TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation

Sangyeun Cho Socrates Demetriades Shayne Evans
Lei Jin Hyunjin Lee Kiyeon Lee Michael Moeng

Department of Computer Science
University of Pittsburgh

Abstract

The slow speed of conventional execution-driven architecture simulators is a serious impediment to obtaining desirable research productivity. This paper proposes and evaluates a fast manycore processor simulation framework called Two-Phase Trace-driven Simulation (TPTS), which splits detailed timing simulation into a trace generation phase and a trace simulation phase. Much of the simulation overhead caused by uninteresting architectural events is only incurred once during the trace generation phase and can be omitted in the repeated trace-driven simulations. We design and implement tsim, an event-driven manycore processor simulator that models detailed memory hierarchy, interconnect, and coherence protocol models based on the proposed TPTS framework. By applying aggressive event filtering, tsim achieves an impressive simulation speed of 146 MIPS, when running 16-thread parallel applications.

1. Introduction

Thoroughly evaluating a new architectural or system design idea is a complex and often time-consuming process, entailing multiple stages of modeling efforts with different levels of accuracy and relevance [9]. Modern computer architecture research relies heavily on simulation techniques, especially to evaluate complex and subtle design trade-offs under a realistic workload. The flexibility and arbitrary level of detail that the software-based simulation techniques can provide is especially desirable [20].

Slow simulation speeds are a serious impediment to improving research productivity despite continuously increasing computer system performance [18–20]. For example, one minute of execution in real time can correspond to days of simulation time [18]; as such, simulating in detail the well-known SPEC2k CPU benchmark suite may take well over a month [19]. The growing complexity in hardware designs, the need for using diverse and long-running real-world workloads, and the current design trend of manycore processor chips all work together to aggravate the situation even further.

In this paper we propose and evaluate a practical simulation framework called *Two-Phase Trace-driven Simulation* (*TPTS*; pronounced “tip-see”), upon which very fast architecture simulators can be built. The goal of our framework is to facilitate fast testing of system design ideas before undertaking (more expensive) full-system simulation. We achieve high simulation speeds by splitting time-consuming cycle-accurate simulation into two distinct phases, the *trace generation phase* and the *trace simulation phase*, and employing a simple yet effective trace filtering technique in the trace generation phase to obviate the need for simulating uninteresting architectural events in the repeated simulation phase. Unlike many existing trace-driven simulation approaches that focus on extracting and simulating only memory references [14], we effectively reuse the detailed timing simulation results collected in the trace generation phase. Therefore, when using our approach, simulating certain local (intra-core) architectural events such as branch prediction and L1 cache access may be completely omitted during multicore processor simulations without introducing a large timing error, where the focus of the study is typically on the interconnection network and last-level memory structures.

We have designed and implemented a prototype multicore processor simulator called *tsim* based on the TPTS framework and evaluate it in terms of simulation speed and the accuracy of results. Compared with a simulator built on *Simics* [8] modeling a similar processor architecture, *tsim* achieves 151× the simulation speed once traces have been prepared. Given that the trace generation time is a factor of around 1.1 the simulation time of a Simics-based simulator, if five simulation runs are needed for a study, an overall simulation speedup of over 4 is achieved (over 8 if ten runs are needed). Further, *tsim* uses only 8.9% of memory space, and results in a 3.2% average CPI error for a suite of shared-memory parallel applications. The raw simulation speed of *tsim* was measured to be 146 MIPS (millions of simulated instructions per second) on a commodity Linux box.

This paper is organized as follows. In Section 2 we propose and describe the concept of TPTS as a framework on which very fast manycore processor simulators can be built. This section also addresses some of the limitations in modeling sophisticated out-of-order processor cores and the chal-

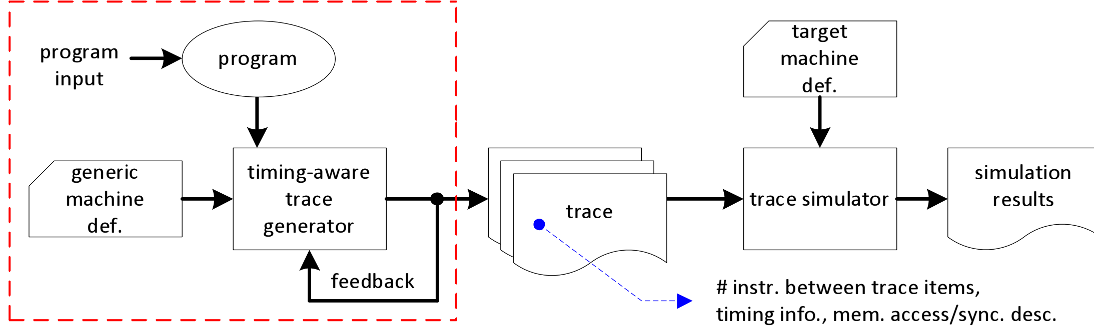

Figure 1. The proposed TPTS simulation flow. Dashed box encloses the trace generation phase.

lenges in handling multithreaded workloads that arise when using the trace-driven simulation approach. Section 3 describes our design and implementation of *tsim*. We present the measured speed, accuracy, and other costs of *tsim* in Section 4. We summarize related work in Section 5. Finally, conclusions and future goals are presented in Section 6.

2. Two-Phase Trace-Driven Simulation (TPTS)

In this section, beginning with the description of the two TPTS phases, we describe the operation, issues, and benefits of the proposed TPTS framework. Figure 1 shows the overall flow of TPTS.

2.1. Phase 1: trace generation

Trace generation plays a critical role for TPTS because the accuracy of trace-driven simulation depends on the information embedded in the generated traces. The usage of the trace should be clear before the generation, and the timing-aware trace generator is designed to meet this usage. The timing-aware trace generator will generate traces for the given program with the configuration of the underlying machine. The machine architecture used in this phase must be a close match to the machine architecture to be used in the trace simulation phase so that invariant timing information collected during trace generation can be accurately used in later simulations. For instance, the L1 cache configurations can be set identically in both the phases. If multiple L1 cache configurations are needed for investigation, multiple traces could be generated. At the same time, the processor components that need to be varied in the simulation phase must be modeled "ideally" in the trace generation phase so that related timing is solely determined in the simulation phase. For example, all L1 misses should "hit" in the L2 cache during trace generation. In studies focusing on the system-wide resources (*e.g.*, interconnection network and L2 caches) in a multicore processor, "fixing" certain intra-core structures such as L1 caches and branch predictor in the trace generation phase is acceptable.

In this work, we focus on the traces containing all memory references and on filtered traces that contain only L1 misses. Each trace item contains the number of cycles and instructions between two consecutive trace items, the type of memory access, and the referenced memory address. After this trace generation phase, the trace files will be fed into the trace-driven simulator. Trace files may be further analyzed or pre-processed before use. For example, we analyze traces generated from different cache access latency assumptions to create trace item annotations to more accurately model out-of-order processors (Section 2.3).

2.2. Phase 2: trace simulation

Once traces are prepared, the trace-driven simulator models the target machine architecture using the traces. Timing of simulated architectural events is derived from two sources: invariant timing information embedded in each trace item and the information from the architectural components that are simulated in the trace-driven simulator. In multicore architecture research often the most important timing information and simulation events are generated from these architectural components. As discussed earlier, the configurability of the machine architecture in this phase is limited by the machine configuration used in the trace generation phase. For example, the collected trace files may only be for a 16-core machine. Thus, any simulation using this trace would also be limited to 16 cores. Hence, the purpose of the simulation is very important in determining the respective roles of the trace generation and simulation phase.

For a multithreaded application to run on N processors, we generate one trace file for each thread and, as such, we import N distinct trace files in the trace simulator. Shared memory synchronization primitives, such as Get-Lock and Barrier, become separate trace items in the trace files such that their function and system-wide effect (*e.g.*, traffic) can be accurately modeled using the machine configuration of the trace-driven simulator.

To approximate the errors from trace simulation, we ran many experiments with the sim-outorder simulator [1] and our testbed simulator using selected SPEC2k CPU benchmark programs [13]. The sim-outorder simulator was used to generate traces for the testbed simulator, which is a trace-driven simulator that models a two-level on-chip memory hierarchy. The goal of the study was to find how closely

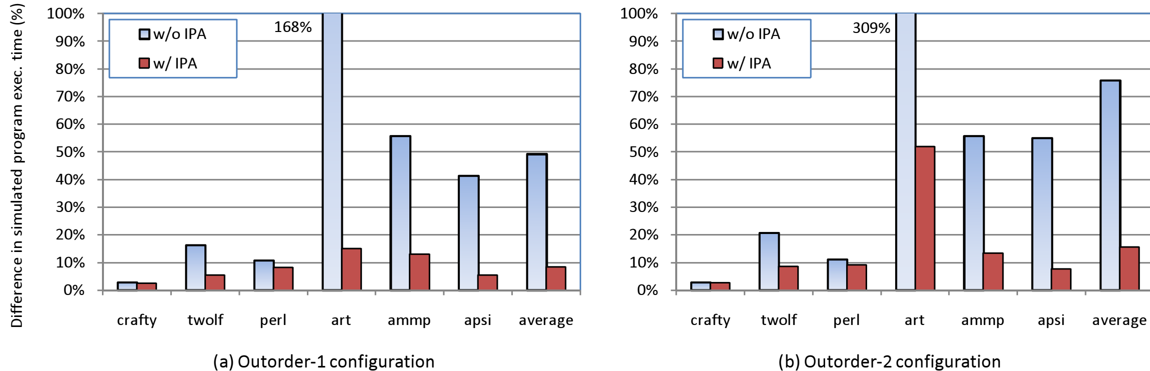

Figure 2. Simulated execution time difference between sim-outorder and our testbed–without and with instruction permeability analysis (IPA). Outorder-1/2 uses a 4-issue out-of-order processor configuration having a 32-/64-entry RUU. We use 16 kB, 4-way L1 caches and a 4k-entry combined branch predictor.

the results from the two simulators agree when an identical machine model is used. We tested the two simulators with both in-order and out-of-order processor configurations and filtered and unfiltered traces. The timing differences come from two major sources: difficulty in fully mimicking the sim-outorder simulator's timing-related behavior [1] and the difficulty of the trace-driven approach to model an out-of-order processor [4]. The major observations we make from this study are: (1) simulation with filtered traces, generated on only a subset of memory accesses, leads to a smaller timing error than simulation with unfiltered traces; (2) an out-of-order processor configuration resulted in larger errors than an in-order processor configuration; and (3) the timing difference between sim-outorder and our testbed for in-order configurations was very limited.

The reason for the first observation is that we may accumulate more error as we use more trace items when there is a discrepancy in timing assumptions between the simulators each used for trace generation and trace simulation. Regarding the second observation, we note that modeling a superscalar processor using a trace-driven simulation methodology has been recognized as a hard problem [4]. In our experiment, simulating a superscalar processor in a naïve manner resulted in a large program execution time difference which reached 309.5%. Finally, for in-order processor configurations capable of executing one or two instructions per cycle, we observed small timing differences of 0.5 to 9.0% with an average of 4.6%.

From this study, we conclude that the proposed TPTS framework is accurate in simulating in-order processor architectures. In the following two subsections, we address in more detail issues related with simulating an out-of-order processor and modeling a shared memory application.

[1] For instance, it is hard to fully eliminate timing errors even for the in-order configurations due to an implementation artifact of sim-outorder. Instructions slip into buffers in different pipeline stages even when there is a stall condition, *e.g.*, a cache miss.

2.3. Modeling out-of-order processors

Out-of-order processors are much harder to model reliably using a trace-driven simulator than in-order processors due to dynamic instruction scheduling [4]. The order and the impact of important simulation events are affected by the microarchitecture in a complex manner and simulation events affect subsequent events. For instance, a cache miss may affect the time to verify subsequent branch predictions in a pipelined, out-of-order processor. Such an effect is difficult to reproduce in a trace-driven simulator using static trace files. Fortunately, we observed through experiments that the program execution time impact of this case is limited, especially when we filter L1 cache hits in the trace files. A more challenging problem occurs when assessing the impact of a long-latency event such as an L2 cache miss.

Suppose that we have an L2 cache miss and the memory access latency is L_{mem} cycles. Trace items, carrying one L2 cache access each, are generated with the assumption that they hit in the L2 cache. Therefore, the program execution time after processing a trace item that causes an L2 cache miss is the sum of the program execution time before the trace item, the cycles recorded in the trace item (measured during the trace generation time), and the L2 cache miss latency L_{mem}. In an out-of-order processor, however, the impact of the same L2 cache miss on the program execution time may be well less than L_{mem} cycles because any subsequent instructions not dependent on the L2 cache miss may make continuous progress while the miss is pending.

To help reduce timing error, we have devised and experimented with a technique we call *instruction permeability analysis*. The idea is to associate with each trace item extra timing information, Δ, to be used as an offset to compensate for any effect of dynamic instruction scheduling on an L2 cache miss (hence memory access) event. To calculate Δ before simulation, we analyze extra traces generated with different L1 cache miss latency assumptions. The goal of

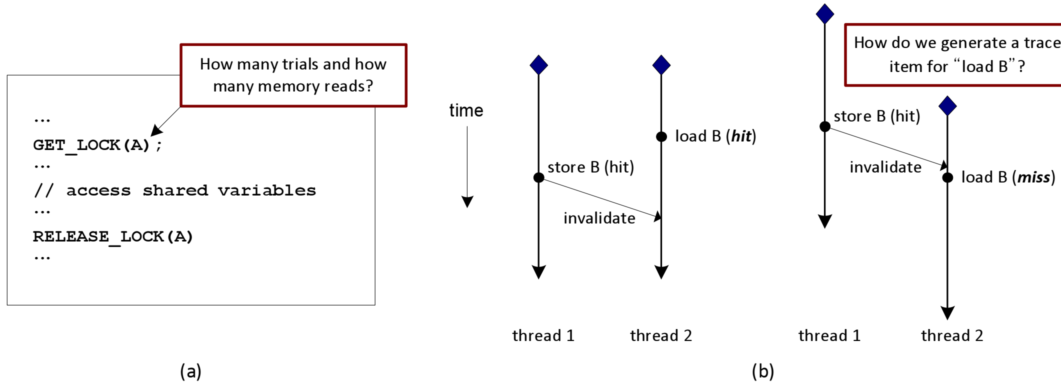

Figure 3. Problems with naïve trace generation for a shared memory parallel workload. (a) Semantics of synchronization primitives may be lost if only low-level memory accesses are considered at the trace generation time. (b) The different ordering of memory accesses caused by unpredictable progress of threads makes trace generation non-deterministic.

the instruction permeability analysis is to answer the question: "How would the program execution time change on an isolated L2 cache miss at position i in the trace?" By examining a relevant interval in the trace, we compute the value of Δ for trace item i.

Our preliminary test results shown in Figure 2, were obtained using two extra traces with alternating L2 cache access latencies. Except for *art* and *ammp*, the timing errors are now within 10%. In the case of *art*, especially when the Outorder-2 configuration is used, many trace items appear out of order in the traces generated for analysis, which hinders our ability to properly align and match trace items and extract Δ. Our result in Figure 2 demonstrates that the proposed instruction permeability analysis technique is quite effective in reducing timing errors. We expect to reduce timing errors further by fine-tuning our algorithm and leave this as a future work.

2.4. Handling a shared-memory multithreaded workload

As the instruction-level parallelism exploited in a superscalar processor poses challenges for accurate trace-driven simulations, the thread-level concurrency manifested in a multithreaded workload presents tricky issues.

The first issue we address is how to represent synchronization operations in TPTS. Figure 3(a) depicts this issue. To correctly simulate multiprocessor synchronization, the synchronization primitives must be recorded in trace files at a high level, such as Get-Lock and Barrier, rather than low-level instructions executed at the trace generation phase. This is to accurately model resource contention and thus dynamic interleaving of thread execution in the simulation phase with the desired machine configuration in the simulation phase, and not in the trace generation phase. Therefore, it is required that we instrument a parallel benchmark program to capture the synchronization primitives in the trace files, either at the source level or at the binary level. Given the practice of using high-level programming constructs such as PARMACS macros and OpenMP in popular parallel benchmarks such as SPLASH-2 [16] and SPEC-OMP [13], the source-level instrumentation is relatively simple and we used this approach. Among previous multiprocessor simulation schemes, MINT [15] uses the same approach. We note that simulating programs written with transactional memory primitives can be done similarly.

The second issue arises when we generate filtered traces. Consider Figure 3(b) where two memory operations from thread 1 and 2 are interleaved differently. The load instruction for location B in the first example hits in the L1 cache, while the same instruction in the second example does not due to a prior invalidation message received from thread 1. The example illustrates the non-determinism in filtered trace generation due to the unpredictable thread interleaving during simulations. In fact, any sampling-based simulation approaches are subject to the same problem [12, 17, 18].

There can be several strategies to address this issue. First, we can ignore the effect and simply generate filtered traces based on the information at the trace generation time. Second, we can turn off filtering in the code regions that may potentially access shared variables. Third, we may identify those memory references that possess non-determinism and skip filtering for their instances. Lastly, we can simply generate full traces. These strategies present a trade-off between cost (for trace analysis, generation, and simulation) and accuracy. In the second and third approach where filtering is selectively applied, the trace items generated for the shared variables are for L1 caches, rather than L2 caches. Hence, to determine if they will hit in the L1 caches, we need access to accurate L1 cache state during simulation. With careful trace file design, we can reconstruct the L1 cache state from the L1 miss and write-back information recorded in trace files.

We use the first and second approaches in this work. In the parallel workloads that we examine in this paper, the first approach did not incur a large timing error because the variation in execution time is quite limited between synchronization points (see Section 4). Since the third approach can achieve a high accuracy without unduly increasing the number of trace items, we leave exploring this approach as a future work.

3. tsim: a Prototype TPTS Simulator

tsim is a prototype manycore processor simulator based on the TPTS framework. *tsim* is capable of simulating a multiprogrammed workload (composed of independent threads) or a shared memory multithreaded workload (composed of data-sharing, synchronized threads). This section gives a progress report of our efforts on *tsim*.

3.1. Processor architecture

tsim models a tile-based homogeneous chip multiprocessor (CMP) with a 2D mesh interconnect, distributed shared L2 cache with invalidation-based coherence with MESI states, a distributed directory, and a configurable number of interleaved main memory controllers. See Figure 4(a) for an example of our multicore processor organization. Figure 4(b) further illustrates the tile organization of *tsim* where each tile consists of a processor core with private L1 instruction and data caches, a distributed shared L2 cache slice, a directory controller slice, and a router. Our tiled architecture is similar to the ones used in recent studies [6, 21].

Each L1 cache is private to a processor core, while physically distributed L2 caches form a logically shared cache by all the processor cores. Cache blocks are interleaved among the L2 cache slices based on their block addresses and so are directory entries among the tiles. Therefore, the physical address of a missed L1 access determines the target tile to which a request is routed. On an L2 cache miss, a memory access request is generated and sent to a main memory controller, again based on the physical address of the miss, similar to [7]. In our current implementation, both L1 and L2 caches are write-back and write-allocate.

We have implemented a cache coherence protocol modeled after that of the SGI Origin 2000 server at full lengths, as described in [11]. We chose the protocol for *tsim* because it is a proven, product-grade protocol, is based on a distributed directory organization like ours, and is documented well. Four protocol requests (Get-Shared, Get-Exclusive, Upgrade, and Writeback) and their acknowledgments are modeled, as well as nack responses and the retry mechanism.

3.2. Efficient event handling

tsim is an event-driven simulator. Events are added to a time-ordered priority queue, where time is measured in elapsed cycles. The main loop of our simulation reads the next pending event from the queue and executes the call-back function associated with that event. Each call-back function performs a designated simulation-related or microarchitecture modeling task, such as updating various data structures to track contention and resource usage, pending or resuming processes based on multiprocessor synchronization actions, recording statistics, or ending simulation. The argument passed to each call-back function is the event that triggered it. Each event consists of a time, a type, a processor number, a call-back function pointer, and a pointer that can be assigned to any data structure.

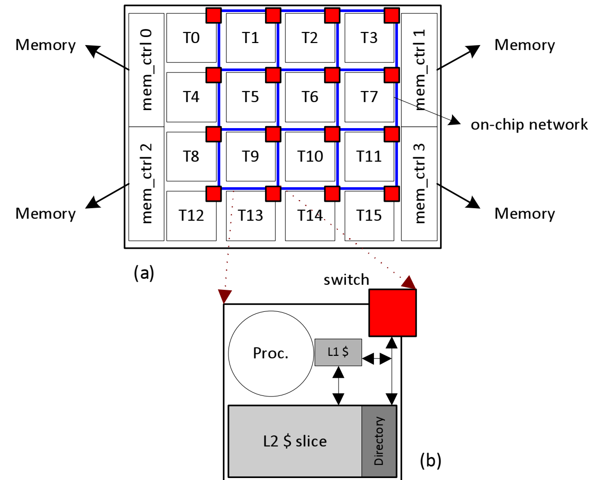

Figure 4. *tsim*'s processor model. (a) The tiled processor chip organization. (b) The tile organization.

Because the efficiency of handling frequent simulation events will determine the overall speed of *tsim*, we made efforts to streamline event handling and management. First of all, we do dynamically allocate/de-allocate memory in the event queue management. A pool of event structures are prepared at the simulation boot-up time so that each event allocation and de-allocation involves manipulating only a few variables. Similarly, because we will be accessing many trace items from our trace files, we buffer larger blocks of trace items in memory to reduce disk accesses.

Although it can vary depending on the processor configuration, our experiments suggest that the total number of pending events at any moment is quite limited–well under 1000, thus requiring a small, fixed memory space to store outstanding events. The small sizes of the event pool and the event queue guarantee that they will reside in the L2 cache of the host system.

3.3. Configurability

tsim provides high simulation flexibility by supporting a highly configurable processor model. A simulated processor can carry a configurable number of cores, on a 2D mesh network having adjustable dimensions. For instance, a 64-core processor can be arranged in an 8×8 network or a 16×4 network. Each processor core architecture is modeled in

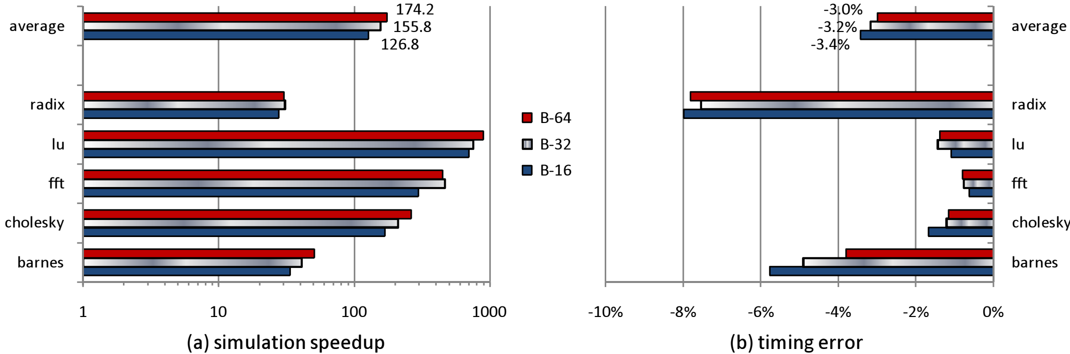

Figure 5. Simulation speedup of *tsim* over *refsim* and absolute timing error.

the timing-aware trace generator and can be configured to the extent the trace generator (*e.g.*, sim-outorder or Simics-based cycle-accurate simulator) allows.

Fast-forwarding and warm-up periods can be set for each processor core, fed with a separate trace file. Fast-forwarding is usually not needed because it can be handled prior to simulation in the trace generation phase. Initial interleaving of threads can be controlled by scheduling a thread execution trigger event for each thread at a (randomly) chosen cycle, with or without inter-thread coordination.

Caches are configured with the three traditional parameters: set associativity, block size, and cache size. Additionally, one can set the cache hit time and the replacement policy. We support a set of conventional replacement policies such as LRU, FIFO, and random. To support multiprocessor cache coherence, each cache block is associated with an extended block status field.

The on-chip interconnect can be configured with a few key parameters: input and output buffer sizes, connection width, routing capacity (*i.e.*, how many packets can be processed in a given cycle), and link/crossbar delays. The number and location of memory controllers and how memory blocks are mapped to memory controllers can be configured as well. The memory access latency is currently a fixed value (like sim-outorder). One may select to add a small random variation to the memory access latency, whose distribution is configurable.

In our current and future projects, we will implement a configurable request queue in the memory controller model (currently we have an infinite queue) as well as queue management policies to improve the memory access throughput, latency, and fairness.

4. Preliminary Evaluation

4.1. Evaluation setup

We use *tsim* and a detailed cycle-accurate execution-driven simulator built on Simics [8] for experiments. We call the execution-driven simulator "*refsim*" to denote "reference simulator," which is also adapted to generate traces for *tsim*. The two simulators were independently developed by two different groups of people. We use a 3.8 GHz Xeon-based Linux box (kernel 2.6.9) with 8 GB main memory for all experiments. Simulators were compiled with gcc (version 3.4.6) at the `O5` optimization level.

For workload, five programs from the SPLASH-2 benchmark suite [16] were employed: one application (*barnes*) and four kernels (*cholesky*, *fft*, *lu*, and *radix*). Synchronization primitives in the source files were instrumented with Simics magic instructions so that the trace generator could correctly capture and record synchronization trace items.

The baseline machine configuration (dubbed B-16) is a 16-tile (4×4) 2D mesh-based processor with 16 kB, 4-way L1 caches and a 512 kB, 16-way L2 cache bank in each tile. Cache blocks are 64 bytes. The L1 and L2 cache latencies are 2 and 8 cycles, respectively, and each network hop delay is 4 cycles. The memory latency is set to 300 cycles. We use a number of machine configurations with different architectural parameters: B-32 (32 kB L1 caches), B-64 (64 kB L1 caches), M-180 (memory latency is 180 cycles), M-240 (memory latency is 240 cycles), C-256 (L2 cache is 256 kB, 7 cycles), and C-128 (L2 cache is 128 kB, 6 cycles).

4.2. Results

Simulation speed: Figure 5(a) shows the simulation speedup of *tsim* relative to *refsim*, not considering the trace generation overhead (described below). *tsim* achieves an average speedup of 126.8, 155.8, and 174.2 for the configurations B-16, B-32, and B-64 respectively. In general, the speedup of *tsim* increases as the L1 cache size is increased. We expect this behavior because our filtered trace file contains only L1 misses; a larger L1 cache results in fewer L1 misses and thus fewer trace items to simulate.

Absolute timing error: The trade-off required to achieve our speedup is exemplified in *tsim*'s absolute error, defined as $(T_{tsim} - T_{refsim})/T_{refsim}$. Figure 5(b) shows the timing difference between *tsim* and *refsim* with configurations B-16, B-32, and B-64. *tsim* achieves a maximum absolute error of 8% with an average error of 3.4%, 3.2%, and 3% for the configuration B-16, B-32, and B-64 respectively.

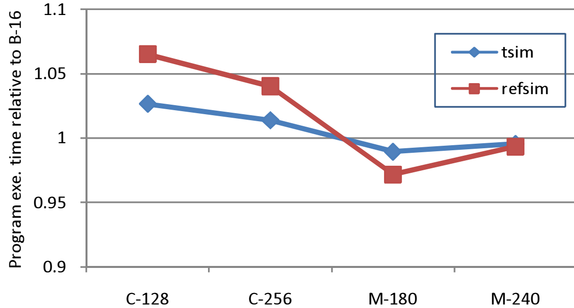

Figure 6. Averaged relative timing differences between *tsim* and *refsim* using all benchmark programs. The baseline configuration B-16 was used to normalize results.

The observed error is attributed to subtle implementation differences between the two simulators, especially in handling synchronization primitives and network contention.

Relative timing error: We study the capability of *tsim* to predict performance trends when architectural parameters are changed. Figure 6 shows the relative timing difference of *tsim* accurately follows the timing difference of *refsim* for various machine configurations. Ideally, the two lines in the plot should be nearly identical; however, subtle implementation differences in the two simulators (developed by two separate groups) used in our experiments result in a small error. We expect to reduce both absolute and relative errors by further tuning of our simulators.

Memory usage: *tsim* requires far less memory space to simulate a workload than an execution-driven simulator. In our current implementation, it uses about 8MB of memory space. Much of this memory space is used for buffering trace items and simulation events. The remaining space is consumed by modeling microarchitectural structures like caches, on-chip routers, and memory controllers.

Note that the memory requirement of *tsim* does not depend on the workload it simulates. However, a full-system simulator would require memory space for various kernel processes, the target application, and many more (intra-core) microarchitectural states. We measured the memory usage of *tsim* and *refsim* and our result shows that *tsim*'s memory space usage is only 7.8–9.8% of that of *refsim*.

Trace generation overhead: Because *tsim* is in essence a trace-driven simulator, there is an implicit overhead of generating traces. Typically, the overhead of generating traces is amortized over multiple simulations. For example, we used the same set of trace files to explore the five different configurations in the above experiment–B-16, M-180, M-240, C-256, and C-128. The timing-aware trace generation phase incurs a small ($<$10%) overhead to a regular cycle-accurate simulation. Hence, assuming that *tsim* achieves a 100$\times$ average simulation speedup, the overall simulation time improvement when running five simulations is $(1 \times 5)/(1.1 + 0.01 \times 5) = 4.35$. The improvement scales with the number of simulation runs using the same trace file, but savings are realized even with two simulation runs.

The storage space needed for trace files was also shown to be affordable, mainly due to trace filtering. For instance, each trace item typically corresponds to 30 to 300 instructions when L1 caches are 32 kB. In our current implementation, a set of trace files carrying a total of 100 billion instructions are between 8 to 80 GB in aggregate size. Our trace file design leaves room for improvement, however, given that standard compression methods (*e.g.*, gzip) give ∼80% file size reduction.

In summary, compared with *refsim*, *tsim* achieves 151$\times$ the simulation speed on average (not considering trace generation overhead), uses only 8.9% of memory space, and leads to 3.2% of timing errors on average for a suite of shared-memory parallel applications. When 16 threads were modeled, *tsim* achieved the simulation throughput of 146 MIPS (millions of simulated instructions per second).

5. Related Work

Trace-driven simulation methodology has long been an indispensable technique for analyzing computer performance [14, 20]. It decouples functional simulation from detailed simulation and typically achieves better simulation speeds than execution-driven simulation. It also allows the simulation of any program that might have been run on different platforms, given the correct tracing infrastructure. In previous and current practice, much trace-driven simulation work has focused on tracing memory references without timing [14] or using a full trace of executed instructions for fast simulation with complete fidelity [2]. We introduce the notion of event-centric trace filtering combined with timing-aware trace generation to speed up architecture simulation. The idea of full tracing and function/timing decomposition has been recently applied to FPGA-accelerated simulation methods [5].

One method of speeding up simulation is to use a smaller program input set, as seen in the MinneSPEC benchmark [10]. This smaller program attempts to maintain the real-world behavior. While in some cases MinneSPEC accurately represents the SPEC2k workload, there are occasionally "substantially different results with MinneSPEC than with the reference inputs," and it should thus be treated as a separate workload.

Another approach to shortening the simulation is to sample various points to reproduce overall program behavior. The SMARTS framework suggested by Wunderlich *et al.* [18, 19] uses this technique. The simulator will fast forward through the program until a simulation point is reached. It then warms up the machine state such as the cache or branch predictors, and finally runs detailed simulation for short period before fast forwarding to the next simulation point. Techniques researched for statistically choos-

ing sample points and determining warm up times can also be applied to our trace-driven approach. Rather than randomly sampling execution points, Sherwood *et al.* [12] have proposed to analyze the program beforehand to find one or more execution points that properly represent the total workload. Like statistical sampling, this technique can easily be used in conjunction with our trace-driven approach.

All sampling techniques require some sort of warm-up phase to bring the cache and other structures into a realistic state. Since the warm-up period for any sampling technique can be time-consuming, there has been research into mitigating this cost. The *memory timestamp record* [3] is a data structure that can store memory accesses, allowing for a shorter warm-up period. The use of checkpoints [17] has also been investigated, where instead of repeating warm-ups, important cache state is generated and saved beforehand, and the simulator need only load that information before it can accurately benchmark a program.

6. Conclusions

This paper proposed and evaluated the TPTS framework upon which very fast manycore processor architecture simulators can be built. The following summarizes our contributions and conclusions:

- The notion of Two-Phase Trace-driven Simulation (TPTS) is introduced. By not repeating uninteresting, yet time-consuming microarchitectural events, TPTS allows us to obtain fast simulation speeds while accurately modeling and simulating system-related aspects of a manycore processor.
- We identify and describe the potential drawbacks in the proposed approach–timing errors when modeling a dynamically scheduled processor and non-determinism during trace generation when shared-memory multithreaded applications are traced. We examine elaborate trace generation and annotation strategies to overcome these problems.
- We develop a multicore processor architecture simulator called *tsim*. It models a tile-based multicore processor having a two-level memory hierarchy, interleaved memory controllers, a directory-based coherence protocol, and a 2D-mesh network.
- Finally, we establish the effectiveness of the TPTS framework using *tsim*. Compared with a similarly configured full-system simulator, *tsim* achieved a sizable speedup while using a fraction of memory space. The absolute and relative errors are shown to be small; *tsim* correctly predicts the multicore performance trend as we change key architectural parameters.

In the future, we plan to further develop the instruction permeability analysis for accurate out-of-order processor simulation and employ more benchmark programs to evaluate the TPTS framework and its trade-offs.

Acknowledgment

This work was supported in part by NSF grant CCF-0702236 and an A. Richard Newton Graduate Scholarship from the 45th Design Automation Conf. (DAC), 2008. Authors thank the anonymous reviewers for their constructive comments.

References

[1] T. Austin *et al.* “SimpleScalar: An Infrastructure for Computer System Modeling,” *IEEE Computer*, 35(2):59–67, Feb. 2002.

[2] L. Barnes. “Performance Modeling and Analysis for AMD’s High Performance Microprocessors,” Keynote at *Int’l Symp. Perf. Analysis of Syst. and Softw. (ISPASS)*, Apr. 2007.

[3] K. C. Barr *et al.* “Accelerating Multiprocessor Simulation with a Memory Timestamp Record,” *Proc. Int’l Symp. Perf. Analysis of Syst. and Softw. (ISPASS)*, Apr. 2005.

[4] B. Black *et al.* “Can Trace-Driven Simulators Accurately Predict Superscalar Performance?” *Proc. Int’l Conf. Computer Design (ICCD)*, Oct. 1996.

[5] D. Chiou *et al.* “FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators,” *Proc. Int’l Symp. Microarch. (MICRO)*, Dec. 2007.

[6] S. Cho and L. Jin. “Managing Distributed, Shared L2 Caches through OS-Level Page Allocation,” *Proc. Int’l Symp. Microarch. (MICRO)*, Dec. 2006.

[7] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: A 32-Way Multithreaded Sparc Processor,” *IEEE Micro*, 25(2): 21–29, March-April 2005.

[8] P. S. Magnusson *et al.* “Simics: A Full System Simulation Platform,” *IEEE Computer*, 35(2):50–58, Feb. 2002.

[9] D. J. Lilja. *Measuring computer performance: A practitioner’s guide*, Cambridge University Press, 2000.

[10] A. KleinOsowski and D. J. Lilja. “MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research,” *Computer Arch. Letters (CAL)*, June 2002.

[11] M. Plakal *et al.* “Lamport Clocks: Verifying a Directory Cache-Coherence Protocol,” *Proc. Int’l. Symp. Parallel Algorithms and Architectures (SPAA)*, June-July 1998.

[12] T. Sherwood *et al.* “Automatically Characterizing Large Scale Program Behavior,” *Proc. Int’l Conf. Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[13] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[14] R. A. Uhlig and T. N. Mudge. “Trace-Driven Memory Simulation: A Survey,” *ACM Computing Surveys*, 29(2): 128–170, June 1997.

[15] J. E. Veenstra and R. J. Fowler. “MINT: A Front End for Efficient Simulation of Shared Memory Multiprocessor,” *Proc. Int’l Workshop Modeling, Analysis, and Simulation on Computer and Telecomm. Systems (MASCOTS)*, Jan. 1994.

[16] S. C. Woo *et al.* “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proc. Int’l Symp. Computer Arch. (ISCA)*, July 1995.

[17] T. F. Wenisch *et al.* “Simulation Sampling with Live-points,” *Proc. Int’l Symp. Perf. Analysis of Syst. and Softw. (ISPASS)*, Mar. 2006.

[18] R. E. Wunderlich *et al.* “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,” *Proc. Int’l. Symp. Computer Arch. (ISCA)*, June 2003.

[19] R. E. Wunderlich *et al.* “An Evaluation of Stratified Sampling of Microarchitecture Simulations,” *Proc. Workshop Duplicating, Deconstructing, and Debunking (WDDD)*, June 2004.

[20] J. J. Yi *et al.* “The Future of Simulation: A Field of Dreams,” *IEEE Computer*, 39(11):22–29, Nov. 2006.

[21] M. Zhang and K. Asanović. “Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors,” *Proc. Int’l Symp. Computer Arch. (ISCA)*, June 2005.