SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:239–258
Published online 21 January 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.956

Two-phase trace-driven simulation (TPTS): a fast multicore processor architecture simulation approach

Hyunjin Lee, Lei Jin, Kiyeon Lee, Socrates Demetriades, Michael Moeng and Sangyeun Cho*, †

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.

SUMMARY

Simulation is indispensable in computer architecture research. Researchers increasingly resort to detailed architecture simulators to identify performance bottlenecks, analyze interactions among different hardware and software components, and measure the impact of new design ideas on the system performance. However, the slow speed of conventional execution-driven architecture simulators is a serious impediment to obtaining desirable research productivity. This paper describes a novel fast multicore processor architecture 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 cycle-accurate simulation-based trace generation phase and can be omitted in the repeated trace-driven simulations. We report our experiences with *tsim*, an event-driven multicore processor architecture simulator that models detailed memory hierarchy, interconnect, and coherence protocol based on the TPTS framework. By applying aggressive event filtering, *tsim* achieves an impressive simulation speed of 146 millions of simulated instructions per second, when running 16-thread parallel applications. Copyright © 2010 John Wiley & Sons, Ltd.

Received 5 May 2009; Revised 29 September 2009; Accepted 6 October 2009

KEY WORDS: trace-driven simulation; computer architecture; multicore processor; performance evaluation

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 [1]. 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 [2]. Notably, among the 46 research papers presented at International Symposium on Computer Architecture (ISCA) 2007, as many as 37 papers relied on an execution- or trace-driven simulation technique‡.

*Correspondence to: Sangyeun Cho, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.
†E-mail: cho@cs.pitt.edu

‡Studies not using a simulation method include those employing a real machine design or hardware prototype and those that are by nature ‘novel’ or ‘non-traditional’ design papers.

Contract/grant sponsor: ACM Design Automation Conference (DAC) 2008

Copyright © 2010 John Wiley & Sons, Ltd.

Unfortunately, slow simulation speeds are a serious impediment to improving research productivity despite continuously increasing computer system performance [2–5]. For example, 1 ms of execution in real time can correspond to days of simulation time [3]. As such, simulating in detail the well-known SPEC2k CPU benchmark suite may take well over a month on a dedicated workstation [4]. The growing complexity in hardware designs, the need for using diverse and long-running real-world workloads, and the current design trend of multicore processor chips all work together to aggravate the situation even further.

This paper presents a novel simulation framework called *Two-Phase Trace-driven Simulation* (*TPTS*; pronounced 'tip-see'), upon which very fast multicore 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 [6], we effectively reuse the detailed timing simulation results collected in the cycle-accurate simulation-based trace generation phase. Therefore, with our approach, simulating many 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. Accordingly, the proposed approach is relevant for multicore research projects whose focuses are on the 'uncore' aspects of the design, such as the interconnection network and the last-level cache memory structures [7–13].

We have designed and implemented a prototype multicore processor simulator called *tsim* based on the proposed TPTS framework. *tsim* models a 2D mesh network, distributed shared L2 caches, and interleaved main memory controllers in detail. We present empirical results obtained from *tsim* that highlights the accuracy and speed of the TPTS approach. Compared with a custom simulator built on *Simics* [14] modeling a similar processor architecture, *tsim* achieves 151× the simulation speed, uses only 8.9% of memory space, and results in a 3.2% average timing error for a suite of shared-memory parallel applications. When 16 threads were run, *tsim* achieved the simulation throughput of 146.5 MIPS on a commodity Linux box.

The remainder of this paper is organized as follows. We summarize the related work in Section 2. Section 3 motivates our approach by presenting a study analyzing where time is spent in modern architecture simulation. In Section 4 we describe the concept of TPTS as a framework on which very fast multicore processor simulators can be built. This section also addresses some of the limitations in modeling sophisticated out-of-order processor cores and the challenges in handling multithreaded workloads that arise when using the trace-driven simulation approach. Section 5 describes our design and implementation of *tsim*, a prototype multicore processor simulator. We present the measured speed, accuracy, and other costs of *tsim* in Section 6. Finally, conclusions are presented in Section 7.

2. RELATED WORK

There is a large body of research work on various aspects of computer architecture simulation. Owing to space limitation, we mainly focus on two recent advances in high-speed execution-driven simulation techniques in this section: sampling-based simulation scope reduction and parallel multicore simulation. We will also discuss the previous studies on trace reduction and hardware-assisted simulation accelerators.

Recognizing the large speed gap between functional and detailed simulation, researchers have proposed sampling-based techniques to reduce the amount of detailed simulation by systematically choosing statistically significant program execution intervals [3–5, 15]. Sherwood *et al.* [15] presented an effective program phase detection and categorization method. Their method is applicable to simulation time reduction because given important program phases one can selectively

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

simulate portions of the program execution in detail while functionally simulating other intervals quickly. Instead of explicitly choosing program phases for detailed simulation, the SMARTS framework [3, 4] uses many short program intervals that are statistically random to the program execution. 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 a short period before fast forwarding to the next simulation point. One can control the simulation time and the confidence of the result by properly setting the sampling size and the sample period. Ekman and Stenström [5] further showed that the number of random simulation points can be reduced with the matched-pair comparison method. While sampling techniques work well for single-thread workloads, caution must be exercised when simulating multithreaded workloads because fine-grained sampling may inadvertently change the workload behavior and the simulation result. In principle, fast trace-driven simulation and sampling techniques are orthogonal; sampling techniques can be used during trace generation to reduce the amount of trace items. Also, within a simulation point, TPTSs trace filtering can effectively reduce the amount of simulation events (which are not interesting to the study).

Another method to speed up simulation is via parallelization. While architects can and do run simulations in parallel to evaluate different workloads and architectural parameters, it can be beneficial to speed up a single simulation via parallelization for quick evaluation of an idea. A natural approach to parallel simulation is to simulate each core with a thread. Chidester and George [16] and Monchiero *et al.* [17] use this approach, and observe that simulation speed depends greatly on the amount of synchronization generated between simulated threads. In addition, these simulators only synchronize cores when explicit synchronization events occur—such as barriers or locks. This adds inaccuracy because resource contention is not modeled on a cycle-by-cycle basis, especially if the number of simulated cores exceeds the number of host cores and some simulation threads must be run sequentially. Several works [18, 19] target cluster simulation and address this issue by keeping the cycle difference between processors under a certain time quantum. This quantum is ideally lower than the amount of time it would conceivably take for one processor to affect another (e.g. network latency + L2 access time). Slacksim [20] from Chen *et al.* adapts this technique for CMP simulation. It also allows for the time quantum to exceed the time for one core to affect another, since that is much lower for a CMP than in a networked cluster. They show that the SlackSim provides significant speedups and retains accuracy in comparison with cycle-by-cycle parallel simulation.

The trace-driven simulation methodology has long been a crucial technique for analyzing computer performance [2, 6]. In previous and current practice, much trace-driven simulation work has focused on tracing memory references without timing [6] or using a full trace of executed instructions for detailed out-of-order processor simulation with complete fidelity [21]. When tracing memory references, much work has been done on reducing the size of traces, particularly for memory system simulation. Sampling techniques for traces are very similar to sampling techniques for execution-driven simulation—sampling chunks of memory references at a time. They are also subject to a 'cold-start bias' [6] from an out-of-date cache, and require a similar warm-up period. Agarwal and Huffman [22] proposed sampling by memory addresses by grouping them into blocks—hence the name, *blocking*, to take advantage of spatial locality [22]. This would be akin to sampling basic blocks to simulate, similar to what has been done by Sherwood *et al.* [15]. An additional sampling method used with multiprocessor simulation is to run traces from a subset of nodes and preserve only those references affecting that subset [23]. Cache properties have also been used to reduce memory traces while retaining the total accuracy. Using the property of cache inclusion [23], a trace can be filtered of references guaranteed to hit in actual simulation. We utilize this property by filtering out L1 hits. Wang and Baer [24] use a direct-mapped 'filter cache' to filter memory references. Further work by Kaplan *et al.* [25] has yielded trace reduction techniques *Safely Allowed Drop* (*SAD*) and *Optimal LRU Reduction* (*OLR*), which accurately simulate the LRU policy (SAD also accurately simulates the OPT policy). These further filter out hits, and OLR is provably optimal for the LRU policy. However, in their initial proposal, these are not yet capable of simulating shared memory with invalidations.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Finally, with the rapid advances of the Field-Programmable Gate Array (FPGA) technology, researchers are looking into accelerating full-system simulation by splitting tasks within the simulation into FPGA and collaborating software modules [26–28]. UT-FAST [26], for example, partitions simulators into a speculative functional model component (software) that simulates the instruction set architecture and a timing model component (FPGA) that predicts the performance. They reported an average simulation speed of 1.2 MIPS, with a future goal of achieving 10 MIPS. The main benefits of a hardware-assisted approach are faster simulation speeds (compared with full-system simulation) and the ability to run an unmodified OS (with proper hardware and software support). While a hardware-assisted simulator is extremely useful for certain projects (e.g. running a new OS to study its scalability), an equivalently fast software-only simulation framework such as TPTS can be of much more need in practice at different performance evaluation stages [1, 2]. Specifically, a software simulation infrastructure is more accessible than a hardware-assisted simulator, is more flexible in terms of retargeting the simulated machine, is easier to debug and validate, and is equally relevant for many application-level performance evaluation projects.

3. AN ANATOMY OF SIMULATION TIME

In this section, we examine the time decomposition in detailed microarchitecture simulations. We use the SimpleScalar toolset (v4.0) [29] and the GEMS toolset [30] as test vehicles. We then present an intuitive discussion on multicore simulation time in order to motivate the proposed TPTS approach.

3.1. Simulating a program on modern simulators

3.1.1. What is the impact of detailed microarchitecture modeling? Table I presents the measured execution time of the benchmark *gcc* on different simulator platforms. We compare the simulation time of a fast yet relatively simple microarchitecture simulation platform (SimpleScalar) with that of a multicore-capable full-system simulation platform (Simics). We also compare the functional and detailed microarchitecture simulation time on both the platforms.

Functional simulation (sim-fast in SimpleScalar and bare Simics) slowed down the program execution time of *gcc* by a factor of 158 (sim-fast) and 437 (Simics) compared with native execution. 1 s of native program execution time is translated into approximately 2.5 and 7.5 min, respectively. Considering the full-system simulation capabilities of Simics, its speed is impressive—within $3\times$ of sim-fast, which does not model OS activities or intrinsic hardware features such as TLBs and I/O devices. Because a typical benchmark program's native execution time is relatively short (in the order of seconds to a few minutes), the slowdowns from functional simulation still lie within an acceptable range.

Table I. Simulation time of *gcc* in the SPEC2k benchmark suite [31] on a small sample input. We used a 3.8 GHz Xeon-based Linux box having an 8 GB main memory for all experiments.

Case	Time (s)	Ratio to *native*	Ratio to *functional*
native	1.054	1	—
sim-fast (*functional*)	167	158	1
sim-outorder	4247	4029	25
Simics (*functional*)	461	437	1
Simics w/ *Opal*	84 000	79 696	182
Simics w/ *Ruby*	41 245	39 131	89
Simics w/ *Ruby+Opal*	155 621	147 648	338

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

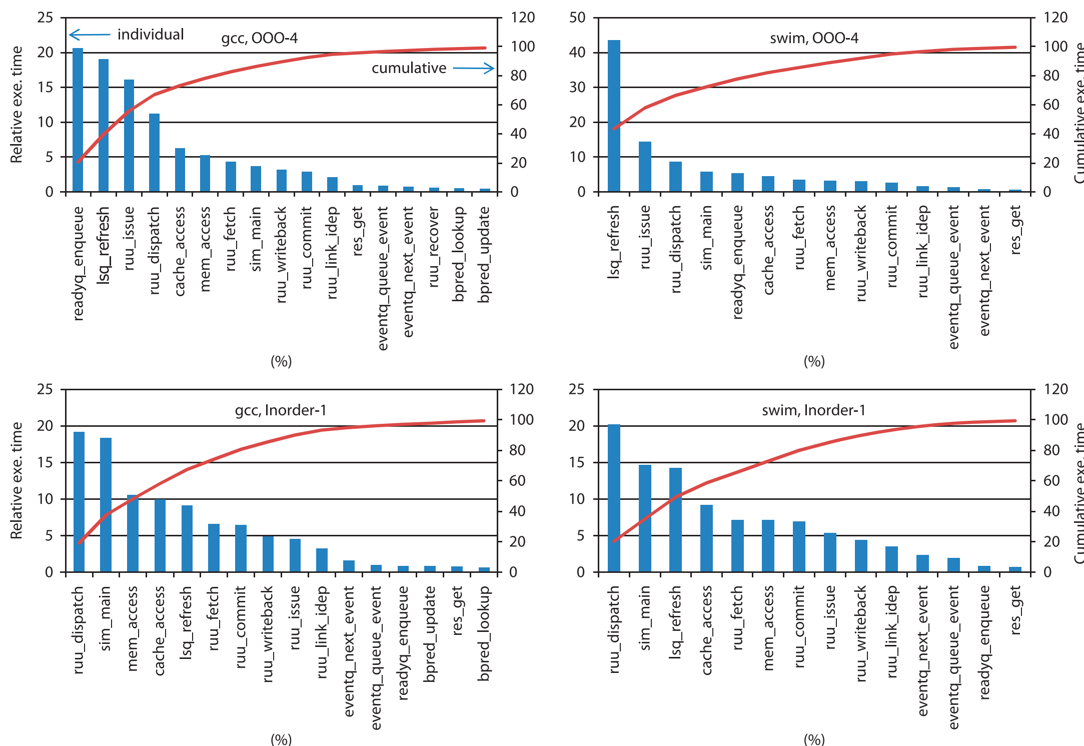

Figure 1. Pareto charts showing the amount of time spent in different functions in the sim-outorder simulator [29]. We simulated 2B instructions after fast forwarding of 500 M instructions. We used *gprof* to collect the results.

Table I further shows, however, that detailed microarchitecture simulation incurs a high overhead; the additional slowdown factor is 25 (sim-outorder) and 182 (Opal[§]) . In the case of Simics, employing both Opal and Ruby resulted in a combined slowdown of 338 compared with its own functional simulation[¶]. While the slowdown factor caused by detailed timing simulation is still less than the slowdown factor of functional simulation (from native execution), the combined slowdown factor now amounts to almost 155 000 with Simics. In this case, 1 s of native execution time will be translated into nearly two days of simulation time. This slowdown is considered excessive especially when an investigator is required to explore a large machine design space with many benchmark programs.

3.1.2. Where is time spent? Figure 1 presents the result of our first set of experiments, showing the decomposition of simulation time into simulator functions using two well-known benchmark programs, *gcc* and *swim* from the SPEC2k CPU benchmark [31]. We employ two machine configurations: ‘OOO-4’ is a 4-issue superscalar processor model having 128 reorder buffer (ROB) entries and 64 load store queue (LSQ) entries; and ‘Inorder-1’ is a single-issue in-order processor. OOO-4 simulations took 3 h (*gcc*) and 3.2 h (*swim*), whereas Inorder-1 simulations took 1.9 h and 1.6 h, respectively.

We make two key observations from the result. First, a large amount of simulation time is spent in modeling pipeline behavior and instruction scheduling; functions having the prefixes `readyq_*`,

[§]Opal and Ruby are part of the GEMS toolset [30] and model a detailed out-of-order processor and a (multiprocessor) memory hierarchy with a high degree of reconfigurability. We configure Ruby to model a single-core processor for the experiment. We use GEMS v1.3 in this work.

[¶]Table I shows that Simics has an intrinsic simulation time penalty when its microarchitecture simulation mode is turned on.

Copyright © 2010 John Wiley & Sons, Ltd.

`ruu_*`, `lsq_*`, and `eventq_*`. Their aggregate time corresponds to 83% (*gcc*) and 86% (*swim*) of the total execution time in the case of OOO-4 and 59% and 68% in the case of Inorder-1. In OOO-4, the time spent on actions to insert and choose ready instructions (`readyq_enqueue` and `lsq_refresh`) is particularly pronounced. In Inorder-1, instruction decoding and register update unit allocation (`ruu_dispatch`) and the simulator's main driver loop (`sim_main`) were big players, mainly because other routines related to complex instruction scheduling have much smaller roles in the simulation. As is suggested by the raw simulation time (over 3 h versus less than 2 h), OOO-4 generates many more events than Inorder-1 due to the dynamic instruction scheduling and bookkeeping.

Our second observation is that the overall time for modeling pipeline control actions or memory handling is similar in the two programs even though the contributions of individual routines are different. Over 80% of the simulation time is spent on pipeline control actions for both the programs in the case of OOO-4 and a little less than 70% in the case of Inorder-1. This is the major time consumer in both OOO-4 and Inorder-1 regardless of the benchmark program. Time for memory handling (`cache_access` and `mem_access`) is relatively limited, 12% (*gcc*) and 8% (*swim*) in the case of OOO-4 and 21% and 16% each in the case of Inorder-1. Interestingly, time for branch prediction is fairly limited in all measurements. This is because the branch prediction mechanism (table lookup and update) is relatively simple, invoked for branch instructions only, and does not involve inspecting other in-flight instructions for OOO execution.

In summary, detailed microarchitecture simulations are slower than native execution by three to five orders of magnitude, depending on the capability of a simulator, the complexity of the machine architecture modeled, and the degree of reconfigurability. The simulation time decomposition revealed that the processor implementation artifacts, such as pipelining and dynamic instruction scheduling, take much of the detailed microarchitecture simulation time.

3.2. Where will be simulation time spent in a multicore processor?

As we discussed previously, simulation time depends on the number and complexity of simulation events, which in turn are caused by the program execution on a particular processor architecture. Parallel events in real hardware are typically serialized and handled one-by-one in a software simulator. In a multicore simulator, events occur not only within processor cores, but also in various system components such as on-chip network, shared L2 caches, and synchronization mechanisms.

The time needed to simulate a program on a multicore processor can be decomposed as follows:

$$T_{\text{simulation}} = T_{\text{functional}} + T_{\text{timing}} \tag{1}$$

$$\begin{aligned} T_{\text{timing}} &= T_{\text{pipelining-artifacts}} + T_{\text{L1-access}} + T_{\text{L2-access}} \\ &\quad + T_{\text{mem-access}} + T_{\text{MP-sync}} \end{aligned} \tag{2}$$

Table I and Figure 1 show

$$T_{\text{native}} \ll T_{\text{functional}} \ll T_{\text{timing}}$$

and

$$\frac{T_{\text{pipelining-artifacts}}}{T_{\text{timing}} - T_{\text{pipelining-artifacts}}} \gg 1.$$

In other words, intra-core events, such as pipelining-related ones, will still account for a large fraction of multicore simulation time because chip-wide events are relatively infrequent.

The clocks per instruction (CPI) of a thread running on a single core in a multiprocessor can be similarly expressed as follows:

$$\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{overhead}} \tag{3}$$

$$\begin{aligned} \text{CPI}_{\text{overhead}} &= \text{CPI}_{\text{pipelining-artifacts}} + \text{CPI}_{\text{insufficient-ILP}} + \text{CPI}_{\text{L1-access}} \\ &\quad + \text{CPI}_{\text{L2-access}} + \text{CPI}_{\text{mem-access}} + \text{CPI}_{\text{MP-sync}} \end{aligned} \tag{4}$$

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

where $CPI_{overhead}$ is the amount of increase in CPI due to various inefficiencies introduced during program execution, such as pipelining artifacts (e.g. branch handling), memory accesses, and multiprocessor synchronization.

We observe that CPI overheads due to pipelining artifacts, insufficient ILP (within a single thread), and L1 cache read hits are to a large extent local to the specific processor core that a thread runs on. That is, even if we change the L2 cache configuration (e.g. size and latency) of a processor, their contributions will not change much. As such, new system design ideas for multicore architecture, such as new network topology and L2 caching strategies, would affect $CPI_{pipelining\text{-}artifacts}$ to a small extent. Accordingly, we will save much simulation time if we obtain the invariant time contributions of relatively uninteresting events once and do not perform simulation for such events repeatedly, motivating the proposed TPTS approach.

The TPTS framework tackles both $T_{functional}$ and T_{timing}. In T_{timing}, $T_{pipelining\text{-}artifacts}$ and $T_{L1\text{-}access}$ can be largely eliminated completely eliminating $T_{functional}$ after trace generation. By comparison, statistical sampling techniques reduce T_{timing} but not $T_{functional}$ [3, 4]. Recent checkpointing techniques attack $T_{functional}$ but not T_{timing} [32, 33]. Some multicore research papers in recent conferences (such as [7]) assume a simple in-order pipeline architecture to reduce $T_{pipelining\text{-}artifacts}$.

4. 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 2 shows the overall flow of TPTS.

4.1. 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 the 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 such as interconnection network and L2 caches in a multicore processor, 'fixing' certain intra-core structures such as L1 caches in the trace generation phase is acceptable.

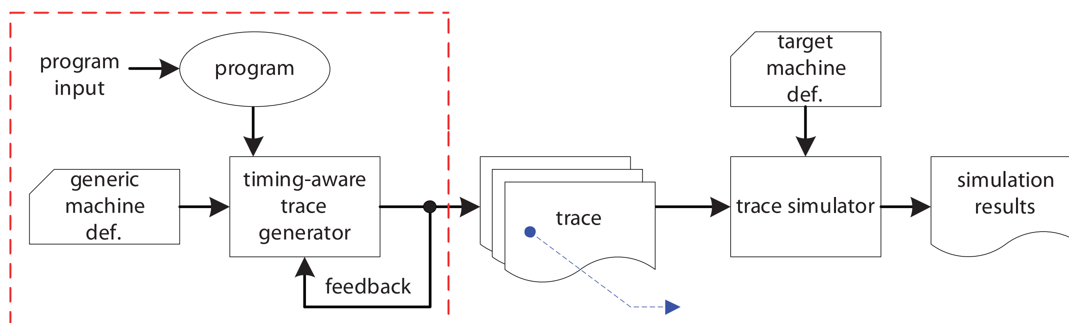

Figure 2. The proposed TPTS simulation flow. The dashed box encloses the trace generation phase.

Copyright © 2010 John Wiley & Sons, Ltd.

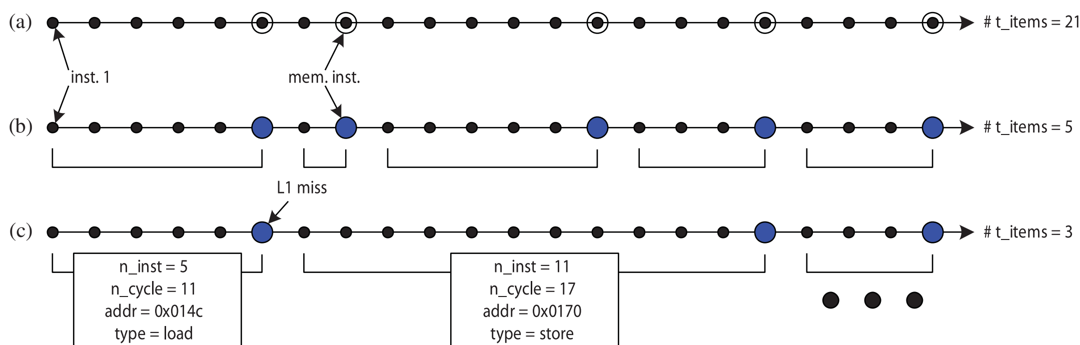

Figure 3. Different trace generation strategies: (a) Every instruction becomes a trace item; (b) every memory access becomes a trace item; and (c) only filtered memory accesses (e.g. L1 misses) become trace items. The method (a) has been used to faithfully model a complex processor while achieving simulation speedup [21, 34] and (b) is a conventional memory trace generation and simulation method [6]. TPTS uses strategies (b) and (c).

In this work, we focus on the traces containing all memory references (Figure 3(b)) and on filtered traces that contain only L1 misses (Figure 3(c)). 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. Depending on the simulated processor type, especially for a complex out-of-order processor, more information may be recorded in a trace item, such as instruction sequence number and dependency information (on a previous trace item) [35]. After this trace generation phase, the trace file will be fed into the trace-driven simulator. Trace files may be further analyzed or pre-processed before use.

4.2. Phase 2: trace simulation

Once traces are prepared, the trace-driven simulator models the target machine architecture using the traces. The 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 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 (or more, but not less). Hence, the purpose of the simulation is very important in determining the respective roles of the trace generation and simulation phase. It also limits the extent to which trace filtering techniques can be applied. For instance, if the simulation is focused on the L2 cache usage and simulates various techniques to reduce the L2 cache miss rate, an identical L1 cache configuration should be used or assumed in the two phases.

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 a separate trace item 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 [29] and our testbed simulator using selected SPEC2k CPU benchmark programs [31]. 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 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. Figure 4 reports the result. The timing differences come

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

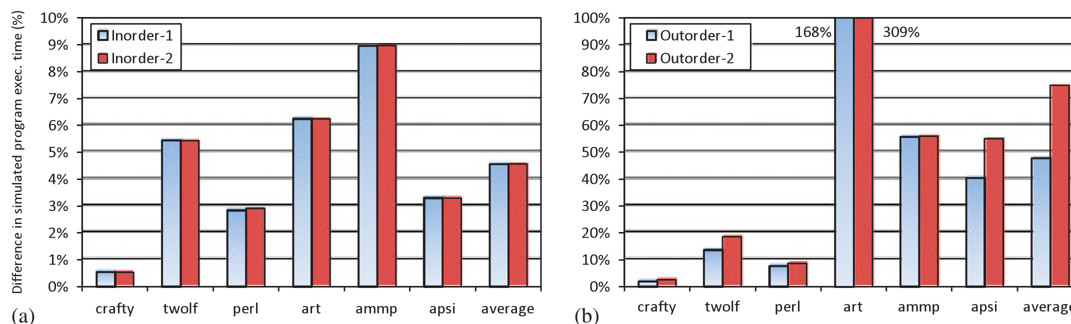

Figure 4. Simulated execution time difference: sim-outorder versus our testbed. Inorder-1/2 is a simple 1/2-issue in-order processor configuration. Outorder-1/2 is a 4-issue out-of-order processor configuration having a 32-/64-entry ROB. L1 caches are 16 kB and 4-way associative. All the configurations use a 4k-entry combined branch predictor. Results for filtered traces are shown: (a) in-order processor configurations and (b) out-of-order configurations.

from two major sources: difficulty in fully mimicking the sim-outorder simulator's timing-related behavior‖ and the difficulty of the trace-driven approach to model an out-of-order processor [34].

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 full, unfiltered traces (4.6% (filtered) versus 13.9% (unfiltered) for Inorder-1 and 49.2% (filtered) versus 83.2% (unfiltered) for Outorder-1); (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 [34]. In our experiment, simulating a superscalar processor in a naïve manner resulted in a large, unacceptable program execution time difference that reached 309.5%. Clearly, the 'traditional' trace-driven simulation method of simply blocking progress at each cache miss is extremely error-prone for out-of-order processor models. 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. Some of these issues are in fact relevant to any trace-driven simulation methodology and are not specific to the TPTS approach.

4.3. *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. 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 a dynamic effect is difficult to reproduce in a trace-driven simulator using static trace files. Interestingly, we observed through experiments that the program execution time impact of this case is fairly limited, especially when we filter

‖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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

L1 cache hits in the trace files. In fact, it turns out that a more challenging problem occurs when assessing the impact of a long-latency event such as an L2 cache miss [35].

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 that are not dependent on the L2 cache miss may make continuous progress while the miss is pending.

In this section, we will introduce two broad strategies to controlling timing error when modeling an out-of-order processor with trace-driven simulation. The first strategy is to assess the potential timing impact of an L2 cache miss at the trace generation time and associate with each trace item (i.e. potential L2 cache miss) the computed cache miss cost. At the simulation time, we determine each trace item to be an L2 cache hit or miss, and we use the pre-computed cache miss penalty in the case of a cache miss. To calculate the miss penalty of a trace item before simulation, we analyze extra traces generated with different L1 cache miss latency assumptions. For example, we could employ two extra traces with alternating L1 cache miss latencies (long and short and short and long, respectively). We call this process of computing the impact of an L2 cache miss on the program execution time *instruction permeability analysis* (IPA). With IPA, the actual cache miss penalty parameter used during simulation and the 'permeability' information associated with a particular trace item are considered together to derive the actual miss penalty [35].

The second strategy is to assess the impact of an L2 cache miss at the simulation time by dynamically reconstructing the processor state and evaluating how many trace items can still be processed during an L2 cache miss. This strategy does not require generating and analyzing extra traces as the first strategy does. Previous studies show that the reorder buffer (ROB) is the most important processor structure that governs the processor's instruction execution capability [35, 36], and we chose to implement the ROB in our testbed simulator. In essence, by keeping track of the number of empty slots in the ROB at any given time, we either continue processing more trace items or stop processing them until some preceding cache misses are resolved. We call this process as *ROB occupancy analysis* (ROA). ROA is an optimistic scheme compared with IPA in the sense that it allows multiple cache misses to be outstanding during simulation.

Our test results are shown in Figure 5. Except for *art* and *ammp*, the timing errors are now within 10% with IPA. 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 correct timing information. That resulted in a large timing error of 51% in *art* when the Outorder-2 configuration is used. While IPA helps reduce the timing error significantly, it falls short of completely eliminating errors. On the other hand, ROA is shown to provide much more accurate results robustly across the programs on both the machine configurations. Our result in Figure 5 demonstrates that relatively simple strategies such as IPA and ROA can successfully improve the accuracy of the trace-driven simulation methodology. More details about how to achieve high accuracy in modeling out-of-order processors using the trace-driven simulation methodology can be found in a recent work by Lee *et al.* [35].

4.4. 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 6(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 with the desired machine

Copyright © 2010 John Wiley & Sons, Ltd.

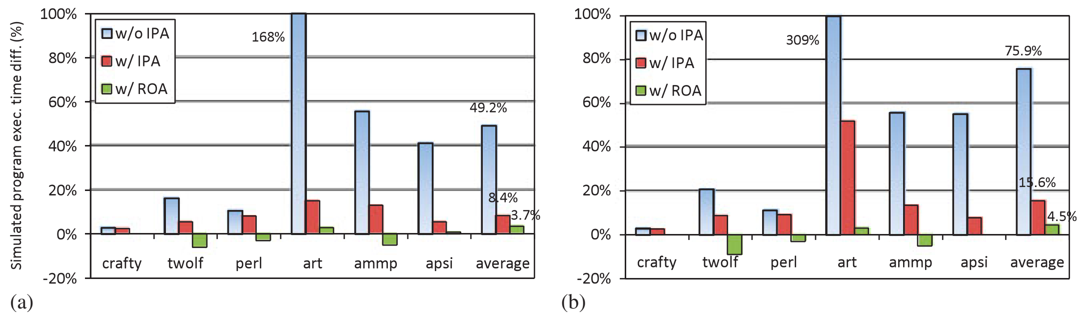

Figure 5. Simulated execution time difference between sim-outorder and our testbed—without and with instruction permeability analysis (IPA) and ROB occupancy analysis (ROA). Outorder-1/2 uses a 4-issue out-of-order processor configuration having a 32-/64-entry ROB. We use 16 kB, 4-way L1 caches, and a 4k-entry combined branch predictor. To compute averages we use the root-mean-square (RMS) method: (a) outorder-1-configuration and (b) outorder-2-configuration.

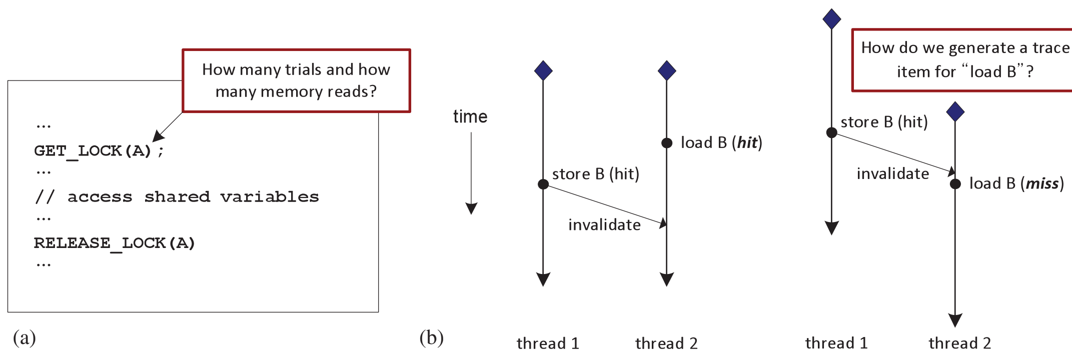

Figure 6. Problems with naïve trace generation for a shared-memory parallel workload: (a) Semantics of synchronization primitives may be lost if only low-level memory accesses are considered at the trace generation time and (b) the different ordering of memory accesses caused by unpredictable progress of threads makes trace generation non-deterministic.

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 [37] and SPEC-OMP [31], the source-level instrumentation is relatively simple and we used this approach. Among the previous multiprocessor simulation schemes, MINT [38] and Tango Lite [39] use the same approach. We note that simulating transactional memory programs can be done similarly.

The second issue arises when we generate filtered traces. Consider Figure 6(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 [3, 15, 33].

There are several strategies to address this issue. First, one can ignore the effect and simply generate filtered traces based on the information at the trace generation time. Second, one can turn off filtering in the code regions that may potentially access shared variables. Third, one may identify those memory references that possess non-determinism and skip filtering for their instances. Lastly, one can simply generate full traces. These strategies present a trade-off between

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

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 examine the first and second approaches in this work. In the parallel workloads that we study in this paper, the first, simplest approach did not incur a large timing error because the variation in the execution time is quite limited between synchronization points (see Section 6). As 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.

4.5. Synthesizing traces

The ability to directly synthesize workloads is often beneficial in detailed microarchitecture research. As the size of a multicore processor is continuously scaled, the balanced design of system resources, such as on-chip network routers and shared caches, will become increasingly more important. Closely understanding the behavior of such system resources is challenging, however, especially when conventional benchmark programs are used during simulation. Such a 'real' program is typically composed of a number of possibly alternating phases and does not consistently or fully exercise a particular system resource, even within a single phase. Summarized simulation results such as program execution time fail to clearly pinpoint and describe interesting intervals unless the program is extensively analyzed and instrumented beforehand [15]. It has been strongly suggested that using real programs reveals little or no insights for designing on-chip networks [40]. Consequently, some recent studies on multicore cache management resort to synthetic, micro-benchmarks [9, 41].

Because of its trace-driven nature, the TPTS framework makes the development of a synthetic workload handy. For example, it is trivial to schedule a system event targeting an arbitrary node at an arbitrary time. Events and the injection rate of them can be coordinated to create an 'impulse' of adjustable strength to the system to study its response. Memory sharing and access patterns leading to the maximum on-chip network traffic can be easily synthesized. Random, permutation, and tornado network traffic can be generated with ease as well. A side benefit is that the same trace synthesis approach can be used to validate and debug a simulator. Finally, we note that writing a micro-benchmark program in a high-level language to post an arbitrary system event at a precise timing is not as straightforward as the direct synthesis of traces.

4.6. Limitations

TPTS is versatile enough to be used in many studies that employ multiprogrammed workloads (comprising multiple independent single-threaded programs such as SPEC CPU benchmarks [31]) as well as shared-memory multithreaded workloads like SPLASH-2 [37]. For these workloads, we generate a trace file from each thread of execution and feed multiple trace files to the simulator. Large-scale workloads with many threads could be constructed in a straightforward manner by simply importing more trace files into the simulator. When multiple threads are used in an experiment, the simulator user must specify how the threads are interrelated. For example, addresses recorded in individual traces must be translated and mapped to a global 'simulated address space' so that shared-memory thread groups each see a common address space whereas other unrelated threads have their own address space.

On the other hand, TPTS is based on the trace-driven simulation methodology and inherits its limitations. Typically, a trace-driven simulator is event-driven and computes dynamically when the trace items recorded in the trace should be processed, in order to account for contentions in the modeled system. However, the content of the trace is static; it records a sequence of events that occurred in a specific program execution at the time of trace collection. If certain system-level events that happen during simulation change how instructions will be executed in the future in an unpredictable way, the static trace loses its fidelity. The static nature of the trace is the source of a few limitations to the use of the trace-driven simulation approach.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

We have previously discussed in Section 4.3 that modeling an out-of-order processor using the trace-driven simulation methodology is not straightforward. A long-latency memory access due to a L2 cache miss stalls dependent instructions while allowing others to slip and be executed during the memory access. Unable to distinguish between the dependent and independent instructions, a conventional trace-driven simulator does not accurately predict the performance of an out-of-order processor. While we presented our initial ideas about how to tackle this problem along with promising preliminary results in Section 4.3, we believe that more research is needed to fully solve this problem.

The effect of OS activities such as task scheduling and interrupt processing is not easily modeled using the trace-driven approach. This is especially true when the timings of program and system events differ during trace generation and simulation. For example, consider the server workloads, an important workload class for multicore processor architecture research. Thread execution in a server workload is typically triggered by an external event (e.g. a network packet arrives) that is asynchronous to the system state. Traditionally, tracing a program has focused on collecting information about the program execution itself but not such external events. Accordingly, tracing is done with a strong assumption about external events (e.g. an event occurs as soon as the previous event has been handled), which significantly reduces the chances of reusing the trace.

Additionally, task scheduling may occur at system calls. If a system call leads to a long latency, the OS may switch the processor to a different thread, which, with trace-driven simulation, is hard to model. To be able to signal when such an action is called for during simulation, one can create a trace item for major system calls such as `fread`, `fwrite`, `pid_block`, and `pid_unblock` [42]. During simulation, these special trace items will trigger switching and 'scheduling' of different traces. If the main tracing infrastructure is not capable of generating trace items in the kernel mode (which is often the case), separate tracing or characterization of OS actions may be required.

We feel that introducing the notion of trace scheduling and generating external or system events during simulation will significantly add to the capabilities of the proposed TPTS approach. Developing an efficient mechanism for managing interactive trace scheduling given a pool of (many) trace files is an interesting research direction in this regard.

5. TSIM: A PROTOTYPE TPTS SIMULATOR

This section describes *tsim*, a prototype multicore processor simulator we developed 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*, describing its features, discussing some of the design choices we made, and identifying a few extensions to be made in the future.

5.1. 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 7(a) for an example of our multicore processor organization. Figure 7(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 [7, 9]. The on-chip network and router architecture is modeled after an industry design [43].

Each L1 cache is private to a processor core, whereas physically distributed L2 caches form a logically shared cache by all the processor cores [7, 9]. 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,

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

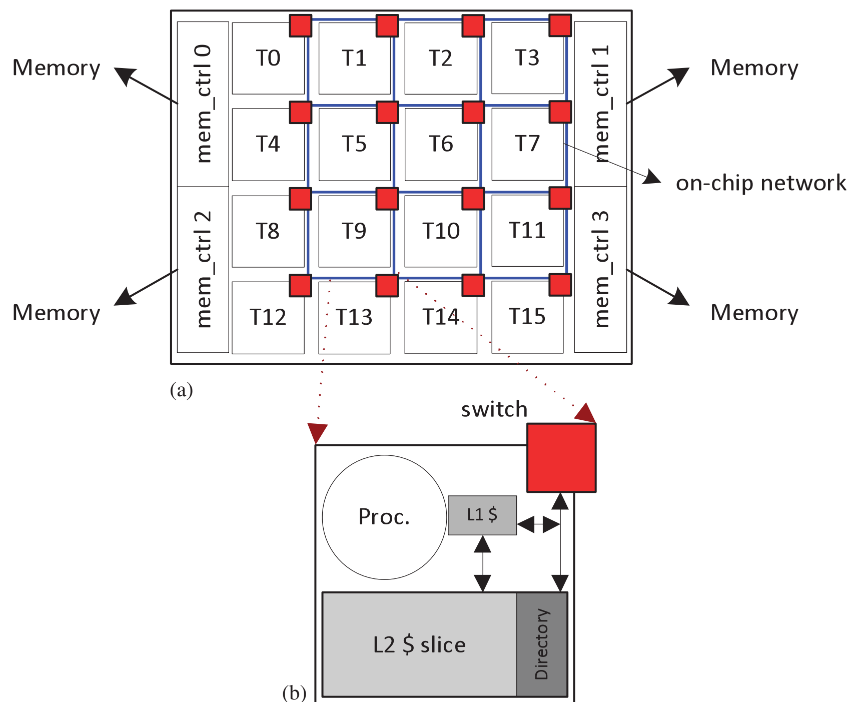

Figure 7. *tsim*'s processor model: (a) the tiled processor chip organization and (b) the tile organization.

again based on the physical address of the miss, similar to [44]. 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 [45]. 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. To increase the simulated machine's performance, we have optimized the original cache coherence protocol, especially when dealing with writes (and invalidations), based on a relaxed consistency model.

5.2. *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 associated with an event.

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 not use any dynamic memory allocation and de-allocation 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 pointers and 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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

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.

5.3. 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 aspect ratios. For instance, a 64-core processor can be arranged in an 8×8 network or in a 16×4 network. Each processor core architecture is modeled in 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.

6. QUANTITATIVE EVALUATION

6.1. Evaluation setup

We use *tsim* and a detailed cycle-accurate execution-driven simulator built on Simics [14] for experiments. We call the execution-driven simulator '*refsim*' to denote 'reference simulator,' which has also been adapted to generate traces for *tsim*. We consider *refsim* as an 'accurate' simulator which we compare the *tsim* simulator to. Moreover, *refsim* has been optimized for simulation efficiency as it has been used as the main research tool in other multicore architecture modeling projects. Both being configurable to a similar degree, the two simulators were developed by two different groups of people based on a common architecture model.

As workloads for experimental evaluation, five programs from the SPLASH-2 benchmark suite [37] were employed: one application (*barnes*) and four kernels (*cholesky*, *fft*, *lu*, and *radix*). While relatively old, the SPLASH-2 suite is frequently found in recent publications. 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. The L1 and L2 cache latencies are 2 and 8 cycles, respectively, and each network hop delay is 4 cycles. The main memory latency is set to 300 cycles. The processor core has a simple scalar pipeline. Traces for experiments were collected using *refsim*. We use a number of machine configurations with different architectural parameters than the baseline configuration: B-32 (32 kB L1 caches), B-64 (64 kB L1

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

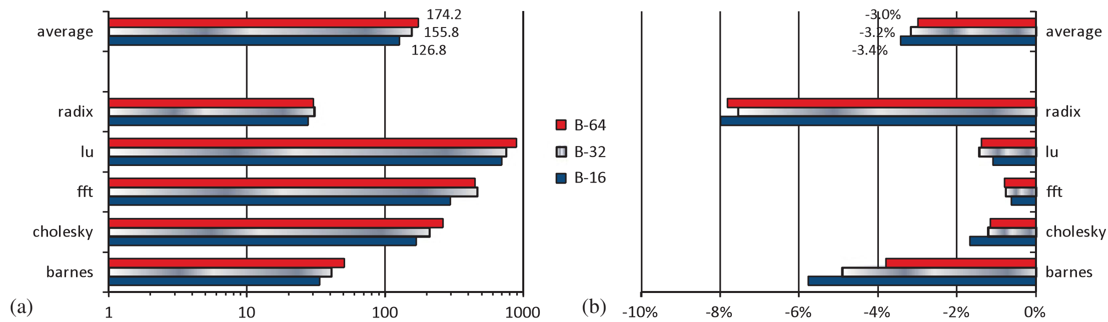

Figure 8. Simulation speedup of *tsim* over *refsim* and absolute timing error: (a) simulation-speedup and (b) timing error.

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). Note that each of these configurations has only one parameter different from the baseline configuration. This is to isolate potential error sources from each other when evaluating *tsim*.

6.2. Results

Simulation speed: Figure 8(a) shows the simulation speed of *tsim* relative to that of *refsim*. *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. Therefore, a larger L1 cache results in fewer L1 misses and thus fewer trace items to simulate. It is noted that we have used relatively small 16 kB L1 caches, not to favor our approach during evaluation.

Absolute timing error: The trade-off required to achieve our speedup is exemplified in *tsim*'s absolute error. Figure 8(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. We observe that the errors of *tsim* were negative (i.e. simulated execution time according to *tsim* is smaller than that of *refsim*) for all the benchmarks studied. We attribute this to the aggressive cache access filtering we employ in *tsim*. While coherence actions cause contentions and delays to the L1 cache operation in *refsim*, *tsim* does not model such artifacts faithfully as we apply L1 cache filtering. We leave addressing this issue as a future work. As with simulation speed, we expect that the absolute timing error will be improved (become smaller) with larger L1 caches.

Relative timing error: In many 'comparative' simulation studies, relative timing error is more important than absolute timing error. A reasonable simulator must correctly predict the performance change direction and the amount of change given modified simulation parameters. Figure 9 shows that 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 research 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 8 MB 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 such as caches, 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*, with an average of 8.9%.

Copyright © 2010 John Wiley & Sons, Ltd.

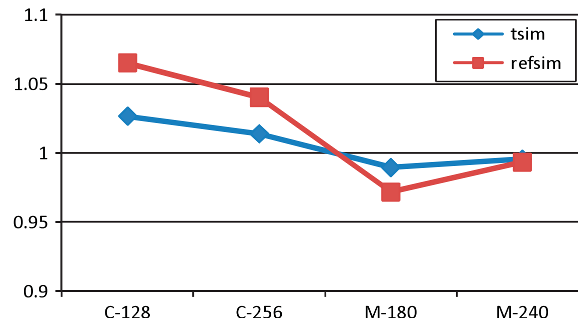

Figure 9. Averaged relative timing differences between *tsim* and *refsim* using all benchmark programs. The baseline configuration B-16 was used to normalize results.

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 $100\times$ simulation speedup, the overall simulation time improvement when running five simulations is at least $(1\times 5)/(1.1+0.01\times 5)=4.35$. The improvement scales with the number of simulation runs using the same trace file, but savings are realized even with only 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 and 80 GB in aggregate size. Our trace file design leaves room for improvement, however, given that standard compression methods (such as gzip) give $\sim$80% file size reduction.

Development effort: *refsim* and *tsim* were independently developed by two groups of researchers. They took roughly 12 man-months (*refsim*) and 8 man-months (*tsim*) each to develop. An important reason for the difference in development time was the ease of debugging. Unlike *refsim* which built on a commercial-grade, black-box simulation framework [14], *tsim* was developed entirely by the development team from scratch. Accordingly, its learning curve was more quickly accelerated. Moreover, due to its fast speed, *tsim* allowed the developers to quickly identify a bug once detected, by rapidly replaying the bug manifestation after proper instrumentation. In both cases, a large chunk of the development time was put into the design and implementation of the coherence protocol. In terms of code lengths, they have $\sim$8,700 lines (*refsim*) and $\sim$12,000 lines (*tsim*), respectively. *tsim* includes $\sim$4,000 lines of code for processing statistics and simulator options.

In summary, compared with *refsim*, *tsim* achieves $151\times$ the simulation speed on average, 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 146.5 MIPS (millions of simulated instructions per second) on a commodity Linux box. Our result is very encouraging; recent FPGA-based simulation accelerators have been reported to obtain 1.2 MIPS [26] to 63 MIPS [27].

7. CONCLUSIONS

This paper presented the TPTS framework upon which very fast multicore 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

simulation speeds while accurately modeling and simulating system-related aspects of a multicore processor. Using the TPTS approach, information gathered during the detailed timing-aware trace generation phase is reused in the repeated trace simulation phase.
- We identify and tackle 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 pose an interesting research problem: Modeling an out-of-order superscalar processor performance using the trace-driven simulation approach. Although a similar problem was previously considered difficult, our preliminary investigation identifies several promising directions.
- We design and implement 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. Its design approach and configuration options are described in detail.
- Finally, we establish the effectiveness of the TPTS framework using *tsim*. Compared with a similarly configured full-system simulator, *tsim* achieved an average speedup of 151 while using a fraction of memory space. The absolute and relative errors are shown to be small; *tsim* demonstrated that it can correctly predict the multicore performance trend as we change key architectural parameters.

As the current multicore processor design trends call for examining many system-wide design parameters and employing heavy workloads, a fast simulation framework such as TPTS is of growing interest. As a future work, we plan on tackling the problem of accurately modeling an out-of-order processor in the TPTS framework.

ACKNOWLEDGEMENTS

This work was supported in part by the A. Richard Newton Graduate Scholarship from the ACM Design Automation Conference (DAC) 2008.

REFERENCES

1. Lilja DJ. *Measuring Computer Performance*: *A Practitioner's Guide*. Cambridge University Press: Cambridge, 2000.
2. Yi JJ, Eeckhout L, Lilja DJ, Calder B, John LK, Smith JE. The future of simulation: A field of dreams. *IEEE Computer* 2006; **39**(11):22–29.
3. Wunderlich RE, Wenisch TF, Falsafi B, Hoe JC. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), San Diego, CA, U.S.A, June 2003; 84–95.
4. Wunderlich RE, Wenisch TF, Falsafi B, Hoe JC. An evaluation of stratified sampling of microarchitecture simulations. *Proceedings of the Workshop Duplicating*, *Deconstructing, and Debunking* (*WDDD*), held in conjunction with *International Symposium on Computer Architecture* (*ISCA*), München, Germany, June 2004.
5. Ekman M, Stenström P. Enhancing multiprocessor architecture simulation speed using matched-pair comparison, *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Austin, TX, U.S.A., March 2005; 89–99.
6. Uhlig RA, Mudge TN. Trace-driven memory simulation: A survey. *ACM Computing Surveys* 1997; **29**(2):128–170.
7. Zhang M, Asanović K. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), Madison, WI, U.S.A., June 2005; 336–345.
8. Chang J, Sohi GS. Cooperative caching for chip multiprocessors. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), Boston, MA, U.S.A., June 2006; 264–276.
9. Cho S, Jin L. Managing distributed, shared L2 caches through OS-level page allocation. *Proceedings of the International Symposium on Microarchitecture* (*MICRO*), Orlando, FL, U.S.A., December 2006; 455–465.
10. Kim S, Chandra D, Solihin Y. Fair Cache sharing and partitioning on a chip multiprocessor architecture. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), Antibes Juan-les-Pins, France, October 2004; 111–122.
11. Nesbit KJ, Aggarwal N, Laudon J, Smith JE. Fair queuing memory systems. *Proceedings of the International Symposium on Microarchitecture* (*MICRO*), Orlando, FL, U.S.A., December 2006; 208–222.
12. Cho S, Li T, Mutlu O. Interaction of many-core computer architecture and operating systems. *IEEE Micro* 2008; **28**(3):2–5.

Copyright © 2010 John Wiley & Sons, Ltd.

13. Iyer RR. CQOS: A framework for enabling QoS in shared Caches of CMP platforms. *Proceedings of the ACM/IEEE Conference on Supercomputing* (*ICS*), Saint-Malo, France, June 2004; 257–266.
14. Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Högberg J. Larsson F, Moestedt A, Werner B. Simics: A full system simulation platform. *IEEE Computer* 2002; **35**(2):50–58.
15. Sherwood T, Perelman E, Hamerly G, Calder B. Automatically characterizing large scale program behavior. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), San Jose, CA, U.S.A., October 2002; 45–57.
16. Chidester MC, George AD. Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modeling and Computer Simulation* (*TOMACS*) 2002; **12**(3):176–200.
17. Monchiero M, Ahn J-H, Falcon A, Ortega D, Faraboschi P. How to simulate 1000 cores. *Proceedings of the International Workshop on Design*, *Architecture*, *and Simulation of Chip Multi-Processors* (*dasCMP*), Lake Como, Italy, November 2008.
18. Mukherjee SS, Reinhardt SK, Falsafi B, Litzkow M, Huss-Lederman S, Hill MD, Wood DA. Wisconsin wind tunnel II: A fast and portable parallel architecture simulator. *Proceedings of the Workshop Performance Analysis and its Impact on Design* (*PAID*), Denver, CO, U.S.A., June 1997.
19. Falcón A, Faraboschi P, Ortega D. An adaptive synchronization technique for parallel simulation of networked clusters. *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Austin, TX, U.S.A., April 2008; 22–31.
20. Chen J, Annavaram M, Dubois M. SlackSim: A platform for parallel simulations of CMPs on CMPs. *Proceedings of the International Workshop on Design*, *Architecture*, *and Simulation of Chip Multi-Processors* (*dasCMP*), Lake Como, Italy, November 2008.
21. Barnes L. Performance modeling and analysis for AMD's high performance microprocessors. Keynote at *International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), San Jose, CA, U.S.A., April 2007.
22. Agarwal A, Huffman M. Blocking: Exploiting spatial locality for trace compaction. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS*), Boulder, CO, U.S.A., 1990; 48–57.
23. Chame J, Dubois M. Cache inclusion and processor sampling in multiprocessor simulations. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS*), Santa Clara, CA, U.S.A., 1993; 36–47.
24. Wang W-H, Baer J-L. Efficient trace-driven simulation methods for cache performance analysis. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS*), Boulder, CO, U.S.A., 1990; 27–36.
25. Kaplan SF, Smaragdakis Y, Wilson PR. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation* (*TOMACS*) 2003; **13**(1):1–38.
26. Chiou D, Sunwoo D, Kim J, Patil NA, Reinhart W, Johnson DE, Keefe J, Angepat H. FPGA-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. *Proceedings of the International Symposium on Microarchitecture* (*MICRO*), Chicago, IL, U.S.A., December 2007; 455–465.
27. Chung ES, Nurvitadhi E, Hoe JC, Falsafi B, Mai K. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays* (*FPGA*), Monterey, CA, U.S.A., February 2008; 77–86.
28. Penry DA, Fay D, Hodgdon D, Wells R, Schelle G, August DI, Connors D. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. *Proceedings of the International Symposium on High-Performance Computer Architecture* (*HPCA*), Austin, TX, U.S.A., February 2006; 27–38.
29. Austin T, Larson E, Ernst D. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer* 2002; **35**(2):59–67.
30. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News* (*CAN*), September 2005; 92–99.
31. Standard Performance Evaluation Corporation. Available at: http://www.specbench.org [1 March 2009].
32. Barr KC, Pan H, Zhang M, Asanovíc K. Accelerating multiprocessor simulation with a memory timestamp record. *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Austin, TX, U.S.A., April 2005; 66–77.
33. Wenisch TF, Wunderlich RE, Falsafi B, Hoe JC. Simulation sampling with live-points. *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Austin, TX, U.S.A., March 2006; 2–12.
34. Black B, Huang AS, Lipasti MH, Shen JP. Can trace-driven simulators accurately predict superscalar performance? *Proceedings of the International Conference on Computer Design* (*ICCD*), Austin, TX, U.S.A., October 1996; 478–485.
35. Lee K, Evans S, Cho S. Accurately approximating superscalar processor performance from traces. *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Boston, MA, U.S.A., April 2009.
36. Karkhanis T, Smith JE. The first-order superscalar processor model. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), München, Germany, June 2004; 338–349.

Copyright © 2010 John Wiley & Sons, Ltd.

37. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 programs: Characterization and methodological considerations. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), Santa Margherita Ligure, Italy, July 1995; 24–36.
38. Veenstra JE, Fowler RJ. MINT: A front end for efficient simulation of shared memory multiprocessor. *Proceedings of the International Workshop on Modeling*, *Analysis*, *and Simulation on Computer and Telecommunication Systems* (*MASCOTS*), Durham, NC, U.S.A., January 1994; 201–207.
39. Goldschmidt SR. Simulation of multiprocessors: Accuracy and performance. *PhD Thesis*, Stanford University, 1993.
40. Dally B. Interconnect-centric computing. Keynote at *International Symposium on High-Performance Computer Architecture* (*HPCA*), Phoenix, AZ, U.S.A., February 2007.
41. Guo F, Solihin Y, Zhao L, Iyer R. A framework for providing quality of service in chip multi-processors. *Proceedings of the International Symposium on Microarchitecture* (*MICRO*), Chicago, IL, U.S.A., December 2007; 343–355.
42. Lo JL, Barroso LA, Eggers SJ, Gharachorloo K, Levy HM, Parekh SS. An analysis of database workload performance on simultaneous multithreaded processors. *Proceedings of the International Symposium on Computer Architecture* (*ISCA*), Barcelona, Spain, June 1998; 39–50.
43. Vangal S, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, Finan D, Iyer P, Singh A, Jacob T, Jain S, Venkataraman S, Hoskote Y, Borkar N. An 80-tile 1.28TFLOPS network-on-chip in 65 nm CMOS. *Proceedings of the IEEE International Solid-State Circuits Conference* (*ISSCC*), San Francisco, CA, U.S.A., February 2007; 98–99, 589.
44. Kongetira P, Aingaran K, Olukotun K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro* 2005; **25**(2):21–29.
45. Plakal M, Sorin DJ, Condon AE, Hill MD. Lamport clocks: Verifying a directory cache-coherence protocol. *Proceedings of the International Symposium on Parallel Algorithms and Architectures* (*SPAA*), Puerto Vallarta, Mexico, June–July 1998; 67–76.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe