MPTLsim: A Cycle-Accurate, Full-System Simulator for x86-64 Multicore Architectures with Coherent Caches

Hui Zeng, Matt Yourst, Kanad Ghose and Dmitry Ponomarev
Department of Computer Science
State University of New York, Binghamton, NY13902-6000
{hzeng, yourst, ghose, dima}@cs.binghamton.edu

Abtsract: *The introduction of multicore microprocessors in the recent years has made it imperative to use cycle-accurate and full-system simulators in the architecture research community. We introduce MPTLsim - a multicore simulator for the X86 ISA that meets this need. MPTLsim is a uop-accurate, cycle-accurate, full-system simulator for multicore designs based on the X86-64 ISA. MPTLsim extends PTLsim, a publicly available single core simulator, with a host of additional features to support hyperthreading within a core and multiple cores, with detailed models for caches, on-chip interconnections and the memory data flow. MPTLsim incorporates detailed simulation models for cache controllers, interconnections and has built-in implementations of a number of cache coherency protocols.*

1. Introduction

We introduce the design of ***MPTLsim -*** a general full-system multicore simulator that supports multiple threads on each core and provides detailed, cycle-accurate implementation of coherent caches, interconnections and memory systems. As a major advantage, the simulation framework is based on the widely-used X86-64 instruction set architectures (ISAs). This permits us to implement a full system simulator that goes well beyond the commonly available user-mode simulators widely used in academia today and enable the accurate simulation of not just the user-level, but also kernel level activities. The use of the X86-64 ISAs also permits native mode execution on the widely available PC hardware platforms to significantly speed up simulation over portions of the simulated code that are not of interest.

Academic research has been largely confined to RISC ISAs that are either obsolete (such as the DEC Alpha ISA) or not in widespread use (such as MIPS, SPARC, PowerPC) across platforms ranging from laptops to servers. In contrast, MPTLsim is a cycle accurate, uop-accurate full system simulator for **multicore** implementations of the X86-64 ISA running on Linux. MPTLsim leverages a cycle-accurate, full-system mode PTLsim simulator [26,27] for the popular X86 and X86-64 ISA that was developed by our research group and publicly released in 2005 (http://www.ptlsim.org).

MPTLsim has a number of attractive features:

- The use of the pervasive X86-64 ISA permits the tool to be used across a wide range of platforms and also allows accurate, practically verifiable results to be produced.
- The use of the X86-64 ISA gives us ready access to the full suite of software toolchains: debuggers, compilers, profiling tools and the like that are being constantly developed and upgraded at a time when similar support for ISAs used in other multicore simulators (such as Alpha, MIPS, SPARC etc.) are becoming unavailable or relatively limited in availability.
- The use of a full system mode simulation framework, where executions of user space code are simulated accurately with executions of operating system kernel and library level code on the native hardware has a number of advantages in itself, including the abilities to: (a) fast forward to any point of interest at a very high speed to any desired point in the execution by executing the application/system code directly on the native hardware; (b) use the native debugging tools in a seamless fashion; (c) switch as often as needed between the simulation mode and the real execution mode to observe deviations from the simulated results and the actual executions as recorded in internal x86 performance monitoring counters (*model specific registers*, MSRs) of the native hardware.
- The cycle-accurate and *uop-accurate* nature of our simulator makes it possible to undertake detailed studies on performance aspects and gather statistics on switching events within the core, caches, interconnections and coherence controllers. Such fine-grained data can be used for studying power/energy dissipations and their rates as well as other transient phenomenon (such as inductive noise, thermal emergencies and so on). We use the term “uop-accurate” to imply that the breakdown of X86 instructions into equivalent sequences of RISC-like operations (called uops) is modeled accurately.
- The use of the native mode execution framework and a parallel version of our simulator suite permit the performance capabilities of native, multi-core hardware to be exploited in speeding up the tool itself.

All of these advantages and features come built into a single, unified out-of-the-box solution that runs on a wide variety of hardware platforms under the popular Linux distributions. We also show that the simulation speeds obtained using MPTLsim are about 8 times better than that of an existing and widely-used multicore simulator in spite of the fact that MPTLsim models all datapath artifacts, the core, the interconnections, and the memory system in the most detailed level required for simulation accuracy.

2. Infrastructure for Simulating Multicore Systems

MPTLsim uses a domain built on a hardware abstraction provided by the Xen hypervisor [25], supporting a number of domains. Domain 0 has a special implication - and provides access to the real drivers and the outside world. All I/O activities from other domains are directed through Domain 0 by forwarding the system calls for I/O to Domain 0. All other domains have a hardware virtualization supported by the core PTLsim components. This virtual hardware supports multiple virtual CPUs (VCPUs) that are implemented using the PTLsim simulation components. When a domain is executing benchmark applications in the native mode, the PTLsim-based VCPUs are effectively bypassed. The VCPUs within a domain effectively implement a process context ("thread" in the industry jargon), that globally share the common address space of the domain. This makes it possible to implement simultaneously multithreaded designs, multiple cores, and multiple multithreaded cores easily using the basic VCPU rubric. MPTLsim builds on PTLsim's approach of using virtual machine technology to host multiple domains on a hypervisor layer. At this time, PTLsim uses the paravirtualization facilities of the Xen hypervisor [25], which provides extremely efficient virtual machine support for the domains, as the kernel is aware of the hypervisor interface. We also rely on some customization of Xen itself to permit efficient support for PTLsim, resulting in an overall 3% to 4% overhead.

MPTLsim makes use of a detailed uop-accurate, cycle accurate out-of-order (OOO) core design implementing the X86-64 ISA. The simulation loop in this case simulates a processor cycle and round robins through all the VCPUs to simulate events in the same cycle within each VCPU, the cache and memory, the interconnection and the coherence mechanisms. In spite of this, as we see later, we get about an average speedup of 7-8 times (depending on the number of the simulated cores) against GEMS running in a very similar mode. As X86-64 instructions are fetched, they are broken down into RISC-like uops in the process of decoding. Basic-block based uop trace caches are used to speed up the overall execution.

To support full system simulation, interrupts need to be correctly simulated in time. As the simulator is running at a slower pace than the real system, interrupts occurring in real-time need to be simulated properly, albeit at the slower simulation pace. To correctly implement interrupt handling by the VCPUs, we use Xen's checkpointing mechanism to checkpoint the state of the VCPUs and modify the device drivers to log the device interrupts and timestamp them with the simulation cycle counter. To process and serve the interrupt, we resume execution from the nearest checkpoint established just prior to the interrupt's time of occurrence and re-execute instructions and the interrupt (which is effectively a *virtual interrupt*!) from the log file - *at the simulation speed.* Specific techniques for supporting native mode execution in MPTLsim parallel those in PTLsim [27].

3. Simulating Hyperthreading in MPTLsim

To support a simultaneously multithreaded (SMT, aka hyperthreaded) core, the OOO single core PTLsim was extensively modified to implement an Intel P4 style hyperthreaded datapath and some variants. Each VCPU maintains a *context* structure that holds all information about the VCPU/thread context. This includes the values of the X86 architectural registers, X86 machine state registers (MSRs), page tables (as PTLsim implements its own memory management mechanism [27]) and various PTLsim internal state variables. One of these state variables in the context is a variable called *running*. Suspended contexts - that were suspended on the execution of the X86 *hlt* instruction, awaiting an external interrupt, for instance or similar blocked on hypercalls - wait on this *running* flag. When this flag is set, a VCPU resumes its processing. This flag is used in implementing virtual interrupt handling and also during the boot process for the VCPUs.

The basic core design and other datapath components have to be modified as well to support multithreading. The core features implemented at this time support several variants of a X86-64 datapath and are as follows:

- Shared resources in the SMT model include physical registers, function units, the issue queue and the caches. Resources dedicated to each thread include the rename table, reorder buffers and the load-store queues.

- The common issue queue in the original PTLsim design is split into 3 separate issue queues: one for integer and memory uops, another for floating point uops and a third one for control flow related instructions. Various selection policies are available for instruction issue step.

- Branch predictors were extended to be either thread private or shared.

- The original PTLsim core lacked prefetchers. The SMT extension to the cores adds in options to select among different prefecting mechanisms for use by the L1 and L2 caches. The choices available at this time are a next line prefetcher, a stride-based prefetcher, as used in some contemporary Intel designs [8] and a global history-based prefetcher [15]. There is also an option for limiting the extent of prefetching ("prefetching degree") from each thread.

- Two broad types of strategies are currently implemented for the use of some of these shared resources - one guarantees a minimum allocation of such resources to a thread and permits the rest of each resource type to be freely shared and the other implements both I-count and DCRA-count mechanism [21, 5].

- A common front end is used to fetch instructions from a shared L1 I-cache. The request queue to the L1 I-cache can be configured to have per context (i.e., per thread) regions or as a unified queue, with round-robin scheduling of requests across contexts. This permits some degree of fairness in accessing the L1 I-cache. Additional facilities are built in to permit a more general form of selecting (or denying) I-cache access requests from the various threads. Similar strategies are implemented for handling the L1 D-cache requests.

- The SMT extensions to the original PTLsim design also includes the incorporation of a memory bus model that supports the pipelining of memory requests/responses and the modeling of simple bank conflicts.

The SMT core design also needs to implement atomic operations correctly to preserve the overall semantics of hyperthreading at the simulation and full system level. The X86 ISA allows the uses of the LOCK prefix to execute memory instructions atomically. The *xchg* and *xadd* instructions and test-and-set instructions, as well as memory barrier instructions (like *sfence*) require similar indivisible memory operations. The original PTLsim design was modified to support these operations correctly in the face of multiple requests by the hardware or the VCPUs and to arbitrate among the atomic accesses in a fair manner across the contenders.

4. Implementing Multiple Cores and Coherent Caches

The implementation of the multicore version of PTLsim is based on the SMT datapath design described in the previous section. A maximum of 16 VCPUs can be supported as threads, as various combinations of cores and threads on the cores, ranging from all 16 threads on a single core to a single thread on each of 16 cores, covering all combinations in-between. The main challenge in extending the SMT cores to support multicore designs is centered on the design of the memory hierarchy, including up to two levels of caches and the incorporation of support for cache coherence.

4.1 Supported Configurations

At this time, the following multicore configurations with symmetric cores are supported for the X86-64 architectures:

- A configuration with private L1 caches and a shared L2 cache or private L2 caches, with prefetchers between the cache levels and a prefetcher for the L2 cache, with no support for coherent caches. These configurations are similar to the current dual-core and quad-core offerings from Intel. The only support for explicit memory coherence in these configurations are as described in Section 3 for the SMT.

- A cache-coherent multicore configuration with private and coherent L1 caches and a shared L2 cache. The interconnection used for maintaining coherence can either be a split-transaction shared bus or an atomic bus or a crossbar switch supporting broadcasts. Both MESI and MOSI cache coherence protocols are supported.

- A cache-coherent multicore configuration with two levels of private caches, with MESI or MOSI coherence for the private L2 caches and MSI coherence for the private L1 caches. The interconnections supported are again an atomic transaction bus or a split-transaction bus and a crossbar switch.

Although, support for directory-based coherent caches is not provided at this time, a generic interface for maintaining coherence is implemented at each level, easing the implementation of a variety of cache coherence protocols.

4.2 Assigning Threads to Cores and Initialization

Two configuration variables are used to specify the multicore configuration and the number of threads running on a core: *threads_per_core* and *number_of_cores*. The multithreaded application to be run can specify an affinity of a thread to a particular core, with cores numbered from 0 onwards. When no specific affinity is specified, the threads are simply allocated to cores based on their thread ids, with thread numbers 0 thru (*number_of_threads* – 1) assigned to Core 0, thread numbers *number_of_thread* thru (2* *number_of_threads* – 1) assigned to Core 1 and so on. Core affinities and thread ids are specified to the MPTLsim hypervisor through a system call and these are eventually stored as part of the context structure for a thread (Section 3). A barrier synchronization implemented directly in the simulator code is inserted into the code of each thread before the multicore simulation starts – this simply ensures that all cores start executing their assigned threads from a pre-determined position in their code. When multiple threads are assigned to a single core, a local thread switch occurs to enable all threads assigned to a core to report at this barrier.

4.3 Implementing Caches and Coherence Controller

The original PTLsim design implements caches as an integral part of each core and implements only the tag part of the cache explicitly, with no explicit structures to hold the data part of the caches, as in other simulators, such as [1]. To ease the configuration of caches, including shared caches and the implementation of a cache coherency protocol, all caches are implemented as a separate global structure, distinct from the cores. When two levels of caches are present, a configuration parameter, *number_of_core_per_l2* is used to specify how many cores share a L2 cache. Additionally, the data part of the caches is also implemented explicitly to hold both the contents of a cache line and state information in accordance with the coherence protocol.

All of the currently supported cache coherency mechanisms support the sequential consistency model. This requires a global serialization mechanism for memory accesses that needs to be implemented using the local cache controllers, the interconnection and the protocol implementation. MPTLsim uses a generic structure of the core-local coherence controller for the L2 caches. The structure for the coherence controller for the L1 caches is similar. Careful attention was paid to the design of the coherence controller to permit them to be adapted to a wide variety of coherence protocols, including directory based protocols.

The cache coherence controller for MPTLsim also designed to support different cache line sizes at the L1 and L2 levels. When two levels of coherent caches are present, the cache inclusion properly is maintained. A lower level cache entry for a L2 line tracks the presence of smaller L1 line-sized parts that are cached at the local L1 level using a vector of presence bits. Two levels of coherent caches make use of these presence bits and the current version of MPTLsim offers a MSI protocol for L1 caches with MESI at the L2 level. As in the case of the processor cores, the protocol and coherence controller activities are simulated on a cycle-by-cycle basis. Appropriate statistics are generated and logged for a large variety of events associated with the protocol actions. The nature of the information collected can be pre-selected.

Each cache controller modeled in MPTLsim maintains a separate set of request queues (in FIFO order) from the core side towards the lower levels – one queue for each of the threads running locally. The use of separate queues permits the implementation of a variety of request scheduling strategies via a module the "input scheduler". Once such a request is selected, a "dependence checker" checks if the request is to a cache line with a pending cache miss, stored in a separate pending "request table" within each controller. Each entry in this table consists of a cache tag and index, status flags and a pointer to a list of pending requests queued up on this line. The total number of entries in this table is set to the maximum number of outstanding requests per core, as specified in a configuration parameter. If the dependence checker detects a match with a pending cache line miss already processed off the request queues, the current request is appended to the linked list of pending requests on the same line, as maintained within the request table entry. Once the miss on a line is serviced, locally queued up requests on this line are also revived and serviced in their original request order. The cache controller design for MPTLsim also implements load forwarding and store merging (in a sequentially consistent manner) as part of the logic associated with the various request and response buffers.

Access requests to the local cache can come from a variety of sources – from the local processor, a lookup request initiated by another controller (such as a snoop triggered by bus activity in a bus-based system), a cache lookup required when a response to a previous request comes in from an external source (the lower level or the memory or another cache, depending on the protocol) or dependent requests re-activated on the arrival of a missing line or a previously queued cache read or write initiated by the local core. These requests are maintained in a structure called the "cache access buffer" and serviced using a dedicated port. The read and write requests from the local core use a second dedicated port and if a port happens to be busy in a given cycle, the local read or write is queued up on the "cache access buffer" structure. Requests in this buffer are serviced in a specific order (from highest to lowest) as follows: lookup requests triggered by other controllers, responses to a previous request from an external source, dependent requests that are re-activated dependent requests and previously queued read or write requests from the local core. On cache accesses from either of the ports, state changes are triggered and require the use of a protocol engine logic for updates to the state of a cache line. Updates to the data part of a cache line are also made, as necessary.

4.4 On-Chip Interconnection and Memory System

At this time, MPTLsim provides three types of built-in interconnection models:

- A shared bus providing atomic transactions.

- A split-bus that decouples bus requests and responses using two separate buses – one for addresses and one for data. Both address and data buses have independent arbiters. The specific bus modeled is as described in [10]. Each bus transaction consists of a number of phases/states as described in [10]. The address bus states include arbitration, *wait_address*_command and *address*_command. The data bus includes arbitration, wait_data_response and data_response. For the bus command *bus_upgrade*, only the address bus is used because the issuing cache controller already has the data. For the *bus_read* or *bus_write* commands, the requested data will be sent on data bus freeing the address bus to serve another request, if present. A writeback request will need to use both the address bus and the data bus and both buses must be granted before the data and address transmission commences, so care must be taken to avoid deadlock. In our implementation, potential deadlocks are avoided via resource reservations.

- A crossbar switch that has separate networks for requests and responses.

MPTLsim allows users to specify the duration of interconnection events in terms of an interconnection clock cycle, which in turn is specified as an integer divisor of the clock rate of each core. Appropriate hooks are also provided for collecting statistics on the various interconnection events and on the interconnection traffic volumes and rates.

5. Related Work

A large number of processor simulation tools have been developed in the last decade. One of the most popular cycle-accurate simulators is the Simplescalar toolset [1]. Simplescalar, as well as its variants [19], only supports the simulation of single-threaded workloads, such as SPEC benchmarks, in user-level mode. SESC simulator [17] developed at the University of Illinois, models a variety of architectures, including dynamic superscalar processors, CMPs, processor-in-memory, and speculative multithreading architectures. However, SESC does not support full-system simulation mode nor does it model contemporary ISAs like X86. A-Sim [9] is a modular and reusable performance modeling framework developed at Intel for Alpha ISA.

While PTLsim is the first open source *cycle-accurate* simulator for X86 instruction sets at the level of uops, numerous *functional* simulators (tools that precisely emulate each instruction in the target instruction set, but do not provide any cycle-accurate timing information) have been developed for x86. Bochs [4] is a well-known open source X86 functional simulator, which supports nearly all X86 features. QEMU [16] supports multiple CPU host and guest architectures and uses dynamic compilation to achieve significantly faster simulation speeds compared to pure interpretation. Simics [20] is a commercial functional simulation suite for various processor families (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 (just like any other functional simulator).

GEMS [14] is a full-system multiprocessor simulator that leverages the power of Simics [20] as an underlying foundation. GEMS is arguably the most popular and versatile multicore simulator available in the public domain. In addition to the simulation facilities, GEMS incorporates extensive features to design, specify and validate new cache coherence protocols. The GEMS toolset adds a fairly detailed out-of-order processor simulation model (called *opal*) and detailed memory model (called *ruby*) on top of the functional full-system simulation environment provided by Simics. This enables the simulation of commercial software such as database systems running on the Solaris operating system. For increased simulation speed, GEMS can be used without the *opal* module by directly feeding the *ruby* module from Simics. For studies where the impact of the out-of-order execution features is secondary, such a configuration can significantly increase the simulation speed. When comparing the simulation speed of the multicore version of PTLsim versus GEMS, we used GEMS with and without *opal* module and report the results for both configurations in the results section. Since it is built on top of the freely available version of Simics, GEMS inherits all of its limitations such as the capability to only simulate the SPARC ISA. It is, for example, impossible to use GEMS to simulate the execution of X86 programs running under Linux operating system. Another potential limitation stemming from the reliance on Simics is somewhat restricted flexibility due to the black-box nature of the functional model. Furthermore, licensing limitations can restrict the widespread adoption of the tool, particularly in non-academic environments. SimFlex [23] – another recently designed tool that uses Simics – potentially has similar limitations. The newly-released FeS2 execution-driven full-system simulator for X86 ISAs, based on the uop decode and scheduling code of our original PTLsim simulator [28] has been designed to eventually work with the Ruby module of the GEMS toolsuite, so multicore configurations can be simulated. Further details of multicore simulations with FeS2 simulated cores are unavailable publicly at this time.

Another approach to the design of full-system simulator is to add the full-system simulation capabilities to existing user-level timing simulators – this is exemplified by the M5 simulator [3]. The M5 simulator provides an easy-to-adapt C++ implementations, plug-in cycle-accurate CPU models, full system support for booting Linux kernel and multi-processor support. In terms of these features, M5 closely matches the capabilities provided by PTLsim, but at this point it only supports the simulation of Alpha and SPARC ISAs.

Several techniques have been proposed to convert the existing uniprocessor simulators into parallelized simulators of multicore processors. Donald and Martonosi [7] proposed a methodology, where each core of a multicore system is simulated by a single POSIX thread, such that the synchronization between these threads is only invoked when the actual inter-core communication occurs. They also demonstrated the application of their technique to parallelize Turandot and Simplescalar simulators. An alternative method [11] uses approximate model for shared resources, which requires second simulation pass and makes it impossible to perform experiments studying the inter-core dynamic adaptation policies – one of the important potential usages of our tool. In [6], a parallel version of Simplescalar using MPI is described.

To accelerate multicore simulations, techniques based on the use of FPGAs in conjunction with software simulation are actively explored by the architecture simulation community. An example of such an approach is the RAMP effort (Research Accelerator for Multicore Processors) – a collaborative research project between several universities and industry groups [18]. While significant simulation speedups can be realized, FPGAs that are capable of simulating the entire high-end x86-based cores are relatively expensive.

Aside from simulations, another technique to analyze program behavior is through dynamic binary instrumentation. Instrumentation inserts extra code into the application to observe its behavior. A recently released Intel's Pin instrumentation tool [12] supports Itanium, IA32 (32-bit x86), EM64T (64-bit x86), Itanium and ARM architectures. Another useful instrumentation framework for building dynamic analysis tools is Valgrind [22]. Unfortunately, instrumentation does not model the details of the out-of-order processors and is not usable for modeling the non-existing hardware. Ideally, the advantages of instrumentation tools (such as easy prototyping and very fast runtimes) should be combined with the accurate models provided by the cycle-accurate simulators. In this respect, MPTLsim can serve as a perfect complement to dynamic instrumentation tools such as PIN to complete the framework for efficient and accurate design space exploration of X86-based multicore processors at the microarchitectural level.

6. Results and Discussions

In this section, we present the results of some initial studies using MPTLsim.

6.1 Simulation Speed

To measure the simulation speed of MPTLsim, we experimented with parallel benchmarks from Splash-2 suite. In all of our experimental studies, we used configurations of individual cores as shown in Table 1. Unless stated otherwise, both L1 and L2 caches were private to the core and kept coherent using MSI protocol for L1 and MESI protocol for L2 using a split-transaction shared bus. The bus clock cycle was one fourth of the core clock cycle.

Table 1. Configuration of the Simulated Processor Cores

Parameter	*Configuration*
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	Clustered IQ, 16-entry each, 64 entry LSQ, 128–entry ROB
Registers	128 Int and 128 FP
Function Units	2 ALU, 2 LD/ALU, 2 ST/ALU, 2 FP/SSE/MMX
L1 I–cache	64 KB, 4-way, 1 cycle latency.
L1 D–cache	64 KB, 4-way at 1 cycle latency (2 LD, 2 ST)
L2 Cache unified	4 MB, 8-way set-associative, 8 cycles hit latency
BTB and branch predictor	1024 entry, 4–way set–associative. Minimum branch misprediction penalty – 10 cycles. Gshare predictor.
Memory	128-bit wide, 160 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative

We estimated the simulation rates of MPTLsim running on an Intel 2.4GHz dual-core 64-bit processor with 4 GBytes of DRAM. We also compared some of our results to a very similar configuration on the GEMS toolsuite. In every sense of the word, a comparison between GEMS and MPTLsim is an "apples-to-oranges" comparison because of the differences in the ISAs (we simulated the SPARC ISA on GEMS), differences in the system call conventions and library functions, as well as differences in the cache coherence protocols. There is, of course, a difference in the implementation philosophies of these two simulators as well. Nevertheless, Figures 1 and 2 depict the number of processor cycles simulated per second for the Splash benchmarks executed on these simulators for 4-core and 8-core configurations. For GEMS, we report the results for two different simulation modes: a faster mode (labeled as *ruby-only* in Figures 1 and 2) that directly uses Simics to drive the GEMS memory module (*ruby*) and a slower mode (labeled as *ruby&opal*) that uses detailed out-of-order timing simulator of the core (*opal* module) to drive *ruby*.

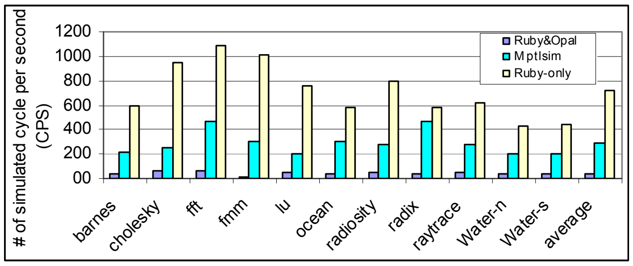

Fig 1. Simulation rates for GEMS and MPTLsim: 4 cores

Each benchmark was executed for 200 million committed instructions (in total, counting from all cores). All simulators were configured identically and the same number of instructions was skipped to warm up caches and branch predictors. While running simulations, we also ensured that no other tasks were executed on these machines.

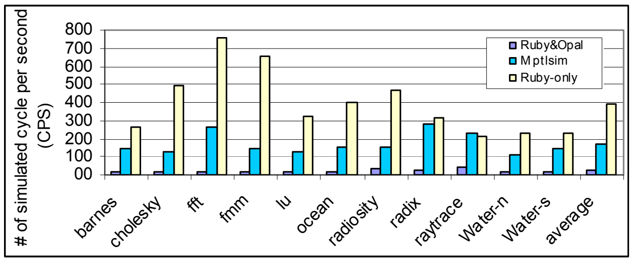

Fig 2. Simulation rates for GEMS and MPTLsim: 8 cores

The results of Figures 1 and 2 clearly demonstrate that the *ruby-only* mode provides the fastest simulation rate. This is no surprise, as this mode provides only the functional simulation of the cores. A more appropriate comparison would be between MPTLsim and *ruby&opal* version of GEMS. Note, however, that MPTLsim provides a cycle-accurate and uop-accurate simulations in contrast to the somewhat inexact timing simulation of opal [14]. In spite of this, MPTLsim provides a significantly faster simulation rate compared to *opal&ruby* variation of GEMS. On the average across all benchmarks, 4400 cycles are simulated in one second in the *ruby+opal* GEMS mode, 29500 cycles are simulated in MPTLsim, and 74200 cycles are simulated by "ruby-only" variation of GEMS. In other words, MPTLsim provides 6.7 times improvement in simulation speeds over GEMS and is about 2.5 times slower than the ruby-only simulations. Interestingly, adding the cycle-accurate out-of-order modeling capabilities to GEMS (through opal module) slows down the simulations almost by a factor of 17. In terms of the individual benchmarks, the closest gap between the performance of ruby-only model and MPTLsim is for radix benchmark – MPTLsim performance is only about 20% lower in this case.

For the simulated 8-core configuration, *opal&ruby* GEMS simulates about 2200 cycles per second on the average, MPTLsim simulates about 17000 cycles per second and ruby-only variation of GEMS simulates about 41000 cycles per second. With larger number of cores shown in this graph, the relative overhead of cycle-accurate out-of-order simulations increases, compared to the situation shown in Figure 1. Specifically, MPTLsim is now 7.7 faster than the opal&ruby GEMS, but still only 2.7 slower than ruby-only model. In other words, the performance gap between MPTLsim and full-fledged GEMS increases much more significantly than the gap between ruby-only model and MPTLsim, as the number of cores increases. These trends are expected to continue as the number of simulated cores increases further, making MPTLsim attractive from the perspective of detailed cycle-accurate simulations and simulation speed. We believe that much of the performance advantages of MPTLsim come from its use of a highly-tuned simulator code and the exploitation of specific features of the X86 ISA in speeding up the MPTLsim code. Some of these reasons are elaborated upon in [26, 27].

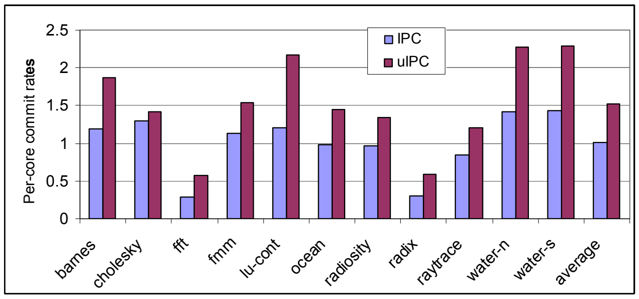

Fig 3. Instruction and uop commit rates

6.2. Some Examples Studies with MPTLsim

Figure 3 represents some interesting studies that can be performed using MPTLsim. Figure 3 shows the commit rates of X86 instructions and their RISC-like uop equivalents for various Splash benchmarks, averaged across each of the cores in the 4-core configuration. The number of uop equivalents for a single X86 instruction across a single benchmark clearly varies from one benchmark to another, ranging from 1.98 uops per X86 instruction (*radix*) to 1.24 uops per X86 instruction (*Barnes*). The reason for these variations can be traced to the dynamics of the instruction mixes in the user mode and the kernel mode for each benchmark as well as on the sharing characteristics and it is beyond the scope of this paper to provide detailed insight into this behavior. This is perhaps a topic worthy of further investigations treading on the holy ground of the RISC vs. CISC debate.

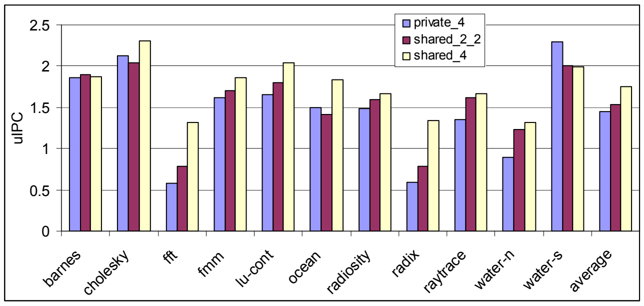

Fig 4. uIPCs for three L2 cache sharing configurations

Our next sample study looks at the impact of sharing L2 caches across cores. Three configurations of a 4-core design are studied here: a) the "private_4" configuration, where each of the 4 cores have their individual private L2 caches – this is the configuration with no L2 cache sharing; b) the "shared_2_2" configuration, where a pair of cores shares a common L2 cache; and c) the "shared_4" configuration, where all cores share a common L2 cache. The total L2 cache capacity was kept the same across all these configurations. The MSI protocol was used to maintain coherence at the L1 level and the MESI protocol was used to maintain coherent L2s where applicable. A split-transaction bus was also used in all configurations. Figure 4 shows the commit rate of uops (uIPC) across the Splash-2 benchmarks for the three L2 sharing configurations. The general trend is that with an increasing degree of cache sharing, the uIPC improves, as coherence traffic on the shared bus drops due to localized sharing by cores that have a common L2 cache. Most of the updates to such locally shared variables in "modified" or "exclusive" states do not trigger invalidations on the bus: there are fewer invalidations and a lower degree of bus contention. This explanation is borne out by the results of Figure 5. Figure 5 shows the relative number of bus upgrade transactions (labeled as UPGR in Figure 5) which are needed for invalidations, as well as the total number of bus transactions decrease with an increase in the number of cores that share the L2 cache(s). Note that only two of the configurations that generate the higher bus traffic are shown in Figure 5 due to space limitations.

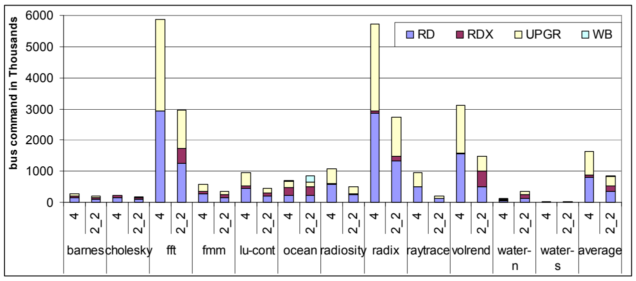

Fig 5. Breakdown of bus transactions for the two of the three L2 cache configurations

7. Concluding Remarks

Over the past 30 years, the architecture research community in academia has made a significant contribution to the development of accurate microarchitectural-level simulation tools. However, there is a noticeable void in the area of similar tools that provide cycle-accurate and uop-accurate simulations for realistic implementations of the X86 ISA. This void extends to the realm of multicore designs based on the X86 cores.

We introduced the design of MPTLsim – a cycle-accurate, uop-accurate full-system simulator for X86 ISA-based multicore designs. MPTLsim models out-of-order cores complete with detailed models for coherent caches, on-chip interconnections and the memory interface. The key advantages to using an X86-based ISA include the ability to access a continuously developed toolchain of programming environments, compilers and debuggers. Another advantage of simulating X86-based ISAs on the widely available X86 hardware platforms include the ability to seamlessly switch between simulation mode and native execution mode. The existence of instrumentation registers on most X86 implementations permit the simulation results to be validated against the results obtained from actual execution of the same program on the real hardware.

We also showed that the simulation speeds obtained using MPTLsim are significantly better than that of an existing and widely-used multicore simulator in spite of the fact that MPTLsim models all datapath artifacts in the most excruciating level of detail.

The initial version of MPTLsim, as described in this paper, will be publicly released through PTLsim's website at http://www.ptlsim.org. Additional work in progress includes the development of support within MPTLsim for distributed shared memory and directory-based coherent caches, the development of power and temperature estimation modules and support for heterogeneous cores.

Acknowledgements

This work was supported in part by NSF through Award CNS 0720811 and by Intel. We would like to thank the reviewers for their valuable comments and suggestions.

References

[1] Austin, T., et al., "Simplescalar: An Infrastructure for Computer System Modeling", IEEE Computer, February 2002.
[2] "Using the M5 Simulator", ASPLOS 2008 tutorial slides at http://www.m5sim.org/dist/tutorials/asplos_pres.pdf
[3] Binkert, N., et al., "The M5 Simulator: Modeling Networked Systems", in IEEE Micro, July-Aug. 2006, pp.52-60.
[4] Bochs IA-32 Emulator Project and related documentation at http://bochs.sourceforge.net.
[5] F. Cazorla, et al. "Dynamically Controlled Resource Allocation in SMT Processors", Proc. MICRO, 2004..
[6] Chidester, M., and George, A., "Parallel Simulation of Chip-Multiprocessor Architectures," ACM Trans. on Modeling and Computer Simulation, vol. 12, no. 3, pp. 176–200, July 2002.
[7] Donald, J., Martonosi, M., "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation", Computer Architecture Letters, August 2006.
[8] Doweck, J., "Intel Smart Memory Access: Minimizing Latency on Intel Core Microarchitecture", Tech. at Intel, September 2006.
[9] Emer, J., "A-Sim: A Performance Model Framework", IEEE Computer Magazine, February 2002.
[10] Gupta, A., Culler, D., Singh, P., "Parallel Computer Architectures", published by Morgan-Kaufmann, 1999.
[11] Li Y., et al., "CMP Design Space Exploration Subject to Physical Constraints", in Proc. of HPCA, 2006.
[12] Luk, C-K., et al, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", PLDI, 2005.
[13] Web page of the Liberty project at Princeton University: http://liberty.princeton.edu/Software/LSE/
[14] Marty, M. R. et al. "Multifacet GEMS: General Execution-driven Multiprocessor Simulator", tutorial presentation at ISCA 2005, available at: http://www.cs.wisc.edu/gems/
[15] Nesbit, K., Smith, J., "Data Cache Prefetching Using a Global History Buffer", Proceedings HPCA, 2004.
[16] Bellard, F., "QEMU internals", Technical Report, 2006. http://www.qemu.org/qemu-tech.html
[17] Renau, J. et al. "SESC Simulator", http://sesc.sourceforge.net
[18] Web page of the RAMP project at UC-Berkeley: http://ramp.eecs.berkeley.edu/index.php?about
[19] Sharkey, J., "M-Sim: A Flexible Multithreaded Architectural Simulation Environment", and software distribution available at http://www.cs.binghamton.edu/~m-sim
[20] Simics documentation at http://www.virtutech.com/news-press/press/2005/2005pr7.html
[21] D. Tullsen, et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." in Proc of ISCA, 1996.
[22] Open source and documentation for Valgrind are available at: http://valgrind.org/
[23] Wenisch, T., et al., "SimFlex: Statistical Sampling of Computer System Simulation", IEEE Micro, Aug. 2006, pp.18-31
[24] Wang, H., Zhu, X., Peh, L. and Malik, S., "Orion: A Power-Performance Simulator for Interconnection Networks" , In Proceedings of MICRO 35, Istanbul, Turkey, November 2002.
[25] Xen Community Overview. http://www.xensource.com/xen
[26] Yourst, M., "PTLsim User's Guide and References", and software distribution available at http://www.ptlsim.org
[27] Yourst, M., "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator", ISPASS, 2007.
[28] FeS2 Simulator pages at: http://fes2.cs.uiuc.edu/, 2008.