5, using a minimal FC phase for tuning yields a shorter total simulation time than transitioning directly to detailed from the FC checkpoint.

For the NAT configuration of netperf maerts, the system under test is the NAT gateway. If coming out of the checkpoint moves the system to a slower CPU model, the gateway slows and injects more delay into an already high-latency network between the server and client. Sometimes the gateway slows to the point that it loses packets, which takes an extremely long time for TCP to detect and recover from.

Figure 3 shows the bandwidths of all the NAT maerts runs on a zero-latency network. When transitioning from PF directly to detailed, the simulation effectively ends. The sender, not knowing about the degradation of the network, continues to send packets at a high rate



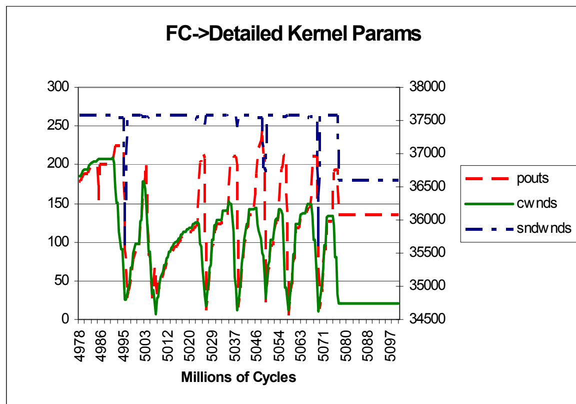
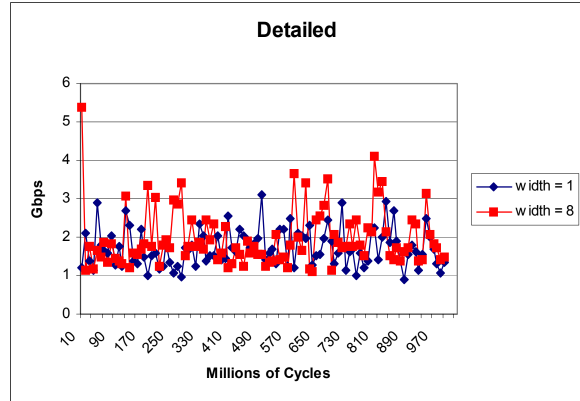


Figure 4: TCP kernel parameters for NAT maerts runs that manage send rate. Pouts and cwnds are in packets, and go with the left y-axis, while sndwnds are in bytes and go with the right y-axis.

that the newly slowed gateway cannot handle. The gateway thus drops packets. In looking at the packet traces, we found they were riddled with retransmissions and lost packets. Towards the end, there was a long stream of duplicate ACKs sent by the receiver. Duplicate ACKs indicate that the receiver has missed a packet in the sequence and requires a retransmission.

In contrast, the simulation continues transmission through the FC phase after the PF to FC transition with both checkpoints, though without stabilizing. However, the PF8 checkpoint run ceases transmitting at the FC→D transition. The cause for this behavior is not clear since the FC→D transition actually represents a speed increase.

To gain insight to what occurred on a TCP level, we tracked some TCP stack parameters for the runs starting from the PF8 checkpoints. Figure 4 shows some TCP parameters maintained by the server to manage its send rate to the client. Cwnd is the congestion window, sndwnd is the send window (which represents the amount of kernel buffer space available in the receiver that is sent via TCP header), and pouts is the number of



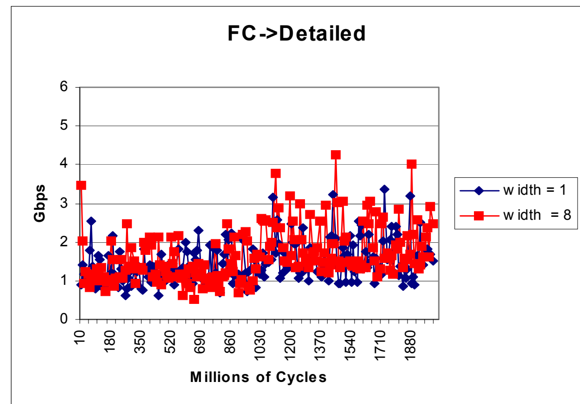


Figure 5: Specweb bandwidths.

packets in flight at the time (packets out). Notice that in the top graph (showing the PF→D case), the receiver's advertised kernel buffer space and the sender's congestion window size plummet immediately after transition. The slowdown from PF to D is so drastic that these parameters actually drop below the packets-in-flight count. However, the number of packets in flight is not supposed to exceed the congestion window, preventing the sender from transmitting additional packets. Meanwhile, the receiver cannot make progress without the retransmission it is asking for with the duplicate ACKs. In a real system, these situations are solved by TCP timeouts and resets, but this process takes far too long to simulate.

A similar parameter drop-off happens in the bottom graph at the FC→D. We are unsure why this fall-off occurred at the transition from a slower CPU to a faster one, and not at the PF→FC transition as might be expected. We plan to look into this situation further as future work.

Results from the specweb benchmark are shown in Figure 5. Recall that our specweb runs differed from netperf in that the detailed simulations ran for 1 billion cycles. Even on this time scale, the benchmark is inher-

ently less stable than netperf, though on even larger time scales it can be quite stable [1].

With specweb the system under test is the server, and it serves web pages requested by the client. It is difficult to see with the naked eye whether there are TCP tuning effects. We combined the detailed bandwidth samples into 150-million-cycle intervals and measured the CoV of these aggregated samples. For the direct PF→D runs, the CoV drops from roughly 8% to 3% when taking a 300 million cycle transition phase into account. The FC→D case transitions more quickly, needing only a 150 million cycle warmup to reduce the CoV near minimum. This implies that, like with the maerts benchmark, the FC phase takes some bite out of the TCP tuning period. However, since the total simulation is only 1 billion cycles long, it is difficult to be confident that these samples are truly representative. Specweb is sufficiently more complex than netperf that further study is needed to make definitive conclusions. Being a sender-limited case, it may not require much tuning time at all (as with netperf stream).

The above graphs were all of simulations using zero-delay wires. When link delays are thrown into the mix, the problem of tuning time is exacerbated further. Figure 6 shows results from the netperf maerts FC→D run with a 400 µs wire delay. This wire delay increases the round-trip time (RTT) by 800 µs, increasing the tuning time significantly.

The top graph of Figure 6 compares measured bandwidth between the zero-delay and 400 µs-delay runs. The simulation with link delay has not reached steady state by the end of the simulation. The bottom graph shows the TCP stack parameters from the 400 µs-delay run. The beginning part of the graph shows how the CPU transition disrupts the simulation and forces the geometric decrease of the congestion window. Packets in flight must follow. The steep increase that follows is the TCP slow-start phase, where the congestion window increases exponentially with each ACK. The next phase is the AIMD phase of TCP described in Section 3.3. Recall that in this phase of operation, the congestion window increases by one packet with each ACK. Since ACKs are delayed by the RTT, the increase in cwnd does not happen nearly as quickly as in the previous experiments.

In reality, 800 µs is a modest RTT. While the RTT from our desktop machines to local University of Michigan servers is roughly 300 µs, RTTs for non-local servers are typically tens of milliseconds, with well-connected international sites (e.g., www.cern.ch) over 100 ms. Thus simulating realistic Internet link delays requires particular care, and may not be practical without novel techniques. Maerts is capable of achieving much higher bandwidths, and taking data from these simulations would not give accurate results.

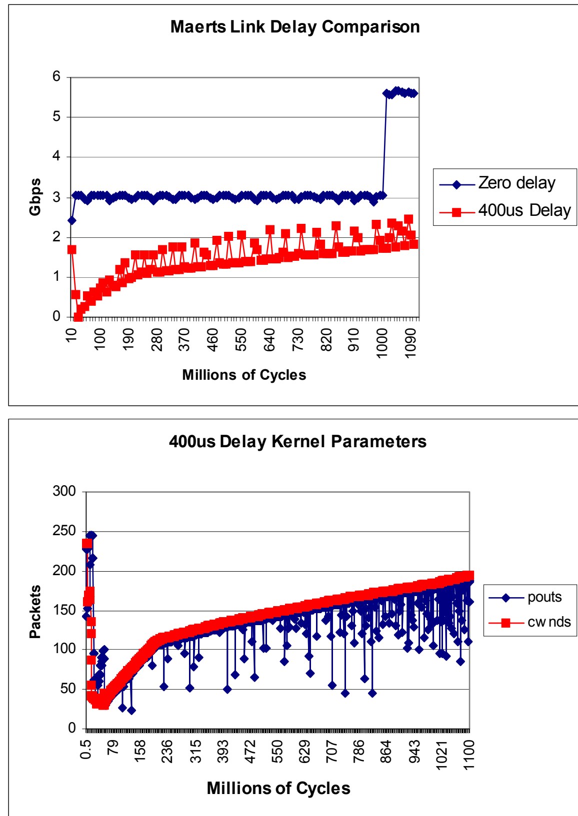


Figure 6: Top: Contrasts between zero delay wires and 400us delay wires. Bottom: TCP congestion windows increase by one with every ACK received. When ACKs are delayed by the link, the cwnd parameter is slow to increase, thus slowing TCP tuning time.

6. Conclusions and Future Work

It is clear that for many cases of networking benchmarks, conventional methods of simulation acceleration via functional simulation may induce artificial behavior. TCP/IP is a self-tuning protocol; when run on a purely functional system model, it will tune its performance parameters to the meaningless effective performance of the functional system. The effects of this artificial tuning range from incorrect data to completely unstable behavior.

We observed that there are two classes of simulations: sender-limited and receiver-limited. The netperf stream benchmark and specweb are examples of the sender-limited case, whereas netperf maerts and NAT are examples of receiver-limited cases. Spontaneously changing sender performance in sender-limited cases

generally does not require noticeable TCP tuning time, as the sender can autonomously adjust its transmission rate. However, spontaneously changing receiver or network performance in receiver-limited cases can lead to dropped packets because of the propagation delay in informing the sender of the new environment. Often, allowing TCP to tune itself before taking measurements is sufficient. However, in some cases, these dropped packets can cause a simulation to yield no meaningful results.

Where retuning is needed, the time required is significant and can vary dramatically depending on the situation (from 10 to 150 million cycles in our experiments, given zero wire delay). Thus, fast-forwarding paradigms like the SMARTS method of simulation acceleration (which advises 1000 cycles of detailed simulation following a period of fast forwarding and functional warming) would very likely result in invalid data when testing networking benchmarks. The retuning period is also a strong function of network round-trip time; when realistic network latencies were added between the sender and receiver, the retuning time quickly became significantly larger than is practical for simulation.

We also observed that TCP adjusts more quickly and with more stability when effective system performance increases rather than decreases. We thus had fewer problems when checkpoints were generated using a purely functional model with lower effective performance (PF1 vs. PF8). Intermediate warm-up using the functional-with-caches (FC) model also reduced tuning time, as this mode is not as slow to simulate as detailed, and the next transition from FC to detailed is an effective performance increase.

The problem of packet drops leading to completely disrupted simulations does not appear to have a simple solution. We have found that often just taking the checkpoint at a different point in execution can sometimes prevent this situation, but we do not have a rigorous method for distinguishing such points. Understanding and addressing these pathological cases is an area of future work.

Other promising directions for research include characterizing needed warmup and transition times in more detail and finding techniques to deal with scenarios that involve realistically large RTTs.

While many of our detailed conclusions are tied to fundamental characteristics of TCP/IP, the issues raised in the paper applies to other strongly timing dependent workloads as well. We expect that future sophisticated run-time systems will employ dynamic optimization techniques aggressively. Architects must take care in these situations as well that the software layers do not tune themselves inappropriately to the artificial effective performance of functional system models.

7. References

[1] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003.

[2] Apache Software Foundation. Apache HTTP server. http://httpd.apache.org.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[4] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proc. 1995 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 14–24, 1995.

[5] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proc. 2004 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, March 2004.

[6] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[7] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulation of multiprocessors. In *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

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

[9] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 25–35, October 1994.

[10] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, October 2002.

[11] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proc. 1996 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[12] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. 30th Ann. Int'l Symp. on Computer Architecture*, pages 84–97, June 2003.