PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator

Matt T. Yourst

Department of Computer Science
State University of New York at Binghamton
yourst@cs.binghamton.edu

Abstract

In this paper, we introduce PTLsim, a cycle accurate full system x86-64 microprocessor simulator and virtual machine. PTLsim models a modern superscalar out of order x86-64 processor core at a configurable level of detail ranging from RTL-level models of all key pipeline structures, caches and devices up to full-speed native execution on the host CPU. Unlike other microarchitectural simulators, PTLsim targets the real commercially available x86 ISA, rather than a discontinued architecture with limited tools and an uncertain future. PTLsim supports several flavors: a single threaded userspace version and a full system version providing an SMT model and the infrastructure for multi-core support. We first describe what it takes to perform cycle accurate modeling of a complete x86 machine at the uop (micro-operation) level, along with the challenges and requirements for effective full system multi-processor capable simulation. We then describe the internal architecture of full system PTLsim and how it interacts with the Xen hypervisor and PTLsim's native mode co-simulation technology. We experimentally evaluate PTLsim's real world accuracy by configuring it like an AMD Athlon 64 machine before running a demanding full system client-server networked benchmark inside PTLsim. We compare the statistics generated by our model with the actual numbers from the real processor to demonstrate PTLsim is accurate to within 5% across all major parameters. We provide a discussion of prior simulation tools, along with their strengths and weaknesses. We describe why PTLsim's x86 focus is highly relevant, and we use our full system simulation results to demonstrate the pitfalls of userspace only simulation. Finally, we conclude by detailing future work.

Keywords: *simulation, x86-64, full system, SMP, SMT*

1. Introduction

Microprocessor simulators have been around since the dawn of computing to serve many roles, including hardware development and debugging, providing binary compatibility and enabling performance studies. Microarchitects and researchers are generally interested in *cycle accurate* simulation tools, which construct a complete microprocessor pipeline in software and simulate the flow of each instruction through this pipeline, so as to collect much more accurate timing information down to individual cycles. *Full system* simulators take accuracy a step further by simulating *all* instructions in both user applications and the kernel, and may also support cycle accurate modeling of multiprocessor configurations and hardware devices inside a virtual machine.

In this paper, we introduce *PTLsim*, a cycle accurate full system x86-64 microprocessor simulator and virtual machine. PTLsim models a modern superscalar out of order x86-64 processor core at a configurable level of detail ranging from RTL-level models of all key pipeline structures, caches and devices up to full-speed native execution on the host CPU. It provides a highly detailed and configurable model of a microarchitecture similar to the Intel Pentium 4, AMD K8 or Intel Core 2, including all pipelines, caches and devices. When PTLsim was first released as open source software in 2005, it supported 32-bit and 64-bit x86 Linux applications in userspace only.

In addition to being the only open source cycle accurate x86 microarchitectural simulator available to researchers, PTLsim supports a number of unique features, including *co-simulation*. PTLsim allows any virtual machine to be seamlessly switched between the host machine's physical CPUs (called *native mode*) and PTLsim's cycle accurate processor core models, all while maintaining strict timing continuity. This unique co-simulation technology enables rapid profiling of only the most relevant code segments, and makes the simulator self-debugging, since it can be continuously validated against a commercial x86 processor. PTLsim supports native mode co-simulation in both the classic userspace-only version and the new full system version presented in this paper.

In this paper, we describe how we have extended PTLsim into a full system simulator, called *PTLsim/X*, by integrating it with the well known Xen hypervisor (virtual machine monitor) technology, and how we have added multiprocessor and multi-threading support to PTLsim. Our approach of integrating PTLsim with a hypervisor like Xen has proved to be a very effective and elegant solution, since it allows us to apply PTLsim's native mode co-simulation technology to an entire virtual machine, with automatic sup-

port for multiple processors and transparent cycle accurate control of all timing parameters.

In Section 2, we first describe what it takes to perform cycle accurate modeling of a complete x86 machine at the uop (micro-operation) level, along with the challenges and requirements for effective full system multi-processor capable simulation. We also disclose the PTLsim out of order core's microarchitecture and features, and we describe how native mode co-simulation integrates with PTLsim's core models. Section 3 describes the internal architecture of full system PTLsim and how it interacts with the Xen hypervisor. We first introduce the features of Xen and why it is so attractive as the foundation of a simulation tool. We then describe how we built PTLsim on this foundation, and how various system-level aspects of the x86 architecture are modeled. The challenges of accurately modeling the flow of time in a simulation environment are also discussed.

We experimentally evaluate PTLsim's real world accuracy in Section 5 by configuring its out of order core and memory hierarchy as close as possible to a real AMD Athlon 64 machine. We then ran a demanding full system client-server networked benchmark (rsync with ssh and networking) inside PTLsim, and compared our results to the Athlon 64 hardware performance counters across numerous metrics. We use these data to prove that PTLsim can be used to very accurately model commercial microprocessors. In Section 6, we discuss a number of well known simulation tools, along with their strengths and weaknesses. We describe why PTLsim offers a distinct advantage through its focus on x86 (the most widely used ISA in the world) and we use our data from Section 5 to back up our assertion that userspace only simulation cannot deliver accurate results for many important workloads, particularly on multithreaded benchmarks. Finally, we conclude by detailing what the future holds for PTLsim. For space reasons we are not able to describe every PTLsim feature in this paper; instead the reader is referred to the PTLsim User's Guide and Reference [1].

2. PTLsim Architecture and Features

2.1. Full System x86 Simulator Requirements

Designing a cycle accurate model of a modern x86-64 microarchitecture is fundamentally different than building a simulator for a RISC architecture. The x86 instruction set is based on the two-operand CISC concept of load-and-compute and load-compute-store. However, all modern x86 processors (including PTLsim) do not directly execute complex x86 instructions. Instead, these processors translate each x86 instruction into a series of micro-operations (*uops*) similar to classical load-store RISC instructions. The internal uop instruction set used by PTLsim has many key differences from RISC instructions, since it is intended to efficiently support the nuances of x86 instructions while still being implementable in hardware. The uops used by PTLsim are intended to be quite similar to the functionality and encoding of uops in the Intel Pentium 4, Core 2 and AMD K8 processors.

The *PTLsim User's Guide and Reference* [1] describes many of the general hardware requirements for efficient execution of x86 code, which we summarize here. First, PTLsim provides a full x86-64 to uop decoder, and includes microcode for more complex instructions. All x86 instructions are *atomic*: either all uops comprising a single x86 instruction complete without exceptions and update the architectural state, or all uops are discarded. Many x86 instructions also update a subset of the processor condition code flags, and the ALUs must handle per-instruction operand size specifications. The x86 architecture stipulates that the processor itself must transparently handle unaligned memory accesses; PTLsim therefore provides efficient unaligned load and store uops. x87 floating point is still used by a great deal of legacy software instead of the newer SSE floating point instruction set introduced with the Pentium 4. Therefore, PTLsim provides full support for x87, albeit with reduced performance compared to SSE. Rather than decoding each x86 instruction every time it enters the simulated pipeline, PTLsim uses a *basic block cache* to store the uops generated from entire basic blocks (instruction sequences terminated by a branch). The basic block cache does *not* affect the architecturally visible performance of the processor model; it simply exists to speed up the simulation.

Full system simulation entails far greater challenges than userspace only simulation. For instance, the instruction decoder and basic block cache must be indexed by much more than just the RIP (x86 instruction pointer) of an instruction. Instead, x86 code is identified by its virtual address, the physical machine page frame number (MFN) on which it starts, optionally the MFN on which it ends (since x86 instructions are variable length and may cross pages), and a variety of contextual information (e.g. kernel or user mode, 32-bit or 64-bit x86-64 mode, status of flags, segmentation assumptions, etc). Self modifying code (SMC) must also be handled correctly: if an instruction overwrites the bytes in memory of another instruction already in the pipeline, the processor must flush the pipeline and re-execute the overwritten instruction. Similarly, even code not currently in the pipeline may need to be invalidated if it is still cached as decoded uops in the basic block cache.

PTLsim must also differentiate between virtual and physical addresses by modeling translation lookaside buffers (TLBs). All loads and stores must be checked for numerous possible exceptions, such as page faults, unaligned accesses that cross pages, non-cacheable pages, memory mapped devices and so forth. If a virtual address is not present in the TLB or is invalid, PTLsim must use a hardware state machine to "walk" the tree of x86 page tables until a virtual address is produced. Whenever pages are accessed or written for the first time, microcode must traverse the page table tree and set tracking bits in each page table entry (PTE); x86 operating systems expect the hardware (or microcode) to transparently perform all of these updates. All memory forwarding and alias checking (i.e. in an out of order processor) must be done with physical addresses; this becomes significantly more complicated when multiple threads are running on a single core using shared caches. Finally, when running 32-bit code, the segmenta-

tion features (segment base, limit and access rights) of the x86 architecture must be enforced.

Exceptions and interrupts must be properly and precisely delivered to the running code. PTLsim uses its microcode to build stack frames, access interrupt descriptor tables (IDT), switch to kernel mode and redirect the processor to the exception handler entry point. Asynchronous interrupts from devices and timers must also be delivered at the proper time to the proper handler. PTLsim must provide virtual hardware models of the real time clock, interrupt controller and other devices, and these models all must be cycle accurate and fully deterministic for debugging purposes. External devices (like disks and network interfaces) outside the virtual machine may also need to deliver interrupts; PTLsim must ensure proper delivery and timing of these events. Lazy context switching of floating point registers must also be detected and handled.

PTLsim also supports modeling of multi-processor or simultaneous multithreading (SMT) machines. Interlocked instructions (e.g. the x86 LOCK prefix, the atomic bit test and set instructions, the xchg and xadd instructions, and various memory fences) must be properly handled for a modern multi-threaded kernel and applications to work correctly. This involves significant complexity, since multiple CPUs or hardware threads may attempt an interlocked access at the same time, and the simulator must arbitrate access without deadlocks. If a multi-core or symmetric multiprocessor (SMP) configuration is used, the simulator must model cache coherence transactions and the movement of cache lines between cores.

These are just a few of the challenges PTLsim must address to be an effective cycle accurate full system simulator - the amount of complexity is significantly higher than the typical RISC architectures modeled by earlier full system simulation tools.

2.2. PTLsim Core Model

PTLsim includes a variety of core models, including the main superscalar out of order core, an SMT (simultaneously multi-threaded) version of that core, and an in-order sequential core used for rapid testing and microcode debugging. Models can be added as plug-ins by simply registering a C++ class with PTLsim and recompiling. To support multiprocessor machines with many VCPUs (virtual CPUs assigned to a domain, in contrast with the physical processors managed by Xen), multiple core instances can operate in parallel; the simulator control logic automatically advances each core by one cycle in round robin order and provides memory synchronization facilities shared by all cores.

The default core model is a modern superscalar out of order design, based on a combination of features from the Intel Pentium 4, AMD K8 and Intel Core 2, but also incorporates some ideas from IBM Power4/Power5 and Alpha EV8. The following is a summary of the characteristics of this processor model:

The simulator directly fetches pre-decoded microoperations from the basic block cache (described previously) but can simulate cache accesses as if x86 instructions were being decoded on fetch. The clustered microarchitecture is highly configurable, allowing multi-cycle latencies between clusters and multiple issue queues within the same logical cluster. Functional units, the mapping of functional units to clusters, issue ports and issue queues and uop latencies are all configurable. The number and type of physical register files can also be configured. Issue queues are based on a collapsing design and are modeled at the RTL level with broadcast based matching to wake up uops. Replay of loads and stores is supported for handling store to load forwarding and store to store merging dependencies, as well as arbitrary types of speculation. The commit unit supports x86 specific semantics, including atomic commits and exception handling and recovery. The data cache hierarchy by default consists of an L1 D-cache, L1 I-cache, unified L2 cache and unified L3 cache, along with a DTLB and ITLB. The sizes, associativity, latency, bandwidth and numerous other parameters can be customized. The movement of cache lines through miss buffers and buses is based on a very detailed RTL-level model of key structures. The processor provides a hardware TLB walk engine. Branch prediction is also fully configurable; PTLsim currently includes various models including a hybrid gshare based predictor, bimodal predictors, saturating counters, etc. Simultaneous multi-threading (SMT) model allows up to 16 threads per core, with separate per-thread fetch queue, ROB, LDQ, STQ, and other structures, but shared caches, issue queues and functional units. Thread synchronization hardware (for interlocked instructions) is fully modeled. The SMT core provides configurable thread priority policies and deadlock prevention schemes. The microarchitectural model is described in more detail in the *PTLsim User's Guide and Reference* [1].

2.3. Native Mode Co-Simulation

PTLsim includes a very powerful feature absent from other cycle accurate simulators: it can dynamically switch the target virtual machine (or in userspace PTLsim, the target process) between the host system's real physical CPUs and one of PTLsim's simulated core models (such as the out of order core, the SMT core or the sequential core). *Native* mode means the domain is executing at full speed on the host's physical x86 processors, even though PTLsim is still waiting in the background. The ability to selectively run parts of the program at full speed presents several unique advantages compared to previous simulation tools.

First, it is no longer necessary to emulate billions of instructions during the initialization phase of a benchmark before starting cycle accurate simulation. With PTLsim, the user can set a *trigger* point at the top of the benchmark's main loop (or any other key point where cycle accurate simulation should begin), then switch back to native mode and execute at full speed until the trigger point is hit. PTLsim allows trigger points to be specified by RIP (the x86 instruction pointer), special `ptlcall` x86 instructions within the benchmark itself that trigger when executed (as described below), instruction counts, cycle counts and a variety of other specifiers. In addition, while in simula-

tion mode, PTLsim's snapshot facility allows the state of all internal event counters to be checkpointed at any time. By subtracting these snapshots using the included PTLstats analysis tools, the "warmup" periods traditionally used to exclude transient cache and branch predictor effects can be greatly reduced.

Second, native mode effectively makes PTLsim self debugging. It is possible, on an instruction by instruction basis, to determine where the architectural state produced by PTLsim's model begins to diverge from the state produced by the native x86 host processor. This is done by repeatedly switching from simulation mode back to native mode at different points (between specific instructions) and observing if the resultant behavior is correct. Using binary search techniques, the problem can be rapidly isolated and debugged. Full system PTLsim/X includes features to completely isolate the virtual machine from non-deterministic outside events, thereby allowing this comparison feature to work effectively.

Third, statistical sampled simulation can be used very effectively: for instance, using PTLsim's scripting language, the user can request that the out of order simulation core be used for 100 million instruction spans out of every billion real instructions, with the majority of the time spent in native mode. PTLsim's statistical snapshot facility can be used to present a continuous picture of the benchmark's performance characteristics; Section 5 shows this feature in action.

3. Full System PTLsim/X and Xen

Xen [14] is an open source x86 virtual machine monitor, also known as a *hypervisor*. Each virtual machine is called a "domain", where domain 0 is privileged, runs Linux, and accesses all hardware devices using unmodified drivers; it can also create and directly manipulate other domains. Guest domains typically use Xen-specific virtual device drivers, and make *hypercalls* into the hypervisor to request certain services that cannot be easily or quickly virtualized. Each guest can have up to 32 VCPUs (virtual CPUs). Xen has essentially zero overhead [15] due to its unique and well planned design; it's possible to run a normal workstation or server under Xen with full native performance.

Under Xen's "paravirtualized" mode, the guest OS runs on an architecture nearly identical to x86 or x86-64, but a few small changes critical to preserving native performance levels. First, the domain can read its own page tables, but must ask Xen to perform any updates (this ensures isolation). Xen assigns arbitrary non-contiguous physical memory physical memory pages (called "machine frame numbers", or MFNs) to the domain, rather than a linear span of pages starting at physical address zero. Interrupts are delivered using an *event channel* mechanism, which is functionally similar to the IO-APIC hardware on the bare CPU (essentially it's a "Xen APIC" instead of the Intel and AMD models already supported by the guest kernel). Events can be mapped to virtual interrupts, physical device interrupts or inter-domain ports. Finally, Xen provides the guest with additional timers, so it can be aware of both "wall clock" time as well as execution time (since there may be gaps in the latter as other domains use the CPU).

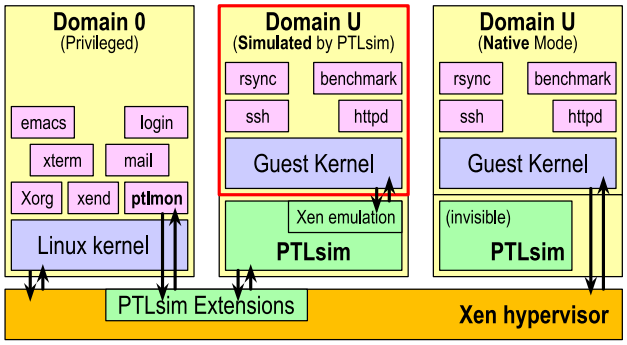
All other features of the paravirtualized architecture perfectly match x86. This makes it possible to run a normal Linux distribution, with totally unmodified drivers and software except for a small part of the kernel and boot code, at nearly full native speed. All major Linux distributions now fully support Xen, along with Solaris and FreeBSD.

Xen also supports "HVM" (hardware virtual machine) mode, which provides nearly perfect emulation of the x86 architecture and some standard peripherals. The advantage is that an "uncooperative" guest OS (namely, Windows) can be run inside the virtual machine on newer x86 processors supporting hardware virtualization extensions (Intel VT and AMD SVM), however this mode offers lower performance than para-virtualization since more emulation (particularly for legacy devices) is required.

Xen is very attractive as the foundation of a full system version of PTLsim for several reasons. First, its simplified view of the x86 architecture (while still preserving binary compatibility) makes constructing a full system simulator significantly easier than having to deal with legacy x86 artifacts like 16-bit boot code as well as the need for device models for chipsets, PCI buses, DRAM controllers and numerous other details handled internally by Xen itself. More importantly, the Xen timing model allows the simulator to completely virtualize the passage of real time, even while running in native mode. This enables the user to easily experiment with parameters normally fixed by the x86 architecture, such as the simulated processor core frequency, timer interrupt frequency, artificial delays in interrupt and DMA delivery, and so forth. Xen is also integral to being able to seamlessly switch between native mode and simulation mode. It is always possible to implement a purely userspace full system simulator with no kernel or hypervisor support, but performance and flexibility would suffer greatly without the availability of native mode. Finally, the majority of the PTLsim code can be isolated inside the target domain (where it can be easily debugged), with only a small number of hooks present in the modified Xen hypervisor itself or in the privileged portion of PTLsim running in domain 0.

The use of Xen does have some disadvantages. First, setting up a dedicated machine to run PTLsim/X is often required: the PTLsim-enhanced Xen hypervisor itself must run underneath all other software, a Xen-enabled kernel must be run on the machine, and the Linux distribution used in domain 0 must support the Xen virtual device drivers and domain management tools. Fortunately, all major Linux distributions now support Xen, as do Solaris and FreeBSD. PTLsim's dependence on Xen's paravirtualized drivers, rather than simulating true bare-metal x86, means that it cannot currently run "uncooperative" operating systems like Windows, since these systems expect to find real x86-standard hardware rather than Xen's virtual devices. Fortunately, HVM support will be appearing in PTLsim later this year.

Figure 1. PTLsim and Xen: Interactions between PTLsim, the Xen hypervisor, and the PTLsim monitor



4. Running PTLsim/X on Xen

The basic architecture of full system PTLsim/X, and its interactions with Xen, are shown in Figure 1.

When a domain is first created but before it begins executing, PTLsim is instantiated in the domain using the PTLsim monitor program (PTLmon), which is responsible for booting PTLsim inside the target domain and coordinating its communication with the outside world. PTLmon is a normal Linux program that runs in domain 0 with root privileges and a special connection to the enhanced Xen hypervisor. PTLmon increases the domain's memory reservation so as to reserve a range of physical pages for PTLsim (by default, 32 MB of physical memory), loads the real PTLsim core code into these pages and sets up initial page tables mapping the PTLsim image. Finally, PTLmon uses a special `contextswap` hypercall added to the Xen hypervisor. This hypercall atomically exchanges the state (registers, page tables and so on) of all VCPUs in the target domain with PTLsim's initial bootup state. The guest domain's x86 state for each VCPU is saved inside PTLsim for when simulation actually begins.

At this point, PTLsim is executing in virtualized kernel mode on the bare x86 hardware as its own mini operating system, but is still isolated by Xen, since it can only map its own pages and all pages allocated to the guest. This isolation makes PTLsim significantly easier to debug, since very little code runs inside the hypervisor itself; the majority of PTLsim's code is locked up within the target domain. PTLsim initializes and establishes a shared memory region and associated virtual interrupts used to communicate with the PTLsim monitor process still running in domain 0. In effect, PTLsim can still use a subset of the regular Linux system calls, but these requests are forwarded back to domain 0, where PTLmon acts as the core simulator's proxy to write output to the console, open log files, and so forth. This is the only means by which PTLsim communicates with the outside world. PTLsim maps into its virtual address space all physical memory belonging to the target domain. All operations in PTLsim are performed on physical addresses, since the PTLsim out of order core only tracks physical addresses in the pipeline after TLB lookups have completed.

Whenever the guest kernel executes privileged instructions or makes Xen hypercalls under simulation, PTLsim's x86 microcode routines map these operations to either real Xen hypercalls or simple updates to PTLsim's simulated state. For instance, when the kernel attempts to reload the CR3 control register (specifying the page table base), either through a `mov cr3,xxx` instruction or via an equivalent `MMUEXT_NEW_BASEPTR` hypercall, PTLsim simply updates its internal simulated copy of CR3 in the active VCPU's context structure and flushes all simulated TLBs. Other operations, such as sending virtual interrupts to outside devices like network adapters or hard disks, are simply passed straight through PTLsim and on to the hypervisor for processing.

4.1. Native Mode in PTLsim/X

PTLsim's unique native mode switching feature requires a fairly sophisticated infrastructure at the hypervisor level so as to guarantee the guest operating system is unable to detect the transition even if it occurs between arbitrary x86 instructions. It must properly hide gaps in execution time down to the nanosecond, and it must support the atomic switching of many VCPUs (in an SMP domain) back into and out of PTLsim. PTLsim always boots into simulation mode to perform initialization tasks, but immediately switches back to native mode to start the guest kernel's boot process.

PTLsim defines the special `ptlcall` x86 opcode `0x0f37` as a breakout opcode. It is undefined in the normal x86 instruction set, but when executed by any domain running under the PTLsim-enhanced Xen hypervisor in native mode, it causes the entire domain to be de-scheduled from the host machine's physical CPUs and execution seamlessly resumes on PTLsim's simulated CPU. In native mode, Xen intercepts the invalid opcode fault, freezes the domain and notifies the PTLsim monitor process, via a virtual interrupt, that it should switch the domain back to simulation mode.

The `ptlcall` instruction may specify one of several operations via registers, and may atomically enqueue a command list for PTLsim to process. For instance, a command list (specified as a text string) may consist of "`-core smt -run -stopinsns 10m : -native`". This command tells PTLsim to switch back to simulation mode, execute 10 million x86 instructions under PTLsim's SMT core, then switch back to native mode. PTLsim comes with a special program, called `ptlctl` (PTLsim control) that is installed and executed *within* the target domain. This program is simply a wrapper around the `ptlcall` instruction, and allows the user (or a shell script running within the domain) to interactively submit command lists to PTLsim.

Because PTLsim's cycle accurate simulation mode is much slower than native execution, wall clock time (and the physical processor's timestamp counter, accessible through the `rdtsc` instruction) will have advanced far past its last simulated value when switching back to native mode. Therefore, PTLsim uses another feature added to Xen to virtualize the timestamp counter and other clocks while in native mode. Xen will transparently emulate `rdtsc` and any

realtime values it passes to guests at timer interrupts, and will subtract a delta such that the transition back to native mode is seamless, ideally without missing a single nanosecond.

4.2. The Nature of Time

Full system simulation poses some difficult philosophical questions about the nature of time itself and the phenomenon of "time dilation". Specifically, if a simulator runs X times slower than the native CPU, both external interrupts and timer interrupts should theoretically be generated X times slower than in the real world. PTLsim always internally generates timer interrupts for the domain at the correct rate, but time dilation of other events is critical for obtaining accurate simulation results on I/O intensive workloads: for events like network traffic, if a real network device fed interrupts into the domain in realtime, and the simulator injected these interrupts into the simulation at the same rate, they would appear to arrive thousands of times faster than any physical network interface could deliver them. This can easily result in a livelock situation not possible in a real machine; at the very least it will deliver misleading performance results.

On the other hand, interacting with a domain running at the "correct" rate according to its own simulated clock can be unpleasant for users. For instance, if the "`sleep 1`" command is run in a Linux domain under PTLsim, instead of sleeping for 1 second of wall clock time (as perceived by the user), the domain will wait until 1 billion cycles have been fully simulated (assuming the simulated processor frequency is 1 GHz). This is because PTLsim keys interrupt delivery and all timers to the simulated cycle number in which the interrupt should arrive (based on the user-specified core clock frequency). In addition to being annoying, this behavior will massively confuse network applications that rely on precise timing information: a TCP/IP endpoint outside the domain will not expect packets to arrive thousands of times slower than its own realtime clock expects, resulting in retransmissions and timeouts that would never occur if both endpoints were inside the same "time dilated" domain.

Interrupt and DMA trace recording and injection is a popular method of precisely synchronizing the apparent arrival times of interrupts and external DMA transactions with the simulated core's cycle counter. In this scheme, a checkpoint of the target machine's physical memory and register state is captured and saved to a file (this is done under the control of a hypervisor such as Xen). The hardware device drivers (or the PCI bus of a real machine, if a logic analyzer is used) are then instrumented to attach a timestamp (i.e. core cycle counter) to every incoming interrupt and DMA write. These event records (comprising a timestamp, interrupt type, any memory overwritten by the DMA transaction and any relevant device register states the simulated CPU may attempt to read) are written to a trace file. The simulator then starts execution at the checkpoint, and reads the interrupt and DMA trace file as if it were a queue: the event at the head of the queue is injected into the simulated processor if and when the simulation reaches the cycle number the event was timestamped with. This technique is used Intel's own internal Pentium 4 based simulation toolsuite [12], and is preferred by commercial microprocessor designers since it guarantees deterministic and infinitely repeatable simulation of real external bus transactions.

PTLsim will soon be adding this capability based on Xen's built-in checkpoint infrastructure and virtual device driver architecture. Xen virtual virtual device drivers are divided into two separate modules: the *backend* driver runs in domain 0 as a Linux kernel module, and it may communicate with a real physical device, while the frontend driver runs in each user domain's kernel and only communicates with the backend driver using a standardized interface. After checkpointing the target domain, the backend drivers for network and disk I/O can be instrumented to record all future events (interrupts) sent to the target domain, as well as any DMA operations into specific pages destined for the target domain. The target domain is then restarted from the checkpoint, and event tracing continues for some finite period while the domain runs at full speed in native mode. Finally, the domain is again restarted from the checkpoint, but this time PTLsim runs in simulation mode, where it reads the event trace file to directly inject events and DMAs into the domain under simulation according to the time dilated simulated cycle counter, rather than in real time.

4.3. Physical Address Translation and TLBs

In full system PTLsim, actual physical addresses are used for tracking and executing all cache and memory operations; virtual addresses are used only up to the point at which each load or store issues. PTLsim uses a simulated TLB (translation lookaside buffer) to map virtual addresses to physical pages, as described in Section 2.1. The delay involved in servicing a TLB miss (on x86 at least) is proportional to the time it takes to execute a chain of four dependent loads (one per level of the 4-level x86-64 page table tree). Modern x86 processors, including the one modeled by PTLsim, use a dedicated state machine to inject loads into the pipeline until the page table tree has been traversed. Each of these loads may hit or miss in the cache, and page tables will compete with user data for cache lines and load unit bandwidth. All of these effects are very complex to model unless the actual physical pages comprising the page table are available. Furthermore, the caches of real microprocessors are tagged by physical addresses, not virtual addresses as in userspace-only simulators. Therefore, two addresses that would not normally cause a conflict miss in the data cache based solely on their virtual addresses may actually conflict when mapped to physical pages with non-sequential physical addresses.

4.4. Multi-Threading and Multi-Processor Support

PTLsim was designed from the ground up to support multiple VCPUs per domain, thereby allowing efficient SMP, multi-core and multi-threaded core models. As described in Section 2.2, the active core model decides how

to distribute the VCPUs in a domain onto individual cores and threads within cores. The `Context` structure in PTLsim is central to multi-processor support. Each VCPU has one `Context` structure encapsulating all information about that VCPU, including its architectural registers, x86 machine state registers (MSRs), page tables and internal PTLsim state. As the simulator commits instructions from a given hardware thread or core, it updates the architectural state in that core's Context structure. Similarly, the `Context` structure is available to all microcode functions and other PTLsim subsystems.

VCPUs may choose to block by executing an appropriate operation (i.e. the x86 `hlt` instruction or an equivalent hypercall), suspending execution until an interrupt arrives. PTLsim cores can simulate this by checking the `running` field in a VCPU's context structure; if zero, the corresponding VCPU is blocked and no instructions should be processed until the `running` flag becomes set, such as when a virtual interrupt arrives. When virtual interrupts arrive, PTLsim's microcode will automatically build an interrupt stack frame and redirect simulated execution into the kernel, just as a real x86 processor does. If a domain has multiple VCPUs, PTLsim runs the simulation entirely on the first VCPU, while putting the other VCPUs into an idle loop at boot time; their sole purpose is to receive events for injection into the simulation.

The x86 architecture allows memory instructions to be modified by adding a special `LOCK` prefix to the opcode. The lock prefix instructs the current VCPU (or hardware thread) to block all other VCPUs from executing loads or stores on the memory touched by the instruction until the instruction completes. PTLsim implements these semantics by allowing each load uop in the interlocked instruction to acquire a lock on a given physical memory location by sending its physical address to an interlock controller shared by all SMT threads within a given core (in the SMT model), and/or by using a user-defined cache coherency mechanism to lock a cache line. If future load or store uops attempt to access this same physical address, they will be replayed in the out of order core until they can acquire the lock themselves. Locks are released when the x86 instruction containing the locked load commits.

It should be noted that PTLsim's multi-core abilities are currently limited in that each core has a dedicated cache hierarchy (L1/L2/L3); the cycle accurate interconnect network and cache coherence logic are up to each user to provide. By default, PTLsim models an "instant visibility" cache coherence model: there are no delays on line movements between cores. However, the infrastructure is in place for MOESI-compatible cache coherence models to be easily plugged into the PTLsim code.

5. Experimental Evaluation

To experimentally evaluate PTLsim's real world accuracy, we configured its out of order core and memory hierarchy as close as possible to a real 2.2 GHz AMD Athlon 64 (K8 microarchitecture) based machine, described in more detail below. We then selected a representative full system client-server benchmark (rsync with ssh and networking), and we ran two trials: native mode (on the real Athlon 64 reference machine), and full system PTLsim/X, configured like a K8 processor. For each benchmark and trial environment, several key statistics were analyzed, including total number of cycles, total number of x86 instructions, L1 cache miss rate, branch mispredict rate and DTLB miss rate.

Our choice of benchmark is somewhat nontraditional and ad hoc, since our goal is highlight the fidelity differences between native execution on real K8 silicon and PTLsim's full system model (including all kernel and user code, TLB misses, page faults, physical address caching effects and device overhead). We have intentionally omitted SPEC CPU results, since this benchmark suite is single threaded and has essentially no interaction with the operating system, making it inappropriate for our purposes. We have selected the well known *rsync* [18] program found on every Linux system. Rsync is a client server file transfer system used to maintain two large sets of files by finding and transferring only the differences between files, even within potentially large files.

Rsync is useful as a benchmark for several reasons. First, it makes intensive use of kernel level code for directory searching and I/O (to build file lists, read and write files and communicate with a receiver rsync process via pipes or TCP/IP). It also performs a CPU-intensive userspace computation to process file lists, execute the rsync delta algorithm [18] and compress data via the gzip algorithm. It is divided into several easily identified phases, such as building the server-side file list, building the client-side file list, isolating differences between files, and actually sending the compressed delta data. Finally, it has the ability to pipe rsync data over an additional transport layer, such as ssh, via interprocess pipes.

In our test configuration, we are using *ssh* (secure shell) as the network transport for rsync, since this is a typical usage. This means *rsync* starts the ssh client as a separate process, which connects (via public key authentication) to the sshd server process. All subsequent rsync traffic is compressed and then piped through the client ssh process, across the encrypted ssh tunnel via TCP/IP, into the server sshd process (where it is decrypted), and on to a server-side rsync process (hence four total processes are involved). To ensure deterministic timing, we ran both the client and server on the same virtual machine, all packets still traverse the Linux TCP/IP stack even though they are going through the local host only. Our file set consists of 6186 text files all under 300 Kbytes, for a total of 48 Mbytes. The files are divided into two roughly equal groups; the test consists of running rsync to synchronize the second group with the first group.

To create the target domain, we installed unmodified SuSE Linux 10.1 (64-bit edition) onto an ext2 format disk image, and we allocated 384 MB of physical memory to the virtual machine. The disk image was loaded into RAM and mounted read/write inside the target domain, and external interrupts were emulated using the `-maskints` PTL-

sim option; this guarantees deterministic cycle level timing. We replaced the normal Linux `init` program with a script that performs four tasks as soon as the system has booted: `sshd` (the ssh server process) is started and the local network interface is configured, "`ptlctl -run`" begins the simulation (or native profiling run), the `rsync` process is run, then "`ptlctl -kill`" is run after rsync exists, thereby terminating the domain and recording the statistics. The entire trial takes approximately 0.7 seconds of real time (i.e. ~1.5 billion cycles at 2.2 GHz) when run on the native CPU. The Xen disk image and settings used to conduct this test are available from [20]. Our test machine was an AMD Athlon 64 X2 4400+ (2.2 GHz) with 1 MB L2 cache per core. Using the usual Xen facilities, we intentionally bound the domain under test to the second core only: this ensures that our activities in domain 0 (bound to the first core only) would not disturb the cache contents or performance counters of the second core; in effect, the virtual machine was run in a hard realtime isolated environment.

For the native mode trial, we utilized the Athlon 64's built in performance monitoring counters. All AMD K8 processors allow the operating system (or in our case, the PTLsim/X hypervisor) to configure up to four event counters, each of which monitors one of roughly a hundred different microarchitectural events. The counters can be read out using the *rdpmc* (read performance monitoring counter) x86 instruction. We used the special `-perfctr` PTLsim option (e.g. `-perfctr L1D-miss-rate -force-native`) to force PTLsim to flush all CPU caches and activate the hardware event counters before immediately switching back to native mode. The counters are read out again when the benchmark has completed (i.e. when the "`ptlctl -kill`" command is executed by our simulation script). Since only four counters are provided by K8 CPUs, we ran each benchmark four times (to separately measure each of the statistics above); the number of cycles and number of x86 instructions was counted every time to ensure the exact same code was executed. These results are tabulated in Table 1.

For the simulation trial, we attempted to configure PTLsim as close as possible to AMD's K8 microarchitecture [16, 17]. We modified `smtcore.h` to declare a 72-entry ROB, 44-entry load/store queue, three 8-entry issue queues (for the K8 integer cluster's three "lanes") and a 36-entry issue queue in a separate cluster two cycles away (for floating point). Unlike most processors, K8 does not use any physical register files: instead, it uses a clever future file design and value capturing issue queue entries. Since PTLsim is based around physical register files, we had to estimate and made each register file 128 entries, so as to make the ROB the bottleneck. We disabled PTLsim's load hoisting (in which loads are speculatively issued before unresolved stores), since K8 does not perform load hoisting in this manner. We also enabled enforcement of cache banking, since the K8 data cache is pseudo dual ported: it is divided into 8 banks along 64-bit boundaries; each bank has one read/write port. Bank conflicts result in a 1-cycle replay of the colliding load or store; typically this happens for less than 2% of all accesses. We also reconfigured the uop latency and functional unit table to match the K8 microarchitecture, and we added a branch predictor equivalent to the K8's 16K gshare-like global history based predictor. Finally, we configured the processor with a 64 KB 2-way D-cache and I-cache, a 1 MB 16-way L2 cache and a 32-entry DTLB and ITLB. The L2 was placed 10 cycles away and the main memory was 112 cycles away; these latencies were experimentally measured on our reference system, and roughly match the numbers in AMD's specifications [16, 17] for a 2.2 GHz chip.

Table 1. Accuracy of PTLsim on multiple metrics compared to reference silicon (AMD Athlon 64 @ 2.2 GHz). Figures are in thousands.

Trial	Native K8	PTLsim	%Diff
Cycles	1,482,035K	1,545,810K	+4.30%
x86 Insns Committed	990,360K	1,005,795K	+1.55%
uops	1,097,012K	1,436,979K	+30.99%
L1 D-cache Misses	6,118K	6,564K	+7.28%
L1 D-cache Accesses	414,285K	418,072K	+0.91%
L1 Misses as %	1.48%	1.57%	+0.9%
Total Branches	138,062K	135,857K	-1.60%
Mispredicted Branches	5,727K	5,392K	-5.84%
Mispredicted %	4.15%	3.97%	-0.18%
DTLB Misses	1,593K	3,895K	144%
DTLB Miss Rate %	0.38%	0.93%	245%

Several other studies have also attempted to configure a simulator to match a specific commercial microprocessor. In [19], the authors start with the Alpha version of SimpleScalar and tune its parameters to closely match a real Alpha 21264. The authors of [19] used a set of microbenchmarks to evaluate their model's accuracy against the on-chip hardware performance counters; the study claims an average IPC difference of around 2% from the actual Alpha processor. In contrast, our study of PTLsim evaluates a complex real world client-server benchmark across both user and kernel code, on a much more complicated instruction set architecture (x86-64), and across many more statistics besides IPC (including cache performance, branch prediction, TLB efficiency and user vs kernel activity). These results are detailed in Table 1 and Figures 2 and 3.

To conduct the actual simulation, we booted the target virtual machine under PTLsim. This created a statistics snapshot every 2.2 million cycles; at 2.2 GHz, this is 1000 snapshots per second. The run took approximately 62 minutes on our reference hardware (the 2.2 GHz Athlon 64). We simulated 1.55 billion simulation cycles, for a simulation throughput of 415540 cycles per second.

Table 1 gives a comprehensive comparison between a real AMD K8 microprocessor and the PTLsim model configured in accordance with the K8 microarchitecture. The native comparison results were obtained using the Athlon 64's built-in performance counter registers, whereas the PTLsim numbers were taken from the PTLstats output.

From these data, it is clear that given an appropriate configuration, PTLsim is able to accurately model the perfor-

mance of real commercial microprocessors with very high fidelity in almost all key statistics. First, the major architecturally visible statistics (x86 instructions committed, total branches, total loads and stores) are all within 2% of each other, indicating that both PTLsim and the K8 core are executing almost exactly the same span of code. PTLsim and its microcode do execute many more uops than K8, but this is to be expected: the K8 microarchitecture always operates on "uop triads" (groups of three operations), rather than individual uops like PTLsim counts. Both the K8 and PTLsim have similar L1 cache miss rates, but it is likely the K8's more sophisticated prefetch unit is able to lower the miss rate more effectively. The branch misprediction rate is also very similar; even though the PTLsim predictor was configured in accordance with published documentation on the K8 microarchitecture [16, 17], the actual hardware has slightly lower predictor accuracy than our model. The DTLB miss rate is higher for PTLsim than for the real hardware, in large part because K8 includes a very sophisticated two level TLB (with 32 L1 entries and 1024 L2 entries in a 4-way array), along with 24-entry page directory entry (PDE) cache and other optimizations for accelerating page table walks; in contrast, PTLsim only includes the 32-entry L1 TLB. The rest of the performance difference, roughly 5% on average, can be attributed to the fact that the microarchitectures of PTLsim and K8 are obviously still different in many small ways, despite our attempts to match the configurations as closely as possible.

It's important to remember that the native execution numbers in Table 1 were obtained by real world measurements. Therefore, even though we took every precaution to ensure isolation, a small amount of non-determinism still exists because of occasional hardware interrupts and DRAM bandwidth contention on the test system. Furthermore, since the K8 only has 4 performance counters (two of which were kept constant to count cycles and x86 instructions), the same benchmark had to be run multiple times with different counter configurations so as to obtain all the required data. Fortunately, the average variance between runs (in terms of total cycle count and x86 instruction count) was less less than 1% in every case.

Figure 2 is a time lapse plot (automatically generated by PTLstats) of the percentage of all cycles spent in kernel mode, user mode, and idle, obtained from PTLsim's `external.cycles_in_mode` statistics node. The various phases of the *rsync* benchmark can be clearly seen in this diagram. This diagram clearly illustrates how full system cycle accurate simulation is critical to obtaining accurate performance numbers on real world code: a substantial amount of time is spent within the kernel (in this case, 15% of all cycles), or idle while waiting for I/O (27%). In a userspace only simulator (or if PTLsim's full system mode is disabled), this time would not be accounted for - resulting in erroneously fast performance results compared to a real machine. Even if a given benchmark does little to no I/O, userspace only simulation still cannot model the effects of TLB misses, the interaction of physical addresses with the cache subsystem and similar low level but important details.

Figure 3 shows a time lapse plot (also automatically generated by PTLstats) of several key microarchitectural statistics, sampled every 2.2 million cycles. In particular, the branch misprediction rate, L1 miss rate and DTLB miss rate are readily visible in this diagram as the benchmark moves through various stages. These data can be very useful for optimizing programs at a level not generally possible when limited by the constraints of the hardware performance counters in commercial silicon microprocessors.

6. Related Work

Simulation tools can be divided into roughly two categories. *Functional* simulators precisely emulate each instruction in the target instruction set, but do not provide any cycle accurate timing information; they are simply interpreters. *Cycle Accurate* simulators, on the other hand, construct a complete microprocessor pipeline in software and simulate the flow of each instruction through this pipeline, so as to collect much more accurate timing information down to individual cycles. Even more detailed than cycle accurate simulators are the ad-hoc simulators used by commercial microprocessor designers, including RTL models and circuit level models; these are not covered in this paper.

For the x86 platform, numerous functional simulators have been developed. *Bochs* [3] is a well known open source x86 simulator, with support for nearly all x86 features. However, Bochs is very slow (around 5-10 MHz equivalent) and is not useful for implementing cycle accurate models of modern uop-based out of order x86 processors (i.e. it does not model caches, branch prediction, etc). *QEMU* [4] supports multiple CPU host and guest architectures (PowerPC, SPARC, ARM, etc) and uses dynamic compilation to achieve significantly faster simulation speeds than pure interpretation. *Simics* [5] is a commercial functional simulation suite for various processors (including x86) as well as user-designed plug-in models of real hardware devices. Like QEMU, Simics uses x86-to-x86 binary translation to achieve good performance. However, Simics does not include cycle accurate simulation features below the x86 instruction level. All of these tools share one common disadvantage: they are unable to model execution at a level below the granularity of x86 instructions, making them unsuitable to microarchitects interested in detailed cycle-by-cycle performance data.

6.1. Integrated, Trace Based and Split Phase Simulators

Integrated simulators, such as PTLsim and a few of the tools described below, simultaneously model both the functional aspects (i.e. to compute the correct result of each instruction) and cycle accurate timing and scheduling structures in a single simulator. This approach has the distinct advantage of guaranteeing both 100% correct operation and very reliable timing data: any functional bugs in the simulator are immediately apparent (since the system under simulation will crash or behave incorrectly), and most timing bugs (such as incorrect bypassing, issuing an instruction

Figure 2. Time lapse graph of cycles spent in each CPU mode (user, kernel, idle). The X-axis indicates the snapshot ID; snapshots were taken every 2.2 million cycles. The phases of the rsync benchmark are clearly visible: (a) startup and page in; (b) ssh connect and network I/O; (c) build client file list; (d) build server file list; (e) compute rsync deltas; (f) transmit data; (g) shutdown wait for domain to terminate. Item (t) shows small peaks in kernel mode for regular timer interrupts.

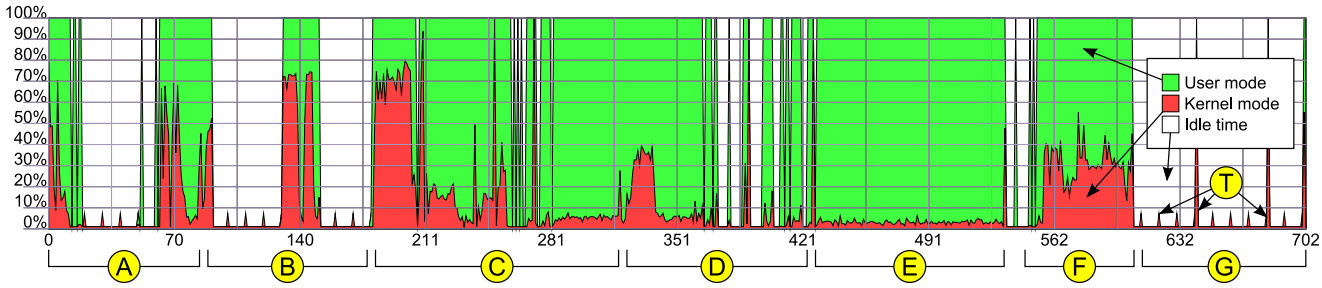
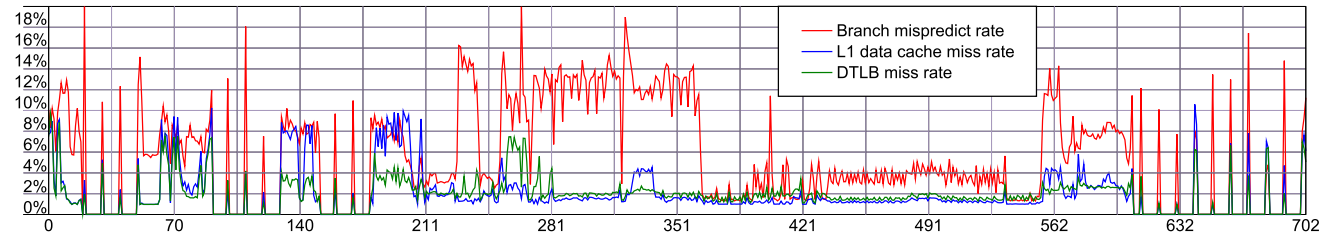


Figure 3. Time lapse graph of key microarchitectural statistics. The X-axis indicates the snapshot ID; snapshots were taken every 2.2 million cycles. Red lines indicate the percentage of all conditional branches that were mispredicted; green lines show the DTLB miss rate as a percentage of all loads and stores; blue lines show the L1 data cache miss rate as a percentage of all loads.



too early or forwarding the wrong store to a load) will also quickly result in easily debugable functional errors. Integrated cycle accurate or RTL-level simulators are often preferred by commercial microprocessor designers since they force rigorous design practices, but they can also be inflexible, since microarchitectural structures are often hard coded with many fragile special cases to ensure correctness.

In contrast, *trace based simulators* first run the entire benchmark on a fast functional simulator, with the result of every instruction recorded into a trace file. A separate timing simulator then reads in the trace file and passes each instruction (and its pre-computed result) through the simulated pipeline. This has several benefits, including the ability to guarantee deterministic behavior, experiment with perfect branch prediction and perfect speculation accuracy and still produce accurate performance data even if the timing simulator is not fully developed or has bugs. Unfortunately, traditional trace based simulators have many disadvantages: they cannot model speculatively fetched (but not committed) instructions, and complex interactions between speculative instruction results and performance (such as in load store queues) cannot be accurately modeled.

Split phase simulators take a middle ground: each instruction is actually "executed" in program order on a fully developed functional simulator immediately before it enters the cycle accurate pipeline, so as to obtain its correct value; in effect, a short trace of instructions and correct values is continually formed in a circular buffer feeding the timing simulator.

6.2. Instruction Set Support

The majority of all cycle accurate simulation tools available to researchers today target either the Alpha or MIPS instruction sets, or artificial simulation-only instruction sets like PISA [6]. We believe this is of great detriment to the future of high fidelity microarchitectural simulation, since most commercial silicon implementations of these architectures have been discontinued (although MIPS is still used in the embedded market, it is generally restricted to simple in-order implementations). This lack of real hardware makes it impossible to do co-simulation, verification or comparison to actual silicon, very little relevant software (for servers and workstations) is still compatible with either instruction set, and the specialized compiler toolchains required to build benchmarks are no longer being maintained and updated to modern compiler technology.

Currently, *PTLsim* is the only open source tool we are aware of that models a modern x86-64 machine at the micro-architectural uop level. The only other tools in this class are locked up inside x86 vendors like Intel and AMD, and even these internal tools lack configurability and other desirable research-oriented features, since they are typically intended to model a specific silicon revision.

Despite the current absence of x86-specific tools other than PTLsim, there are numerous other cycle accurate superscalar out of order simulators [13, 2] for the Alpha and MIPS instruction sets, as well as a few for SPARC and

PowerPC. For completeness, we will still review several these well known simulation frameworks for other instruction sets, and compare these tools with PTLsim. While the following list is by no means complete, it represents the majority of the simulation tools in use by researchers today.

6.3. Overview of Simulation Tools

SimpleScalar [6] is one of the better known and oldest simulators used in academia. It supports the Alpha instruction set, as well as PISA, a MIPS-like synthetic ISA developed specifically for SimpleScalar. It models an out of order processor using an RUU (an ROB-like structure) to control instruction flow and approximate various performance statistics. SimpleScalar is a *split phase* simulator, as described in Section 6.1. Unfortunately, SimpleScalar and its variants are beginning to show their age and suffer from several key disadvantages. First, its instruction set support excludes x86, the most widely used ISA in the world. In addition, non-standard or outdated compiler versions (such as the default, gcc 2.7.2) must be used to compile programs specifically for the PISA or Alpha ISA; therefore it is only useful for simple self-contained benchmarks like SPEC CPU. Second, SimpleScalar is a userspace only simulator, meaning it only simulates the microarchitecture up to the point where system calls are made. This is not necessarily a bad thing for CPU-intensive benchmarks, but it obviously makes it impossible to accurately model TLBs (and the difference in cache conflicts between virtual and physical addresses) or account for time spent in the kernel, waiting on multi-processor synchronization or accessing devices.

Fortunately, modern variants of the original SimpleScalar code have addressed many of these weaknesses. For instance, *M-Sim* [10] is a SimpleScalar 3.0 based Alpha simulator focused simultaneous multithreading. M-Sim explicitly models issue queues, replay and so forth, and adds pseudo-SMT support: several independent benchmark processes (in different logical address spaces) can be run in parallel on the SMT core. However, M-Sim still lacks support for simulating x86 code, and has no native mode co-simulation support. Since M-Sim is a userspace only simulator, it cannot run true shared memory multi-threaded programs involving lock contention, and cannot model kernel-level interactions with the operating system scheduler, nor can it model cache conflicts between threads based on real physical addresses.

The *M5 Simulator System* [7] for the Alpha ISA provides an easy to adapt C++ implementation, plug-in cycle accurate CPU models, full system support for booting Linux (for Alpha) kernels and multi-processor support. Therefore, M5 more closely matches the feature set of PTLsim, but is unfortunately still tied to the Alpha ISA and hence has no native mode co-simulation support nor the ability to run unmodified x86 software. M5 does have the advantage of more customizable cycle accurate device models typically found on Alpha systems (for disks, network adapters, buses, etc), whereas PTLsim currently supports only the Xen virtual device models for these device types. Both M5 and PTLsim support both full system and usermode-only operation, as well as full SMP and multi-threaded SMT operation.

SESC [8] is another well known simulator for the MIPS ISA, but unlike M5, which focuses on microarchitectural pipeline accuracy, SESC provides more extensive support for a variety of multi-processor models and nontraditional microarchitectures. SESC uses an event driven split phase core model, with the MINT [9] MIPS emulator framework as a functional backend.

TFSim (Timing First Simulation) [11] is a new twist on the *split phase* approach. In TFSim, each instruction is executed by *both* a functional simulator (the Simics [5] SPARC model) and a cycle accurate timing simulator. The timing simulator drives the functional simulator: when an instruction is fetched, the functional simulator also executes that instruction, precomputing its result. The instruction then passes through the cycle accurate pipeline where it is executed again. The two results are compared at commit, any mismatches result in a pipeline flush on roughly 0.003% of all cycles. PTLsim's sequential uop-level functional simulator provides the same functionality to PTLsim's cycle accurate cores as Simics provides to TFSim's timing core, respectively, such that we could integrate the TFSim concepts of self validation and approximate execution into PTLsim. Most importantly, unlike Simics, PTLsim is fully open source - this allows the internal uop ISA shared by both functional and timing accurate cores to be adjusted for an optimal fit.

6.4. General Advantages and Disadvantages

In Section 5 and Figure 2, we illustrated how full system simulation is critical to obtaining accurate performance results when interaction with the kernel or virtual memory is present. In multi-processor or multi-threaded environments, full system simulation becomes important to *all* applications, even if they are strictly compute-bound. Several of the simulation tools reviewed in this section, including SESC [8] as well as SimpleScalar-derived simulators like M-Sim [10], support "pseudo" symmetric multithreading or multi-core models. In this arrangement, two or more independent user-space programs in separate address spaces are run in parallel on an SMT or CMP model. These simulators typically cannot run true shared memory multithreaded programs involving lock contention, limiting their usefulness to simple single process benchmarks.

It is also unclear how accurate results can be obtained without simulating the kernel's thread scheduler and interrupt handling code, TLB pressure from multiple hardware threads, and physical address based NUMA and cache coherence interactions. It should be noted that full system simulators like PTLsim (and a few other tools such as M5 [7]) do not have this problem, since they model the bare hardware and devices, allowing all kernel and user code to be simulated. PTLsim also faithfully models all lock contention in terms of real interlocked x86 instructions and their uops using the same semantics as Pentium 4 hyperthreading. However, it should be noted that we cannot fairly

compare PTLsim's accuracy or performance with the other simulators in this section, since all the other tools are for non-x86 instruction sets.

None of the existing simulators reviewed so far support native mode co-simulation, in which uninteresting portions of the target code are executed directly on the host CPU at full speed. With existing tools, users must slowly execute through billions of cycles of initialization code in benchmarks to get to the interesting parts, and since most other tools are based on discontinued instruction sets, it is impossible to compare the simulator's output with a real machine on an instruction-by-instruction basis using the x86 hardware debug and single step facilities. In contrast, PTLsim's native mode technology eliminates both of these limitations, as described in Section 2.3.

7. Conclusion

In this paper, we described how we extended PTLsim into a full system simulator by integrating it with the Xen hypervisor, and how we added multi-processor and multi-threading support to PTLsim. We first first described what it takes to perform cycle accurate modeling of a complete x86 machine at the uop (micro-operation) level, along with the challenges and requirements for effective full system multi-processor capable simulation. We also disclosed the PTLsim out of order core's features, and we described how native mode co-simulation integrates with PTLsim's core models. We detailed the internal architecture of full system PTLsim and how it interacts with the Xen hypervisor, as well as how we model the various system-level aspects of the x86 architecture. The challenges of accurately modeling the flow of time in a simulation environment were also discussed.

We experimentally evaluated PTLsim's real world accuracy by configuring it as close as possible to a real AMD Athlon 64 machine. We then ran a demanding full system client-server networked benchmark (rsync with ssh and TCP/IP) inside PTLsim. We compared numerous statistics between PTLsim and the real K8-based core's performance counters, and we used these data to prove that PTLsim can be used to very accurately model commercial microprocessors, in many cases within 5% of the real hardware. We described why PTLsim offers a distinct advantage through its focus on x86 (the most widely used ISA in the world), allowing us to provide native mode execution for high speed as well as co-simulation for validation and debugging. We used our simulation data to show that userspace only simulation cannot deliver accurate results for many important workloads.

Our future work with PTLsim will focus on two areas. First, we will construct detailed multi-processor interconnect models with full MOESI cache coherence and the associated overhead (instead of the current instant visibility model). These models will be built upon the multi-core and SMT infrastructure PTLsim already provides, and will automatically enhance the multi-processor fidelity of all simulation cores. Second, we will extend PTLsim to support Xen's HVM (hardware accelerated virtualization) technology, allowing arbitrary operating systems to be modeled even if they do not support the Xen virtual devices. We believe that PTLsim will be a major step forward in the field of microprocessor simulation, powered by the widely used x86 instruction set and the high fidelity full system multi-processor modeling PTLsim delivers.

Acknowledgements. We would like to thank Hui Zeng (PTLsim SMT core developer) and other members of the CAPS research group at Binghamton for their technical contributions. We also thank the reviewers for their invaluable suggestions. This research was sponsored in part by NSF grant CNS0454298.

References

[1] M. Yourst. *PTLsim User's Guide and Reference.* Technical report at http://www.ptlsim.org

[2] D. Lilja, J. Ye. *Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations.* IEEE Trans. on Computers, Vol 55 N 3, p268, March 2006.

[3] Bochs IA-32 Emulator Project. http://bochs.sourceforge.net

[4] F. Bellard. *QEMU Internals.* Tech Report, 2006. http://www.qemu.org/qemu-tech.html

[5] P. Magnusson et al. *Simics: A Full System Simulation Platform.* IEEE Computer, Feb. 2002 (Vol 35 N 2), p50.

[6] T. Austin et al. *SimpleScalar: An Infrastructure for Computer System Modeling.* IEEE Computer, February 2002

[7] The M5. Simulator System. http://m5.eecs.umich.edu

[8] P. Ortego, P. Sack. *SESC: SuperESCalar Simulator.* Tech Report, Dec. 2004. http://sesc.sourceforge.net/sescdoc.pdf

[9] J. Veenstra, R. Fowler. *MINT: a front end for efficient simulation of shared-memory multiprocessors.* Proc. of Modeling, Analysis, and Simulation of Comp and Telecom Systems, 1994, p201.

[10] J. Sharke. *M-Sim: A Flexible, Multithreaded Architectural Simulation Environment*. Tech Report CS-TR-05-DP01, Dept. of C.S., State Univ of New York at Binghamton, Oct 2005. http://www.cs.binghamton.edu/~jsharke/m-sim/

[11] C. Mauer et al. *Full System Timing-First Simulation.* Proc. ACM SIGMETRICS 2002, p108

[12] O. Mutlu. *Efficient Runahead Execution Processors.* PhD Dissertation (Univ. of Texas at Austin), Chapter 4, p69. August 2006.

[13] WWW Computer Architecture Page: Simulators. http://www.cs.wisc.edu/~arch/www/tools.html

[14] Xen Community Overview. http://www.xensource.com/xen

[15] B. Clark et al. *Xen and the Art of Repeated Research.* Proc. USENIX, 2004.

[16] Software Optimization Guide for AMD64 Processors, Appendix A. AMD Pub. 25112, Rev 3.06, September 2005.

[17] H. de Vries. *Understanding the Detailed Architecture of AMD's 64-bit Core.* http://www.chip-architect.com

[18] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization: The* rsync *Algorithm.* PhD Dissertation (Feb 1999), http://samba.org/~tridge/phd_thesis.pdf

[19] R. Desikan et al. *Measuring Experimental Error in Microprocessor Simulation.* Proc. ISCA 2001.

[20] PTLsim model of AMD K8 Microarchitecture. http://www.ptlsim.org/benchmarks.php