FastMP: A Multi-core Simulation Methodology

Shobhit Kanaujia Irma Esmer Papazian Jeff Chamberlain Jeff Baxter
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052
{shobhit.o.kanaujia, irma.esmer, jeffrey.d.chamberlain, jeff.baxter}@intel.com

Abstract

Current architecture trends focus on designs that exploit thread-level parallelism using multiple cores on chip [15], [16]. With increasing number of cores, the simulation run time increases accordingly with best-case linear scaling. These large turnaround times prohibitively limit the ability to evaluate performance tradeoffs during the design phase.

In this paper, we propose a multi-core simulation methodology aimed specifically at addressing runtime scalability. We use SPECrate, a commonly used throughput metric for multi-processor evaluation as our test case, but expect the methodology is applicable to performance simulations for general class of homogeneous multi-threaded workloads that do not share data. The approach is to simulate a subset of cores in detail and use real-time analysis of the detailed cores' behavior as the basis for approximating the memory traffic of other cores. We provide a detailed evaluation of FastMP by measuring the simulation speedup and measurement error compared against fully detailed simulation of all cores for core counts of 2, 4 and 8. We show results for every workload in the SPEC CPU 2000 suite. Our methodology introduces reasonable errors and obtains average runtime speedups of 1.9, 3.1 and 5.9 for 2, 4 and 8 core simulations respectively.

Keywords
Simulation, Architecture, SPECrate, Multi-core, Multi-processor.

1. Introduction

Transistor densities have increased over time in accordance with Moore's law. Architects have exploited this with increasing support for larger instruction windows. Additionally clock frequencies have consistently increased over product generations. However, these trends have reached bottlenecks such as power dissipation, design complexity, and diminishing returns from increasing ILP support. This has led to the recent industry-wide trend of multi-core designs [15], [16]. A multi-core design typically contains several cores on a single chip, which share the memory infrastructure, and possibly portions of the cache hierarchy. Figure 1 illustrates two possible configurations for multi-core architectures with the resources shared by the different cores highlighted.

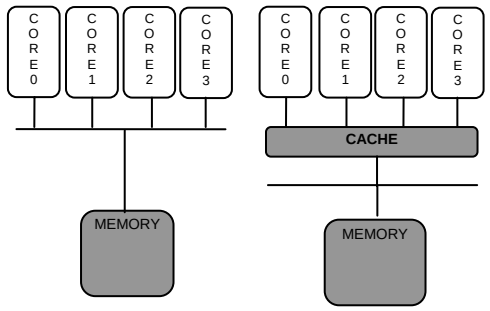

Figure 1. Two possible configurations for multi-core architectures.

Detailed simulation of multi-core architectures consists of multiple simultaneously active threads of execution (one per context). Accurate simulation is a commonly used technique for evaluation of computer architectures and can provide design insights before the hardware is available. However, with higher core counts, the simulation times increase due to increase in the simulation state and code space. Additionally there can be a decrease in the progress rate of each thread due to interference from sharing of resources, thus increasing the total simulated cycles. Figure 2 shows the scaling of average simulation times (for SPECrate) with increasing core counts using a conventional sequential simulator model. With typical design space exploration requiring multiple simulation passes, such

long simulation times clearly pose a problem going ahead.

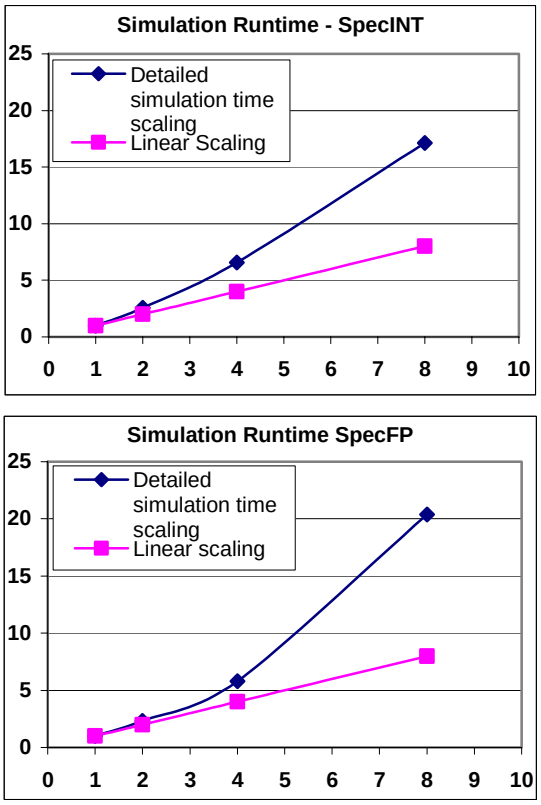

Figure 2. Simulation run time scaling with increasing core counts.

1.1. Experimental setup

We conduct all our experiments using an execution driven cycle accurate proprietary simulator that models a pipeline like the processor productized as the Intel® Core™ Solo Processor [11] on 65 nm Process and the more recent Intel® Core™ Microarchitecture [12]. We simulate a multi-core model where the cores have independent caches and only share the memory path. Table 1 lists the important parameters of our simulator configuration. This hypothetical configuration is comparable to contemporary designs. The simulator is layered on top of an architectural simulator that executes "Long Instruction Trace (LIT)"s. Unlike the name suggests a LIT is not a trace. It is a snapshot of processor architectural state that includes the state of system memory. Included in the LIT is a list of "LIT injections" which are system interrupts that are needed to simulate events like DMA. Our experiments use multiple 30 million instruction long LITs for each benchmark. These LITs are created after careful analysis and are assigned weights to accurately represent the overall application behavior. Each LIT is accompanied by a separate warmup file, which contains memory transactions and is used for warming up the caches before detailed simulation. We evaluate FastMP using the entire SPEC CPU suite. In all, there are total 26 SPEC-CPU benchmarks comprised of 415 LITs.

Table 1. Simulator configuration.

Core	3 GHz, 4-issue machine, 96 ROB entries, Intel® Core™ Microarchitecture branch predictor.
Per Core Cache hierarchy	Separate first level Instruction and Data Caches: 32KB, 8-way, 64-byte line, 1 cycle latency Unified second level : 8MB, 16-way, 64-byte line, 8 cycle latency Multi-level adaptive prefetchers
Memory	Bandwidth: 9GBps, 122nsec idle latency.

1.2. Organization of this paper

This paper begins by discussing previous work related to speeding up architectural simulation in section 2. Then, in section 3, the paper discusses the details of the FastMP methodology, provides simulation results, and compares them to fully detailed results for accuracy and speedup. Section 4 describes a feedback problem with the basic FastMP mechanism. It proposes an adaptive scheme for traffic injection, which reduces the effect of the feedback problem. We conclude the paper in section 5 with a summary of our work and future directions.

2. Related Work

There are several suggested techniques, which are alternatives to complete detailed simulations and aimed at decreasing the overall simulation time. Here we provide a summary of some of the previous related work.

One category of approaches applies statistical sampling to reduce the total duration of detailed simulation. The underlying premise is to extrapolate the characteristics of the population from a sampled subset. Wunderlich et al [6], and Conte et al [7], are examples of this approach, which do sampling dynamically during the execution. SimPoints [5] suggests a slightly different sampling approach which determines the simulation points "off-line". Simpoints

does application profiling and hotspot analysis, to identify representative phases. For final simulation, only those phases are simulated and their weighted performance is used to calculate the performance of the application. SimPoints is similar to our approach of choosing representative LITs, which provide the starting point for our simulations. However, even after choosing LITs, the overall simulation execution times per context are non-trivial, and they start exploding as the number of processors to be simulated simultaneously increases. Another approach is to use a reduced input set. MinneSpec [8] and the SPEC train and test inputs are examples of this approach.

There have also been studies which specifically focus on speeding up multi-processor simulation [1], [2], [3], [4]. These approaches parallelize the simulator to exploit the inherent parallelism when simulating multiprocessors. In some cases, the parallel simulation involves direct execution of the operations on native hardware; however, the simulator still executes any operations unavailable on the host. Chidister et al. [4] demonstrate a parallel multiprocessor simulator using SimpleScalar for each thread. Similarly Barr et al [3] modify the ASIM infrastructure to make it parallel. Mukherjee et al [2] identify key simulator operations and try to minimize their dependence on the host. Penry et al. [1] focus on automating simulator parallelization and integrating the hardware into the CMP structural models.

A more recent approach is to speed up the simulator by using FPGA based hardware software co-simulation. Chiou et al [13] partition the simulator into timing and functional models and implement the timing model in the hardware. Dave et al [14] implement both the timing and functional model in a tightly coupled fashion on an FPGA. Using FPGAs for speeding up simulation is a promising approach although a more exhaustive evaluation using full system models and multiple workloads is required.

Our approach is orthogonal to most of the approaches summarized here. Our methodology assumes as its starting point an arbitrary MP-capable simulation infrastructure that has some amount of wall-clock time to execute a given number of instructions. We assume that for any simulation infrastructure, and any given set of inputs to that simulator, unless you scale down the number of instructions simulated per thread as you add threads (or you have done an excellent job of parallelizing the simulator), the runtime is going to scale poorly (at best linear) as you add cores to the environment. What we are proposing is a methodology that takes a fixed simulator and a fixed set of inputs and seeks to improve this per-core runtime scaling problem with reasonable error margins without reducing the input coverage.

3. FastMP Methodology

3.1. General Description

The essential idea of the FastMP methodology is to take advantage of the redundancy inherent in a homogeneous workload like SPECrate to reduce the amount of computational work required by the simulator. For the purposes of this paper, we focus exclusively on the SPECrate suite, but we consider SPECrate to be a proxy for the general class of homogeneous workloads that do not share data. We also expect the methodology can be extended to speed up simulation of workloads that do share data so long as they are homogeneous (e.g., TPC-C, SAP), but we leave that as a topic for future study.

Before discussing the FastMP methodology in detail, we discuss a typical setup for a fully detailed SPECrate simulation. As mentioned previously, our starting point is a collection of LIT traces for each workload with associated weighting factors that previous analysis has determined to be representative of the workload as a whole. For a uniprocessor SPEC score, the simulator produces a CPI for each trace and then constant scaling factors are applied to the weighted average of these CPIs to produce a SPEC score for each workload. To generate an N-user SPECrate score, the mathematical approach is identical to the UP case with the exception of the scaling constant that converts the runtime of an N-user run to a SPECrate score (this constant is defined by the SPECrate methodology). The simulator's role for the N-user case is the same as the UP case: the simulator generates a CPI number for each trace. The difference, of course, is that for N-user scores the CPI we want is the CPI for each trace when run in parallel with N-1 additional copies.

The fully detailed approach to generating N-user SPECrate score is to run an N-core simulation in which the same trace is running simultaneously on each core. Offsets are applied to ensure that each simulated thread is starting at a different point in the code segment when the simulation begins. The intent is to approximate those offsets that will inevitably occur in a real system. The actual offsets that occur on a real system will depend on both the architecture and the application and will vary even on a fixed system from one run to the next because of non-repeatable system events and interrupts. A study of those details is beyond the scope of this paper. We used offsets in the range of 100k to 1M instructions per core, but the FastMP methodology that we propose

in this paper is independent of that choice and a different approach to dealing with offsets could easily be adapted to work with the FastMP technique. For all our experiments in this paper, we use a 100k instruction offset between adjacent cores.

Once offsets are chosen and simulations are run ideally one would average the CPIs that each core generates and take that as the N-user CPI for the trace. Our studies have shown that the SPECrate workloads are sufficiently homogeneous that the CPI of a single core is close enough to the average CPI that there is no need to take the average to get an accurate CPI estimate. In Figure 3 we calculate the delta between the CPI of the first core and the average CPI for all the cores in a 4-core system, then average the absolute value of those deltas across our entire suite of SPEC traces.

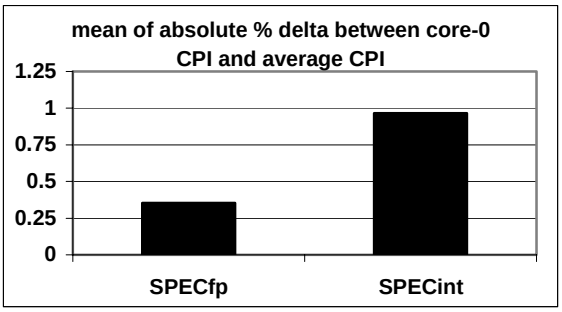

Figure 3. Comparison between first core and the average CPI.

This observation is the inspiration for FastMP: if the workload is sufficiently homogeneous that we only require the detailed CPI for one of the cores for determining the aggregate CPI, then it may not be necessary to simulate other cores to the same level of detail to produce a performance estimate. In this paper, we show that for homogeneous workloads that do not share data (like SPECrate), fully detailed simulation of all N cores is not necessary. The FastMP methodology that we propose provides a mechanism for stubbing out some of those cores without significantly affecting the CPI number that the simulator produces.

For the remainder of the paper, the cores that are not simulated in full will be referred to as fast-cores and the cores that are simulated in full will be referred to as detailed-cores. Not having to do detailed simulation of all of the cores provides savings in two ways. First, it reduces the data footprint of the simulator because the internal architectural state of the fast-cores is not stored. Second, it reduces runtime because no time is spent modeling the state transitions for the fast-cores. We propose a transaction module to replace the fast-cores. The transaction module records detailed-core memory transaction information such as address, transaction-type, and time delta with respect to previous transaction. This in-memory trace of the detailed cores' load on the system is used as a driver for approximating the load generated by the fast-cores. The capture and injection of transactions happen before the point where the cores begin sharing resources. Figure 4 shows the case where there is a single detailed core driving three fast-cores. The transaction module is essentially a set of circular buffers called trace buffers (one for each detailed core) where the detailed-cores are the producers and the fast-cores are the consumers. The head-pointer points to the location for recording the detailed core's next transaction. There is a tail-pointer for each of the fast-cores, which points to the next transaction to be injected for that core. For large MP systems, the memory footprint of this transaction module can become large, but it is insignificant when compared against the memory footprint required to run detailed models of the cores the module is replacing.

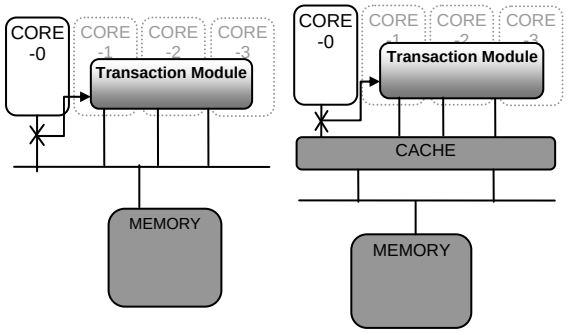

Figure 4. Replacing fast-cores with the transaction module.

Since the fast-cores are replaying detailed-core transactions, there needs to be some history of previous transactions before the tail pointers can be initialized. The initialization phase at the beginning achieves this. In this phase, the transaction module records detailed-core transactions and initializes the tail-pointers for the fast-cores based on the retired instruction-count from the detailed-core. The initial instruction offsets for each of tail pointers are chosen in a manner identical to the way offsets are chosen in the fully detailed simulation. Figure 5 shows the initialization for a 4-core system. At the end of the initialization phase, the simulator statistics are cleared so this part does not contribute to final measurements.

The initialization-phase is followed by the run-phase during which the detailed-core continues simulation and transactions are injected for the fast-

cores. Figure 6 shows the algorithm to insert the fast-core transactions. The transaction module maintains an outstanding transaction queue for each fast-core. This queue stores the injected transactions for their entire lifetime. The size of this queue is set in accordance to an architecturally equivalent queue from the detailed-core and it bounds the total number of outstanding transactions for each fast-core. This backpressure is modeled to throttle the fast-cores when the injection rate implied by the transaction buffer is driving the system at a collective rate that is higher than the system is currently able to sustain. If you think of FastMP as an algorithm for converging on a solution to the problem of approximating the performance of the system, this backpressure is the correction mechanism that prevents the FastMP algorithm from running head long into a divergent region of the solution space.

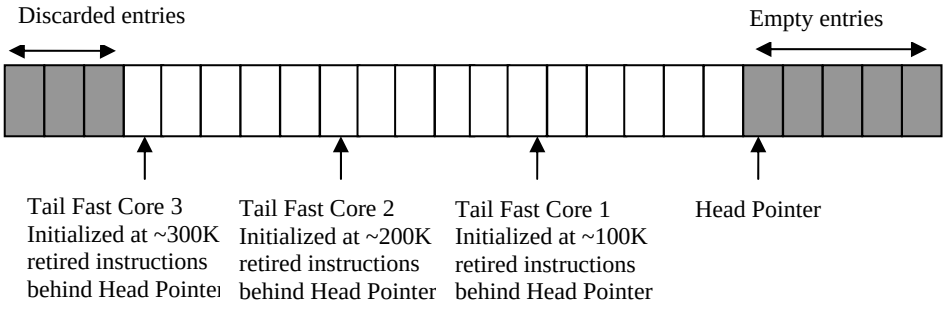

Figure 5. Transaction module at the end of the initialization phase for a 4-core system.

```
for (agent_id = 1; agent_id <= num_fast_agents; agent_id++)
{

TailPtr = TailPtrs[agent_id];

if (Current_time > (previous_injection_time[agent_id] +
delta[TailPtr])
        {
        If(OutQueue[agent_id]!=FULL)    {
                // Inject Transaction
                // Advance Tail Ptr
                // Update
                previous_injection_time[agent_id]
                }
        }
}
```

Figure 6. Algorithm for inserting fast-core transactions.

3.2. FastMP Results

Figure 7 shows the speedup obtained by using FastMP Methodology for two, four and eight cores in the case where we have a single detailed core driving the FastMP simulation. The speedup is calculated for SPECint and SPECfp benchmark suite by taking the ratio of the median simulation run times of all the traces in that benchmark suite. Clearly, FastMP provides excellent simulation speedup consistently across different core counts. Table 2 shows the percentage error in CPI measurements from FastMP compared to fully detailed simulation. We show the mean and the maximum of absolute errors across all the LIT traces of a workload. For majority of the workloads the FastMP methodology closely matches the fully detailed simulation. Most of the integer workloads in general have a very low error even with increasing core counts. However, there are certain SPECfp workloads, for example swim, equake, applu, which have a very high error especially for four and eight cores. Not surprisingly, the problematic workloads are also the ones with high memory traffic. Section 4 provides insight into these outliers and discusses the solution.

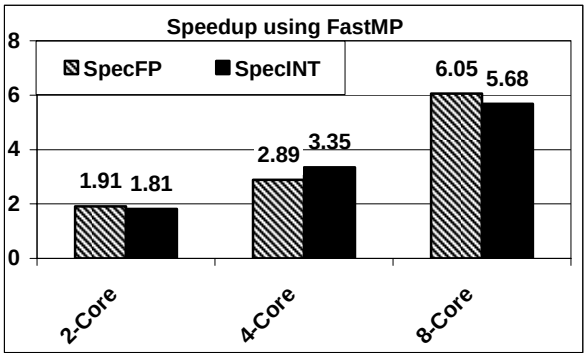

Figure 7. Speedup with FastMP methodology for two, four and eight cores.

Table 2. FastMP accuracy results.

		2-core FastMP		4-core FastMP		8-core FastMP	
Workload	Suite	Mean	Max	Mean	Max	Mean	Max
168-wupwise	FP	0.4%	1.2%	1.4%	6.0%	6.8%	24.1%
171-swim	FP	0.6%	2.0%	6.4%	11.4%	21.0%	28.1%
172-mgrid	FP	0.5%	2.7%	1.0%	2.9%	11.5%	24.8%
173-applu	FP	0.2%	0.4%	4.9%	9.4%	16.6%	38.6%
177-mesa	FP	0.1%	0.2%	0.1%	0.3%	0.2%	0.4%
178-galgel	FP	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%
179-art	FP	0.3%	4.4%	0.1%	1.0%	0.6%	3.6%
183-equake	FP	0.3%	1.1%	3.7%	6.7%	17.1%	29.9%
187-facrec	FP	0.2%	0.9%	1.0%	2.6%	0.4%	1.8%
188-ammp	FP	0.2%	0.8%	0.2%	1.1%	0.4%	1.4%
189-lucas	FP	0.1%	0.4%	2.6%	9.9%	12.3%	28.0%
191-fma3d	FP	0.3%	0.8%	1.5%	4.5%	8.7%	25.0%
200-sixtrack	FP	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
301-apsi	FP	0.2%	1.1%	0.6%	3.1%	4.0%	23.6%
164-gzip	INT	0.1%	0.3%	0.1%	0.6%	0.1%	1.1%
175-vpr	INT	0.1%	0.2%	0.1%	0.3%	0.4%	3.2%
176-gcc	INT	0.1%	0.3%	0.1%	0.4%	0.1%	1.1%
181-mcf	INT	0.1%	0.5%	0.1%	0.2%	0.9%	8.0%
186-crafty	INT	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%
197-parser	INT	0.0%	0.1%	0.1%	0.2%	0.1%	0.4%
252-eon	INT	0.2%	0.7%	0.2%	0.8%	0.3%	1.4%
253-perlbmk	INT	0.1%	0.2%	0.2%	0.6%	0.6%	1.8%
254-gap	INT	0.1%	0.4%	0.1%	0.4%	0.1%	0.4%
255-vortex	INT	0.1%	0.2%	0.1%	0.4%	0.2%	0.9%
256-bzip2	INT	0.1%	0.1%	0.1%	0.2%	0.1%	0.2%
300-twolf	INT	0.1%	0.5%	0.2%	0.8%	0.2%	0.8%

4. Adaptive Scheme

4.1. Feedback issue

For some of the workloads that place high bandwidth demands on the system (e.g., swim or applu), the interaction between the threads from different cores can cause oscillatory behavior in the competition for limited system resources. For example, when the net bandwidth demand summed across all the cores is consistently exceeding the bandwidth limits of the system, the competing threads of execution will encounter delays that cause memory requests to queue up at the point of injection into the system. The bandwidth demanded by a delayed thread when it is finally able to issue transactions is proportional to the length of the delay. This can lead to oscillatory bursts in bandwidth utilization for workloads that are consistently operating under such bandwidth constrained conditions. We observe such oscillations in our fully detailed simulations (as can be seen in Figure 8) and expect them to occur in real hardware when bandwidths are tightly constrained.

These oscillatory situations prove problematic for the FastMP methodology outlined in previous sections. The problems are correctable with minor enhancements to the methodology, which we outline in the following section.

The reason that additional enhancements are required to deal with oscillatory behavior is as follows. The FastMP methodology essentially creates a feedback loop. The fast-cores' current transaction injection rate is throttling the detailed-core's current injection rate. Yet the detailed-core's current injection rate is also determining the fast-core's future injection rate, which in turn is going to feedback and throttle the detailed-core's future injection rate. If the current fast-cores' injection rate is aiming too high, it could drive the detailed core's injection rate too low. As a result, the future fast-core injection rate will become too low allowing the detailed-core to inject at a rate higher than its injection rate in a detailed simulation of an equivalent system. This feedback loop sometimes acts as an amplifier of oscillatory behavior, which can lead to high amplitude oscillations that one would not observe in the fully detailed simulation. To put it simply, the FastMP methodology does not always respond to oscillatory conditions in the same manner as the fully detailed simulation, and sometimes those differences can cause the FastMP methodology to produce wild CPI oscillations that one would not see in the real system. Figure 8 gives an example of such behavior. The plot shows the Cycles per Instruction (CPI) over time for a problematic trace, for both the detailed-core from FastMP and a core from fully detailed simulation. This graph clearly illustrates that the small oscillations inherently present in the detailed simulation become growing oscillations when using the FastMP Methodology.

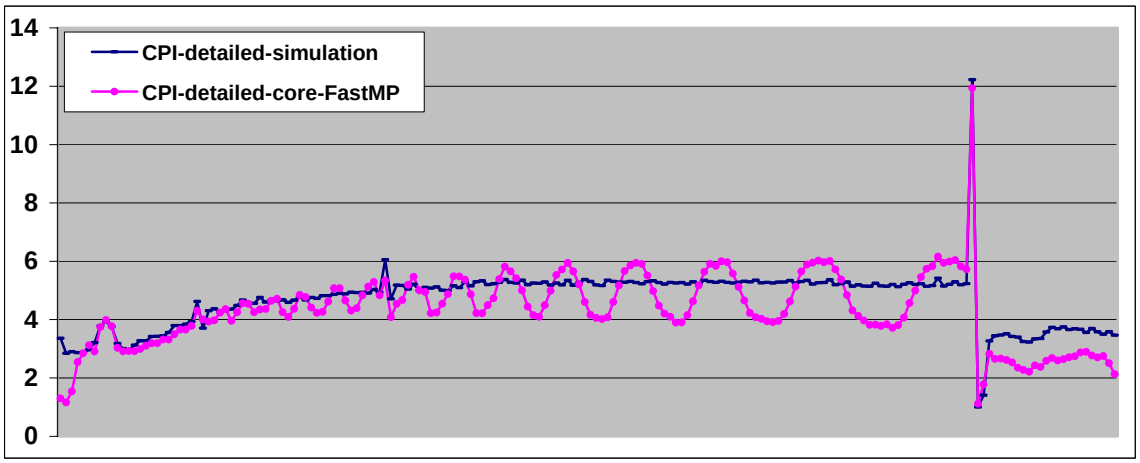

Figure 8. A time plot of CPI illustrating growing oscillation in a problematic FastMP run.

4.2. Adaptive scheme

In this section, we propose a mechanism for preventing divergent oscillations in the FastMP methodology. To understand this correction mechanism, one must first step back and view the FastMP mechanism as a model of a non-linear oscillating system. Viewed in those terms, the problem exemplified by Figure 8 is simply that the FastMP methodology is failing to dampen the oscillations of the system. To correct for it we need to add a dampening force to the FastMP system.

To achieve the desired dampening, we sought to incorporate into the methodology a mechanism for measuring divergence from the point around which the system is oscillating and then to apply a correction term to the timing data as it is extracted from the trace buffer. We use the timing data in the trace buffer as the basis for detecting divergences that indicate a need for correction. To accomplish this goal, two numbers are tracked over a moving window that is a fixed number of transactions wide (we tried windows ranging from 10k to 150k and eventually chose 100k). The first number tracked, T_target, is the number of cycles it should have taken to issue the transactions that occurred in that window when using the timing data from trace buffer. The second number tracked, T_actual, is the number of cycles it actually took, in the simulator, to issue those transactions. A correction is required when there is a difference between T_target and T_actual.

The precise algorithm we use to apply corrections is based on the analogy to a non-linear oscillating system. When T_actual is greater than T_target, it means the fast-core injections are lagging behind and there is a need "to inject additional energy into the

system" to prevent a large undershoot in the injection rate. When T_actual is less than T_target, it means the fast-core injections are too aggressive and there is a need to "pull energy out of the system" to prevent a large over shoot of the injection rate. The way to achieve the goal in both cases is to make an on-the-fly adjustment to the time that the next transaction in the trace buffer is injected. For overshooting case, if the trace buffer says to issue the next transaction in C cycles, then instead inject at time C', where C' > C. For undershooting, pick C' < C. The exact formulation which we choose for C' is the following:

$$C' = (T\_target / T\_actual)^{\wedge}2 * C$$

There are two ways of interpreting this formula. First, by squaring the ratio we have a correction term, which gets increasingly more aggressive as the ratio further deviates away from unity. When the ratio is close to unity, the square of the ratio is almost the same as the original ratio, but when the ratio is far from unity the square of the ratio is considerably farther from unity than the original ratio. Second, this formulation is consistent with the analogy to a non-linear oscillating system: in non-linear oscillating systems, a typical dampening term will vary with the square of the deviation from the system's point of stability.

One might question why T_target is chosen as the stable point of the system when formulating the adaptive mechanism's correction term. The reason we choose T_target goes back to our underlying assumption that the workload being simulated is homogeneous. Since the FastMP algorithm assumes that all cores in the simulated system have the same performance characteristics, any measured difference between the performance characteristics of the simulated cores is the correct measure of divergence. With that in mind, we take the behavior of the detailed-core (T_target) as the basis against which divergence in the behavior of each fast-core (T_actual) is measured. This does mean that we are measuring divergence locally for each thread. Alternative metrics for measuring divergence, which take a global view (for example, taking a mean of the divergence across all cores), have not been studied, but may be an interesting topic for additional study.

4.3. Results with adaptive scheme

Table 3 shows the percentage error in CPI measurements from FastMP compared to fully detailed simulation. We include the results from the base FastMP scheme for comparison. Our choice of a window size of 100k transactions is based on our extensive experimentation with window sizes ranging from 10k to 150k. We choose 100k because it provided the best accuracy overall. Clearly, adapting the injection rate from the fast-cores improves accuracy and significantly reduces the mean errors for the problematic workloads without affecting the workloads that initially had low errors. Figure 9 clearly shows that the adaptive scheme reduces the oscillations present in the system, which leads to a higher accuracy. In the 8-core cases, there are certain workloads like wupwise, swim, applu, equake and fma3d, which have high worse case errors. With increasing core counts, it is possible that the error will increase. One way to offset this is to use more than one detailed core in the system. A detailed study of this is part of our future work. The simulation speedup using the adaptive scheme remains similar to Figure 7.

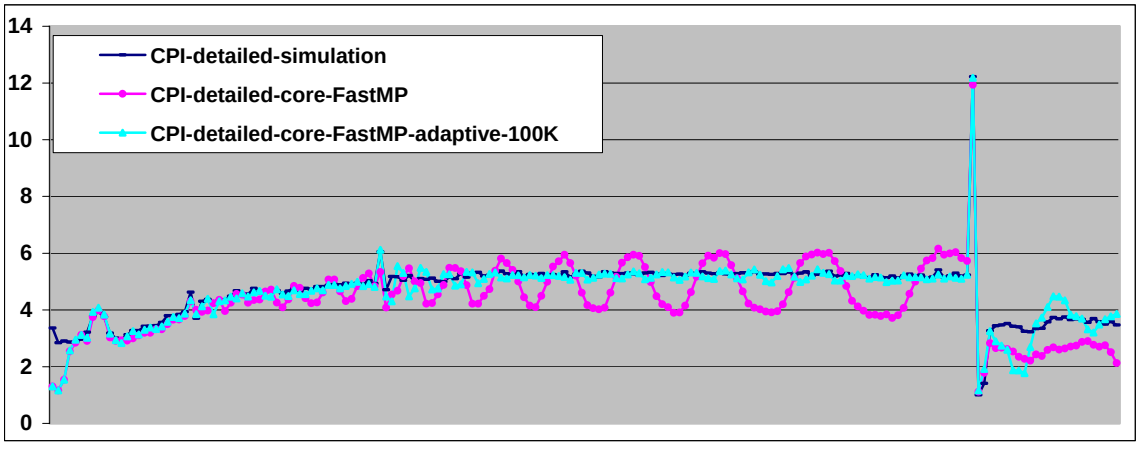

Figure 9. Reduced oscillations with adaptive FastMP scheme

Table 3. FastMP results using the adaptive scheme.

		2-core FastMP		2-core FastMP Adaptive		4-core FastMP		4-core FastMP Adaptive		8-core FastMP		8-core FastMP Adaptive	
Workload	Suite	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max
168-wupwise	FP	0.4%	1.2%	0.2%	0.3%	1.4%	6.0%	0.5%	1.7%	6.8%	24.1%	3.6%	13.2%
171-swim	FP	0.6%	2.0%	1.1%	2.3%	6.4%	11.4%	1.9%	4.2%	21.0%	28.1%	6.7%	23.7%
172-mgrid	FP	0.5%	2.7%	0.5%	2.7%	1.0%	2.9%	1.0%	3.1%	11.5%	24.8%	4.6%	9.2%
173-applu	FP	0.2%	0.4%	0.5%	1.1%	4.9%	9.4%	1.6%	4.1%	16.6%	38.6%	6.3%	15.6%
177-mesa	FP	0.1%	0.2%	0.1%	0.2%	0.1%	0.3%	0.1%	0.3%	0.2%	0.4%	0.2%	0.4%
178-galgel	FP	0.0%	0.1%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.1%
179-art	FP	0.3%	4.4%	0.1%	0.5%	0.1%	1.0%	0.2%	1.7%	0.6%	3.6%	0.7%	6.7%
183-equake	FP	0.3%	1.1%	0.5%	1.3%	3.7%	6.7%	0.4%	1.9%	17.1%	29.9%	6.4%	14.9%
187-facrec	FP	0.2%	0.9%	0.2%	0.9%	1.0%	2.6%	0.9%	2.6%	0.4%	1.8%	0.4%	1.8%
188-ammp	FP	0.2%	0.8%	0.2%	0.8%	0.2%	1.1%	0.3%	1.1%	0.4%	1.4%	0.6%	2.3%
189-lucas	FP	0.1%	0.4%	0.1%	0.7%	2.6%	9.9%	0.8%	3.1%	12.3%	28.0%	5.4%	9.0%
191-fma3d	FP	0.3%	0.8%	0.4%	1.3%	1.5%	4.5%	0.8%	2.1%	8.7%	25.0%	4.5%	18.2%
200-sixtrack	FP	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
301-apsi	FP	0.2%	1.1%	0.1%	0.4%	0.6%	3.1%	0.4%	1.6%	4.0%	23.6%	0.5%	2.3%
164-gzip	INT	0.1%	0.3%	0.1%	0.3%	0.1%	0.6%	0.1%	0.6%	0.1%	1.1%	0.2%	3.3%
175-vpr	INT	0.1%	0.2%	0.1%	0.2%	0.1%	0.3%	0.1%	0.2%	0.4%	3.2%	0.5%	3.6%
176-gcc	INT	0.1%	0.3%	0.1%	0.2%	0.1%	0.4%	0.1%	0.4%	0.1%	1.1%	0.1%	1.1%
181-mcf	INT	0.1%	0.5%	0.1%	0.4%	0.1%	0.2%	0.1%	0.2%	0.9%	8.0%	0.3%	2.1%
186-crafty	INT	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%
197-parser	INT	0.0%	0.1%	0.0%	0.1%	0.1%	0.2%	0.1%	0.2%	0.1%	0.4%	0.1%	0.7%
252-eon	INT	0.2%	0.7%	0.2%	0.7%	0.2%	0.8%	0.2%	0.8%	0.3%	1.4%	0.3%	1.4%
253-perlbmk	INT	0.1%	0.2%	0.1%	0.2%	0.2%	0.6%	0.2%	0.6%	0.6%	1.8%	0.6%	1.7%
254-gap	INT	0.1%	0.4%	0.1%	0.4%	0.1%	0.4%	0.1%	0.4%	0.1%	0.4%	0.1%	0.4%
255-vortex	INT	0.1%	0.2%	0.1%	0.2%	0.1%	0.4%	0.1%	0.4%	0.2%	0.9%	0.2%	1.0%
256-bzip2	INT	0.1%	0.1%	0.1%	0.1%	0.1%	0.2%	0.1%	0.2%	0.1%	0.2%	0.1%	0.2%
300-twolf	INT	0.1%	0.5%	0.1%	0.5%	0.2%	0.8%	0.2%	0.8%	0.2%	0.8%	0.2%	0.8%

5. Conclusion and Future Work

This paper presents a simulation methodology aimed at speeding up multi-core simulation runtimes for homogeneous workloads. This FastMP methodology achieves speedup by doing detailed simulation of a subset of the cores in an MP simulation and approximating the traffic from the remaining cores by analyzing the detailed-cores' traffic. We evaluate our methodology by implementing it in a robust cycle-accurate simulator and use the entire SpecCPU suite for experimentation and thoroughly studying the case were a single detailed core is driving the MP simulation. Our base implementation provides a significant simulation run-time speedup and low error for a majority of the workloads. We identify a feedback issue with FastMP and propose a simple extension to the base technique, which adapts the injection rate from the fast-cores to prevent under-damped oscillations in high bandwidth cases. This adaptive scheme further closes the gap between FastMP and detailed simulation and retains the average FastMP speedups of 1.9, 3.1, and 5.9 for 2, 4 and 8 cores respectively.

As future work in this area, we plan to pursue larger system configuration studies and experiment with core counts above eight. We also intend to do detailed studies of the tradeoff between error and the total number of detailed cores used to drive the simulation. Finally, we intend to investigate extensions to the current technique to enable homogeneous shared memory server workload, such as TPC-C or SAP.

Acknowledgments

We would like to thank all our reviewers for their valuable feedback. Special thanks to Lee W. Baugh who worked on an early implementation of FastMP.

REFERENCES

[1] Penry, D.A.; Fay, D.; Hodgdon, D.; Wells, R.; Schelle, G.; August, D.I.; Connors, D., Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors, High-Performance Computer Architecture, 2006. The 12th International Symposium on, vol., no.pp. 27-38, Feb 11-15, 2006.

[2] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A.Wood. Wisconsin Wind Tunnel II: A fast and portable architecture simulator. In Workshop on Performance Analysis and its Impact on Design (PAID), June 1997.

[3] K. C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, and J. Emer. Simulating a chip multiprocessor with a symmetric multiprocessor. In Boston Area Architecture Workshop, January 2005 (BARC 2005).

[4] M. Chidister and A. George. Parallel simulation of chip multiprocessor architectures. ACM Transactions on Modeling and Computer Simulation, 12(3):176–200, July 2002.

[5] G. Hamerly, E. Perelman, and B. Calder, "How to Use SimPoint to Pick Simulation Points", ACM SIGMETRIC Performance Evaluation Review, 2004.

[6] R Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating Microarchitectural Simulation via Rigorous Statistical Sampling", International Symposium on Computer Architecture, 2003.

[7] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In International Conference on Computer Design, pages 468--477, 1996.

[8] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research", Vol. 1, June 2002.

[9] Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins, "Characterizing and Comparing Prevailing Simulation Techniques," International Symposium on High-Performance Computer Architecture, Feb 2005.

[10] http://www.spec.org, “SPEC – Standard Performance Evaluation Corporation”.

[11] http://www.intel.com/products/processor/coresolo/, “Intel Core Solo”.

[12] http://www.intel.com/technology/architecture/coremicro/, “Intel® Core™ Microarchitecture”.

[13] D. Chiou, H. Sunjeliwala, D. Sunwoo, J. Xu, N.Patil, “FPGA-based Fast, Cycle-Accurate, Full-System Simulators” in 2nd Workshop on Architecture Research using FPGA Platforms, 2006.

[14] N. Dave, M. Pellauer, Arvind, J. Emer, ”Implementing a Functional/Timing Partitioned Microprocessor Simulation with an FPGA”, in 2nd Workshop on Architecture Research using FPGA Platforms.

[15] http://www.intel.com/multi-core/,“Intel Multi-core”.

[16] http://multicore.amd.com/en/Technology/, “AMD Multi-core”.