Network-Oriented Full-System Simulation using M5

Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt

***Abstract*—As network I/O bandwidths scale up to multiple gigabits per second, the technical challenges of dealing with these high data rates will become a driving factor in computer system design. Unfortunately, investigation of network-oriented system design issues is hampered by a lack of suitable simulation tools.**

This paper presents M5, a simulation system targeting network-intensive workloads. M5 is capable of simulating multi-system networks within a single process. Within each simulated system, M5 provides a detailed performance model of the I/O subsystem, including the bus timing and coherence effects of network DMA transfers. Because the majority of networking code is in the operating system, M5 models system hardware faithfully enough to run an unmodified commercial OS kernel.

We show that to accurately model a network-intensive workload, DMA effects must be modeled with detail. Omitting these effects can lead to inaccuracies of up to 75x for performance statistics such as cache misses.

I. Introduction

The importance of high-bandwidth TCP/IP networking in modern computer systems is tremendous and continues to grow. The criticality of networking to conventional web, database, and file servers is obvious. In addition, the rising popularity of packet-switched I/O architectures and network-attached storage indicates that nearly all I/O on future servers may take the form of network traffic. General-purpose systems also play roles as network infrastructure components—e.g., routers, firewalls, VPN endpoints, or overlay network nodes—displacing dedicated networking hardware thanks to their greater flexibility, relative simplicity of administration, and potentially lower cost. Efficient networking support is of growing interest even for end-client systems, given the increasing popularity of peer-to-peer distributed systems and the potential demand for high-bandwidth video services.

The technical challenges presented by network I/O are growing in tandem with its commercial importance. Network bandwidths have been increasing at a rate outpacing even CPU performance gains: the high-end Ethernet standard moved from 100 Mb/s in 1995 to 10 Gb/s in 2002—a hundred-fold improvement. Even the compounded doubling of CPU performance every 18 months provides only a twenty-five-fold speedup over this same period. Given these trends, support for high-performance networking must move beyond the boundaries of the I/O subsystem and become a system-wide design consideration. Unfortunately, work in this area is hampered by the dearth of appropriate tools for analyzing the interaction of network I/O with overall system architecture. As a result, researchers have neither a sufficient understanding of how network I/O impacts system performance nor an available vehicle for evaluating design decisions and architectural enhancements in the context of network I/O.

To address this situation, we have developed a new simulation system, called M5, which targets network-intensive workloads. M5 is capable of simulating multiple systems—e.g., a server and several clients—plus the network interconnect within a single process, allowing detailed control over simulated network characteristics such as bandwidth and latency. Within each simulated system, M5 provides a detailed performance model of the I/O subsystem, including the bus timing and coherence effects of network DMA transfers. Because networking applications spend a significant portion of execution time in the operating system, M5 models system hardware faithfully enough to run an unmodified commercial OS kernel. M5 provides numerous other key features, such as a detailed, execution-driven out-of-order simultaneous multithreading (SMT) CPU model; an event-driven memory system supporting multiple levels of cache hierarchy interconnected with split-transaction busses; and support for cache-coherent bus-based multiprocessor systems. Furthermore, M5 is written in a modular fashion, using C++ objects to encapsulate key abstractions such as CPUs, caches, busses, I/O devices, etc. A flexible runtime configuration language allows users to specify the desired object types and parameters and their interconnection.

Our preliminary results indicate that accurately modeling network I/O has a significant impact on the memory-system behavior of SPECweb99.

This paper provides an overview of M5 (Section II), a brief description of our SPECweb99 workload (Section III), and preliminary results from M5 on the impact of network I/O on system performance (Section IV). We finish with related work (Section V) and conclusions (Section VI).

II. The M5 Simulator

M5's primary goal is to provide a modular, configurable simulation environment for a wide range of architectural studies. M5's object-oriented structure is a key enabler in meeting this goal. Section II.A discusses the benefits and features of M5's object-oriented design in detail. Section II.B discusses how complex simulated systems are described using M5's con-

This work was supported in part by the National Science Foundation under grants CCR-0105503 and CCR-0219640, by gifts from the Intel Corporation, and by a Sloan Research Fellowship.

Authors are with the Advanced Computer Architecture Laboratory, EECS Department, University of Michigan, Ann Arbor, MI 48109-2122. (email: {binkertn,ehallnor,stever}@eecs.umich.edu).

figuration mechanisms. Section II.C gives a brief overview of the component models currently implemented within M5. Sections II.D and II.E discuss M5's support for full-system simulation and I/O modeling.

A. Object Orientation

Internally, M5 is written in C++, and all major simulated structures are implemented as C++ objects. Externally, M5's configuration language allows users to instantiate and interconnect these objects flexibly to model a wide variety of system configurations. This section discusses several synergistic benefits of M5's object-oriented framework, including object isolation, object interchangeability, component sharing, object replication, and the simple provision of common object behavior.

1) Object Isolation: One key benefit of object orientation is compiler-enforced isolation of internal object details. Careful use of C++'s "private" and "protected" access controls prevents arbitrary interactions between objects, making M5 more maintainable and flexible. In particular, having a clear interface between a simulation object and the rest of the system makes it more likely that researchers can modify a component's behavior using only localized code changes and without breaking seemingly unrelated parts of the simulator.

Limiting inter-object communication to well-defined interfaces also aids in maintaining realistic simulation models. A simulation component can export only that functionality which could be reasonably expected of an actual implementation. An invocation of a method on another object typically indicates a physical connection between components. Of course, the flexibility to model unrealistic components or interactions still remains (e.g., to perform limit studies), but ideally these less realistic features are clearly indicated as such in the documentation.

2) Object Interchangeability: M5 promotes object interchangeability by standardizing object interfaces for key component types. Multiple models for a particular component, such as a CPU, can be substituted easily within a particular configuration without impacting other simulated system components. These interchangeable models may differ in level of detail (allowing simulation speed vs. accuracy trade-offs), in the component's functional behavior (to study design alternatives), or both.

Specific component models are selected during a runtime initialization step using the configuration process described in Section II.B. Runtime component selection is enabled by using C++ interface inheritance and virtual functions. A C++ base class defines a standard interface for a particular type of simulation object, such as a CPU or cache, using virtual functions as appropriate. Different realizations of these objects all derive from the base class, and are thus interchangeable within the simulator. For example, two CPU models—a fast functional model and a detailed out-of-order timing model—are both implemented as classes derived from M5's base CPU class, and are both compiled into the M5 executable. Either of these models can be selected at program initialization and plugged in to an otherwise unmodified system configuration. Incorporating both models in the same executable is important for network simulation, where the system being studied (e.g., the server) may require the detailed CPU model, but other systems (e.g., the clients) can use the fast CPU model to take advantage of its much greater simulation performance.

3) Component Sharing: One of M5's long-term goals is to simplify sharing of component models among researchers in different groups. A shared simulation infrastructure with sharable components would allow innovation to proceed at a quicker pace by enhancing the repeatability and comparability of experiments and lowering the barriers to collaboration.

Object isolation and interchangeability contribute directly to achieving this goal. Ideally, researchers can develop or enhance models in their area of expertise and distribute them to other M5 users. Researchers in other areas can plug these enhanced models directly into their simulations. For example, a group researching enhanced memory systems might easily drop in an alternate CPU model from another source. Researchers in the same area can compare alternatives by evaluating various components in an identical environment.

Of course, this ideal scenario exists only to the extent that innovation can take place within M5's defined interfaces; we expect that these interfaces will evolve over time to reach the desired level of flexibility. However, even for changes which span multiple objects, M5's structure should clearly compartmentalize the modifications.

4) Object Replication: Another advantage of object orientation is that objects fully encapsulate a component's state, allowing easy replication. Given a CPU object class, a basic multiprocessor can be simulated simply by instantiating multiple CPU objects and connecting them to a common bus. (The resulting model works, but in a very rudimentary fashion; to simulate interesting multiprocessors in M5, we also added a coherence protocol to the caches and bus model.)

This replication capability is critical to our network-oriented research. Because a simulated system is a collection of objects (CPUs, caches, memories, etc.) with no global variables, we can simulate a multi-system network by instantiating multiple collections and interconnecting them with a network link object.

5) Common Object Behavior: Object orientation allows M5 to provide common features and behaviors to all simulation components. All component model objects derive from a common base class, SimObject. M5 furnishes a significant infrastructure based on the SimObject class; any class derived from SimObject inherits common mechanisms for configuration, naming, instantiation, and parameter handling. Many abstract entities (e.g., workloads) also inherit from SimObject so that they can leverage this infrastructure.

SimObject instantiation is managed in part by creating "builder" (a.k.a. "factory") objects for each component class. A builder object maps an external object class name to an internal C++ class, a set of parameters, and an instance creation

```
Class Cache : public SimObject { ... }

REGISTER_SIM_OBJECT("BaseCache", Cache)

BEGIN_DECLARE_SIM_OBJECT_PARAMS(Cache)
    Param<int> size;
    Param<int> assoc;
    Param<int> block_size;
    SimObjectParam<Bus *> out_bus;
END_DECLARE_SIM_OBJECT_PARAMS(Cache)

BEGIN_INIT_SIM_OBJECT_PARAMS(Cache)
    INIT_PARAM(size, "capacity in bytes"),
    INIT_PARAM(assoc, "associativity"),
    INIT_PARAM(block_size, "block size in bytes"),
    INIT_PARAM(out_bus, "bus to next level of memory")
END_INIT_SIM_OBJECT_PARAMS(Cache)

CREATE_SIM_OBJECT(Cache)
{
    int nsets = size / (assoc * block_size);

    return new Cache(getInstanceName(), nsets, block_size, assoc, out_bus);
}
```

Fig. 1. Example SimObject declaration.

function. A small set of preprocessor macros hides the details of the internal C++ mechanisms used. Figure 1 shows a simple example, where the external name "BaseCache" is associated with the internal Cache class (which inherits from SimObject). The code defines three integer parameters for the cache, which control the instance's capacity, associativity, and block size, respectively. The fourth parameter is a pointer to another component object of class Bus. The system interconnection structure is built by passing SimObject pointers as parameters to other SimObjects. The final block of code in Figure 1 is the creation function, which uses the externally visible parameters to generate arguments to the Cache object's constructor. Although greatly simplified relative to M5's actual cache model, this example is complete, excepting the definition of the Cache class itself. Simply compiling this code and linking the resulting object into the M5 executable makes the "BaseCache" component visible to users. The remainder of the configuration process will be described in more detail in Section II.B.

M5 also uses inheritance to provide a framework for serialization (checkpointing) of simulation state. An abstract base class (Serializable) provides a virtual function interface for saving and restoring internal object state to a file. Object developers need only implement these local functions to enable checkpoint and restore capability. M5's serialization framework automatically saves and restores the overall object structure, and invokes these methods on every instantiated Serializable object as appropriate. SimObject derives from Serializable, so component models automatically have access to this interface. The Serializable class is separate from SimObject to allow serialization of transient objects, such as scheduled events on the event queue, without forcing them to bear the configuration-related overhead of SimObjects.

B. Configuration Process

Because M5 is capable of modeling relatively complex networks of systems, it requires a powerful method for specifying simulation configurations. M5 uses a hierarchical specification language to allow reuse of specification data while maintaining maximum flexibility.

Raw specification data is provided to M5 using a section/keyword/value structure. The bulk of this data is loaded via ".ini" files, using syntax similar to that in Microsoft Windows configuration files of the same extension. Section names are enclosed in square brackets. "Keyword=value" lines bind string values to keywords in the scope of the current section. The left-hand side of Figure 2 shows an example configuration file with six sections. Keyword assignments can also be performed on the M5 command line.

Once the raw data is loaded, M5 parses it to create the simulation's configuration hierarchy. Hierarchy nodes are represented by data sections. The distinguished section name "Universe" represents the root of the hierarchy. The value of the "children" keyword within each section identifies the node's children in the hierarchy, represented as a list of *name*:*section* pairs. An optional "{*n*}" suffix indicates *n* copies of a given child. Data sections without a "children=" line represent leaf nodes. Note that a single data section can be used to specify several nodes within the hierarchy. The right-hand side

```
[Universe]
children=sys:System{2}    // creates sys0 & sys1

[System]
children=cpu:CPU bus:MemBus mem:MainMem

[CPU]
type=SimpleCPU
children=ic:L1Cache dc:L1Cache
icache=ic
dcache=dc

[L1Cache]
type=BaseCache
size=64K
assoc=2
block_size=64
out_bus=bus

// MemBus and MainMem sections not shown...

[sys0.cpu]
workload=workload0

[sys1.cpu]
workload=workload1
```

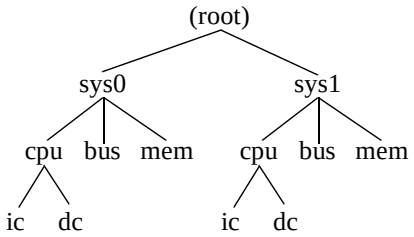

Fig. 2. Example configuration file and corresponding hierarchy.

of Figure 2 diagrams the configuration hierarchy corresponding to the example file.

After creating the hierarchy, M5 traverses it to create the indicated component objects. Each section with a "type" keyword instantiates a SimObject of the indicated type. Object parameter values are determined by parsing the values of the corresponding keywords in the data section. (Note that, in the "L1Cache" section, the keywords other than "type" correspond to the cache parameters declared in Figure 1.) Each object is given a unique name derived from its path from the root of the hierarchy. For example, the full name of the data cache in system "sys1" is "sys1.cpu.dc". (This name is provided to the object via the getInstanceName() method in Figure 1.)

Reuse of subtrees within the configuration hierarchy is supported by M5's name resolution algorithm for SimObject pointer parameters. An object name is resolved to a specific SimObject by looking for a match first at the same tree node as the referencing object, then, if no match is found, by traversing up the tree toward the root. Thus a particular object is visible to all objects below it in the hierarchy, but another object with the same base name can be used in a disjoint subtree. This algorithm allows detailed control over object replication and sharing. For example, all the caches in Figure 2 refer to "bus" as the value of their out_bus parameter. For "sys0.cpu.ic" and "sys0.cpu.dc", M5 resolves this reference to "sys0.bus", but "sys1.cpu.ic" and "sys1.cpu.dc" share "sys1.bus".

To provide additional flexibility, object parameters can be specified based on instance names as well as on configuration hierarchy nodes. This feature allows reuse of configuration subtrees even when repeated structures are not completely identical. For example, the two sections at the end of the configuration file in Figure 2 specify different workloads for the two simulated systems, although their configurations as described via the hierarchy are otherwise identical.

C. M5 Simulation Objects

M5 currently contains a variety of component models.

There are two primary CPU models: SimpleCPU, an in-order, non-pipelined, one-CPI functional model (similar to SimpleScalar's *sim-safe* [4]); and FullCPU, an out-of-order, superscalar, pipelined, simultaneous multithreading (SMT) CPU (originally derived from SimpleScalar's *sim-outorder*, but largely rewritten for additional realism and SMT support). Either of these models can be compiled to emulate the Alpha or SimpleScalar PISA ISAs. (Although heterogeneous-ISA multiprocessor simulations would not be too difficult to construct under M5, our build procedure currently does not support that option.) A third component, MemTest, follows the CPU object interface but internally generates random read and write accesses to exercise the memory system. As described in Section II.A.2, all three of these models are derived from the abstract base CPU class, and can be used interchangeably.

Although SimpleCPU is a monolithic object, the FullCPU model is partitioned into additional objects for greater flexibility. In particular, we have developed a family of instruction queue (IQ) models to support our research in this area [15].

Thanks to M5's object interchangeability, we can compare conventional IQ designs with more radical proposals simply by substituting an alternate IQ object in the configuration file.

M5's memory system is fully event-driven, allowing flexible and accurate modeling of timing and contention. We currently implement two cache models: a basic LRU cache and an indirect index cache [7]. The various levels of the memory hierarchy are connected by a coherent, split-transaction bus. The bus model operates in two modes: a "timing" mode, making liberal use of events to accurately model bus delays and contention; and a "non-timing" mode, which shortcuts the timing model to pass requests directly between caches and memory for simulation efficiency. Although either bus model can be used with any CPU, the former bus model is typically used with the FullCPU model and the latter with SimpleCPU. The cache models are oblivious to the bus timing mode.

D. Full-System Support

Providing full-system simulation in M5 involved implementing the CPU's privileged instructions and state, virtual address translation, platform devices such as timers and I/O bridges, and the I/O devices themselves. Our user-mode CPU simulators implemented the Alpha ISA (based on SimpleScalar's Alpha machine.def), which we extended to model the 21164's privileged state. We used SimOS-Alpha [3] as a reference for both the 21164's privileged-mode behavior and the Turbolaser Alpha server platform, though most of the needed code was rewritten from scratch or heavily modified to conform to M5's object structure and internal interfaces. Although SimOS-Alpha is capable of running only Tru64 V4.0, we further enhanced the Turbolaser model to allow M5 to boot a much more recent version of Tru64, V5.1. Tru64 V5.1 is an industrial-strength OS with a combination of advanced networking and multiprocessor support (including NUMA resource allocation) unmatched by current open-source operating systems. However, given the announced termination of Tru64 development, combined with the rapid pace of progress in the open-source community, Linux and/or a BSD flavor will eventually approximate Tru64's capability; at some point, we will likely switch to one of these more readily available OSes.

To aid in bringing up the system, we incorporated support in M5 for gdb's remote debugging feature. A gdb process running on a real machine can connect to a socket provided by M5, at which point the gdb process can be used to debug the Tru64 kernel running on the simulated system. In conjunction with a separate gdb process attached to the simulator itself, a developer can determine where the kernel is when a particular simulator condition is encountered. This debugging feature has been invaluable in making full-system simulation work.

E. I/O Modeling

Because of our focus on networking workloads, we paid particular attention to the accuracy of our network components. M5's network adapter device closely mimics the design of a modern Ethernet adapter. The model uses programmed I/O to talk to device registers and DMA to copy packets and descriptor data to and from the device. Both the network and disk devices use a detailed DMA timing model which schedules individual block transfers across the memory bus, invalidating the contents of the CPU cache appropriately. The simulated network interconnect models a fixed-bandwidth link between adapters. The current model allows us to accurately model well-designed 100Mb/s and 1Gb/s ethernet adapters. Tests of two simulated systems attached back-to-back with a 100Mb/s link have shown >90% link utilization.

III. Evaluation Workloads

To do our initial evaluation of M5, we chose netperf [9], a simple network link utilization microbenchmark, and SPECweb99 [19], a commonly used web workload benchmark.

A. Netperf

Netperf is a simple networking benchmark tool developed at Hewlett-Packard that is useful for quickly ascertaining the maximum bandwidth of a TCP connection between two machines. Because the tool does little other than generate network traffic, we found this tool to be ideal as a microbenchmark for M5.

Netperf has two components: netserver and netperf. Netserver is run on one machine and waits for netperf connections from another machine. Netperf, the client, connects to a netserver, does a short handshake, and then sends data to the server at the highest possible rate sustainable for 10 seconds. At the end of the 10 second run, netperf calculates the average bandwidth achieved over that period of time.

The inner loop in netperf is very short, basically filling up a buffer and calling the write syscall. This benchmark consequently has very little user time, and spends most of its time in the TCP/IP protocol stack of the kernel or in the idle loop waiting for DMA transactions to complete.

B. SPECweb

We chose SPECweb99 as a more realistic macrobenchmark workload. SPECweb99 includes both static and dynamic HTTP requests, and models a server which uses cookies to provide user-customized rotating advertisements. We use Apache 2.0.43 [1] as our web server.

We first built our workload on a testbed consisting of two Alpha systems, a 600 MHz 21164 for the server and a 500 Mhz 21264 for the client, directly connected by a 100Mb Ethernet link. Both systems ran Compaq Tru64 V5.1. We installed Apache using the worker multi-processing module (MPM) and with support for dynamic shared objects (DSO) enabled. We also turned off the access log to decrease the disk storage required.

The performance of the sample Perl CGI script included with SPECweb99 proved to be inadequate to drive the network link at sufficient bandwidth for a valid run. To remedy this, we modified a native Apache script, written by IBM for a

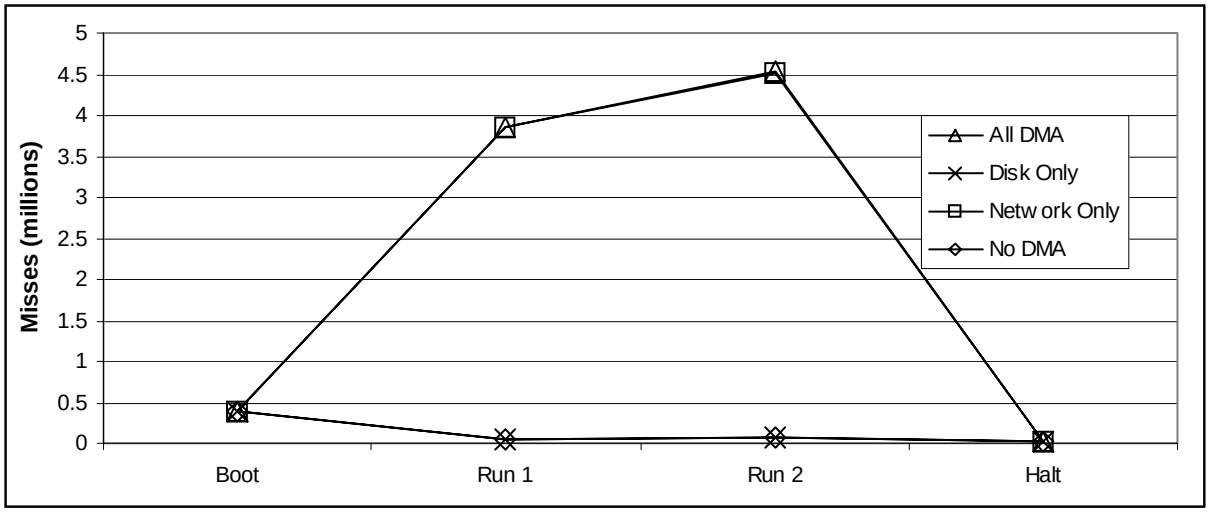

Fig. 3. Effect of DMA modeling on the number of Bcache misses for the netperf server.

SPECweb99/Apache submission on an IBM eServer, to work under our environment.[1] We compiled this script as a dynamic shared object and loaded it in to the web server at runtime. After additional tuning of Tru64 shared memory and file descriptor limits, this script enabled us to complete a valid SPECweb99 run of 190 simultaneous connections.

Once the workload was debugged and tuned, we transferred the Apache and SPECweb99 directory trees onto disk image files where they could be mounted from within an M5 simulation. The workload then came up under M5 on the simulated network without modification.

IV. Preliminary Results

We performed preliminary simulations of the workloads on M5, configuring the server and client systems to approximate the characteristics of our testbed's 600 MHz Alpha 21164 platform, including an 8 KB direct-mapped L1 data cache, an on-chip 96KB 3-way associative L2 cache, and an 8MB off-chip L3 cache (known as the Dcache, Scache, and Bcache in Alpha terminology). The network simulated is similar to that of a modern gigabit Ethernet connection between the simulated systems.

For each workload, we ran four experiments in which we varied the extent to which DMA traffic was modeled on the bus. In the base case, both disk and network DMA transfers were hidden from the timing simulation, neither occupying bus bandwidth nor invalidating cache state. We then selectively modeled either disk and/or network DMA traffic. For DMA traffic that was not modeled explicitly, we use an approximate DMA latency based on the memory latency, transfer size, and bus bandwidth.

Since accurate DMA modeling changes cache miss patterns and I/O delays, modifying the execution path, we needed a method to accurately compare corresponding portions of the various runs. We defined a dummy instructions (using a reserved opcodes) that signal a particular point in the workload to the simulator. We then inserted these instructions into each of the workloads at various points in their execution, and collected statistics for the intervals between these marker instructions. Thus while the number of simulated cycles is different between runs for a given interval, the amount of work accomplished is the same.

For the netperf simulations, we took performance snapshots of the Bcache during four distinct intervals: the initial boot phase, two separate full 10 second netperf runs, and from the end of the second run to system shutdown. Figure 3 plots the number of Bcache misses observed in each interval for the various models. The Run 1 and Run 2 data points show a large disparity between the simulations that do and those that do not include network device DMA. Netperf has a very small memory footprint, but streams large amount of data through the memory system to the network interface. As DMA transfers move the received data into memory, they invalidate blocks in the cache hierarchy, resulting in a higher miss rate. Run 2 shows a higher number of Bcache misses than Run 1 because the Dcache is warmed up with netperf's instructions and data structures. As a result, there are fewer Dcache misses and the benchmark can achieve a higher bandwidth, causing more data to be streamed through the memory hierarchy.

For the SPECweb benchmark, we ran each simulation for 30 billion cycles starting at system boot. The boot process occupies approximately the first 100 million cycles at which point the workloads are started automatically. Our first measurement interval includes this boot time plus the time to configure the network interface, mount filesystems, and load the web server. The second interval covers the initial SPECweb99 warm-up phase, which largely consists of the webserver waiting for the clients to start issuing requests. The final interval is a section of the first set of web transactions between clients and the server.

1. We downloaded this code from the SPEC web site, at http://www.spec.org/web99/results/api-src/IBM-20020819-API.zip.

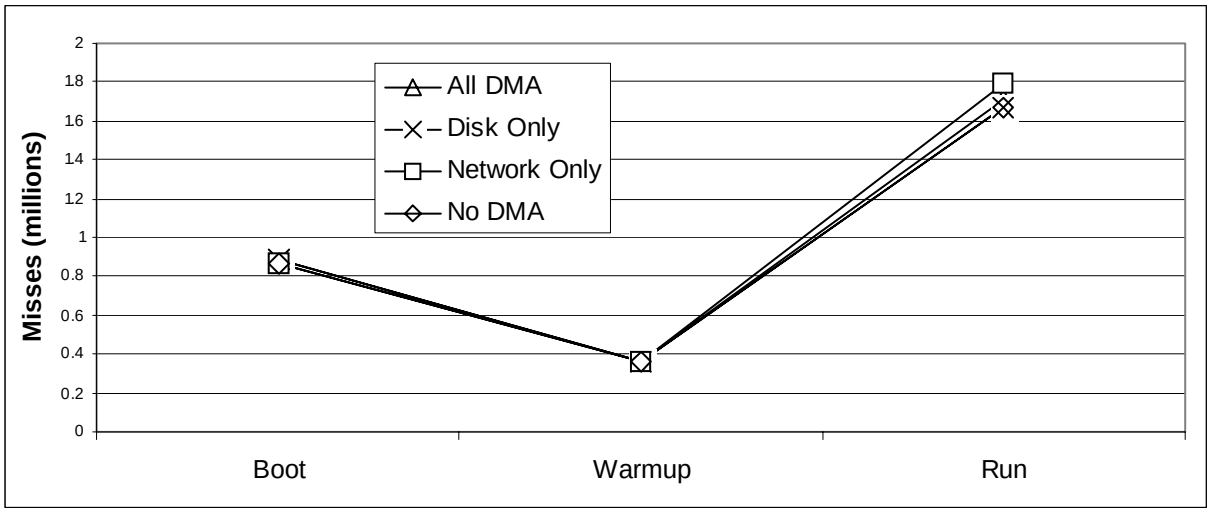

Fig. 4. Effect of DMA modeling on the number of Bcache misses for Apache running SPECweb99.

Figure 4 shows the total number of Bcache misses for the three intervals described above. DMA has a negligible impact for the first two intervals because very little DMA is occurring. This effect is similar to what is happening during the boot and halt intervals of the netperf benchmark.

The Run interval shows a small but measurable impact from modeling DMA. Network-only DMA modeling leads to the highest number of Bcache missed, followed by the full DMA case. The no-DMA and disk-only cases have about the same impact on Bcache misses. It seems counter-intuitive that the network-only number is higher than the all DMA number. We believe that modeling disk DMA causes the network DMA to become less frequent, in turn causing the workload to actually do less work during the interval. This causal relationship is similar how the Dcache miss rate affects the performance of the two netperf runs. Thus the all-DMA case achieves lower total network bandwidth than in network-only DMA, causing less work to get done during the Run interval.

V. Related Work

The original MIPS/IRIX SimOS [8, 17] is, to our knowledge, the only simulator other than M5 that supports all of the features necessary for studying network-oriented workloads: full-system simulation including OS code; a detailed, coherent memory system model; a detailed timing model of network DMA activity; and the ability to simulate multiple networked systems and their interconnect within a single simulator instance. However, we are unaware of any published studies which use SimOS to analyze the impact of network I/O on system performance.

Although the original SimOS and M5 share similar capabilities, SimOS places much heavier emphasis on simulation performance, including a high-speed emulation mode using binary rewriting. In contrast, M5 emphasizes modularity and modeling flexibility, with the hope of providing a more broadly reusable infrastructure. Thanks to several years of processor performance improvements since the development of SimOS, significant simulations are feasible under M5 without heroic levels of optimization.

Later derivatives of SimOS, including SimOS-Alpha [3], SimOS-PPC [10], and PHARMsim [5], do not include a timing model for the network interface. In one study using a SPECweb96 workload on SimOS-Alpha [16], researchers did not simulate the effects of network DMA, and loosely approximated network timing by running the client and server system simulations in separate processes on the same physical host so that they would progress at similar rates. Other full-system simulators, such as TFSim [12] (based on Simics [11]) and ML-RSIM [18], also neglect to model network I/O in detail.

Because of the lack of available tools, researchers investigating the architectural impact of networking workloads have used measurements of real systems [6, 14, 2] or simulations of small subsets of the workload (e.g., the TCP stack [13] or the dominant functions of the Apache web server [20]) run as user-level code. The former approach limits research to analysis of existing machines, without providing a path for evaluating proposed enhancements or future systems. The latter approach, while potentially providing interesting insights, cannot conclusively identify which portions of the system comprise the fundamental bottlenecks, or indicate the impact of proposed enhancements on overall system performance.

VI. Conclusions and Future Work

As we demonstrated in Section IV, network I/O has a measurable impact on system performance of network-oriented workloads. As these workloads grow in importance, simulators must provide detailed models of network I/O paths, along with full simulation of OS code, to properly analyze and evaluate future architectures. M5 provides a flexible and effective solution to meet this requirement.

In the immediate future we will further integrate existing M5 features into full-system mode, including support for multi-

threaded/multiprocessor systems and the detailed out-of-order CPU model. We will then extend our analysis by varying different parameters of the CPU (issue widths, frequency), cache hierarchy (sizes, bus bandwidth, associativities), and network (bandwidth, latency).

In the longer term, our goal is not only to understand the interaction of network I/O with current and future system architectures, but to propose and analyze architectural and/or OS enhancements which address the key bottlenecks. M5 will be a necessary tool in this research.

To spur further investigation of network-oriented workloads in the architecture community, we are committed to making a public release of M5 in the summer of 2003.

ACKNOWLEDGMENT

Many thanks to Steve Raasch and Dave Greene for their contributions to M5. This work was supported by the National Science Foundation under grants CCR-0105503 and CCR-0219640, by gifts from the Intel Corporation, and by a Sloan Research Fellowship.

REFERENCES

[1] Apache Software Foundation. HTTP server project. http://www.apache.org.

[2] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 90–101, January 1998.

[3] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[5] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti. Precise and accurate processor simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[6] Peter Druschel, Mark B. Abbott, Michael Pagels, and Larry L. Peterson. Network subsystem design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.

[7] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[8] Stephen A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior.* PhD thesis, Stanford University, February 1998.

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

[10] IBM Austin Research Lab. SimOS PowerPC web page. http://www.research.ibm.com/arl/projects/SimOSppc.html.

[11] Peter S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[12] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 108–116, 2002.

[13] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Cache behavior of network protocols. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 169–180, June 1997.

[14] Michael A. Pagels, Peter Druschel, and Larry L. Peterson. Cache and TLB effectiveness in processing network I/O. Technical Report 94-08, Department of Computer Science, University of Arizona, March 1994.

[15] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 318–329, May 2002.

[16] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 245–256, November 2000.

[17] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[18] Lambert Schaelicke and Mike Parker. ML-RSIM home page. http://www.cse.nd.edu/ mlrsim.

[19] Standard Performance Evaluation Corporation. SPECweb99 design document. http://www.spec.org/ web99/docs/whitepaper.html.

[20] Haiyong Xie, Laxmi Bhuyan, and Yeim-Kuan Chang. Benchmarking web server architectures: A simulation study on micro performance. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.