Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1814 – 1823

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Challenges of reducing cycle-accurate simulation time for TBP applications

Alexandru C. Iordan*, Magnus Jahre, Lasse Natvig

Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Cycle-accurate simulation is an important tool that depends on the computational power of supercomputers. Unfortunately, simulations of modern multi-core platforms can take weeks or months. In this paper, we look into the challenges of employing a sampling based technique for reducing simulation time of multi-threaded applications. We introduce FASTA, a simple 3-phase methodology for reducing the simulation time of Task Based Parallel (TBP) applications. FASTA takes advantage of the periodic behavior of parallel applications and identifies a small number of representative execution samples. By exploring a large design space we show that even though we can not use FASTA for every type of application, there are some for which a 12x speedup can be achieved with an accuracy error as low as 2.6%.

Keywords: Task based programming; simulation; chip-multiprocessor.

1. Introduction

Increasing in complexity with each generation, developments in hardware and software design require ten of thousands of computational hours for simulation and testing. For computer architects simulation is also an indispensable technique for exploring novel ideas. The main advantages of simulation are increased flexibility for design space exploration and outputs with extensive, noninvasive and detailed measurements. In this way, simulation bridges the gap between analytical modeling and real world measurements. Unfortunately, the cost of these advantages is long simulation times.

In today's ICT market, Chip Multiprocessors (CMPs) or multi-core architectures are the platform of choice in almost all segments. CMPs were first introduced as a solution to a design constraint known as the *power wall*. For decades, CPU designers took advantage of the increasing transistor count on a chip to boost computational power. They did this by increasing the chip's clock frequency and by exploiting instruction level parallelism (ILP) more aggressively. However, cooling system limitations and diminishing returns from ILP created the need for a new architectural approach. CMPs utilize the hardware resources provided by production technology improvements to create several less complex cores. This architecture is able to exploit both instruction and thread level parallelism allowing for an aggregate performance increase. At the same time, they require a lower power budget [1].

The switch to the multi-core model has placed a new burden on programmers. CMPs can not be fully exploited using sequential programming and parallelization is required to achieve maximum performance. For this reason,

*Corresponding author
E-mail address: iordan@idi.ntnu.no.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.350

there is a considerable interest in programming models that can simplify parallel programming. In this work, we focus on *Task Based Programming* (TBP), a parallel programming model that has received significant attention recently [2, 3, 4, 5]. TBP's key idea is that the programmer partitions the application into tasks and specifies the dependencies between them [4]. A task is a light weight unit of work, and all independent tasks can be executed concurrently. A runtime task management library distributes tasks to processing units and enforces all dependencies.

There have been many attempts to reduce simulation time. For single-core simulation, sampling-based techniques are very effective at reducing simulation time with a small loss in accuracy. SimPoint tries to identify a set of samples that are representative for the entire execution [6]. The complete execution is inferred from simulating and weighing only these samples thus reducing the total simulation time. In a different approach, SMARTS uses statistical sampling theory and samples the execution at fixed interval [7]. A more detailed study of simulation time reduction methods can be found in [8] showing that sampling achieves high accuracy for single threaded applications.

There are two major issues when sampling multi-threaded applications. First, we have to use a metric that is proportional to forward progress of execution. Single-threaded applications are sampled using metrics like the committed number of instructions or IPC/CPI. As argued by Alameldeen and Wood, these performance metrics are not suited for parallel executions [9]. For example, while executing a busy wait loop, the instruction count of the thread will increase but the execution will not move forward. Second, we need to account for the interleaving of threads in different executions. Because of shared resources and data dependencies, different runs of the same parallel application can have different instruction or task streams.

In this paper, we use a 3-phase approach called FASTA that aims to reduce simulation time of TBP applications running on CMP platforms. First, we run a fast, low detail simulation to sample the execution and gather measurements. Next, we use a clustering algorithm to group together similar samples and to select a representative point for each group. Finally, we simulate the representative samples in detail and we use the results to estimate the full detailed execution. Through our experiments we found that the thread interleaving impacts the accuracy of the FASTA results more for some classes of parallel applications than others. This impact becomes more prominent with the increase of core count. However, we found a class of parallel applications that are not affected by the interleaving problem. In these cases, FASTA results show an average error below 4% and a speedup of 12x maximum.

2. Background

Researchers have been motivated to find ways to reduce simulation time due to simulation's key role for software and hardware development [8]. One proposed technique is to reduce the input set of the benchmarks [10, 11]. This approach assumes that the characteristics of the full input set can be preserved even though fewer instructions are actually executed. However, the accuracy of this approach is generally poor [8].

Another technique for addressing long simulation time is truncated execution. In this approach, a continuous section of X instructions is selected and simulated in detail. If this section is not in the beginning of the application, *fast-forwarding* or *checkpointing* can be used to start the simulation at the desired point. When fast-forwarding, a simulator only emulates the hardware system up to the desired execution point. With checkpointing, a snapshot of the simulated system state is stored, allowing for a restart from that point.

In single-threaded environments, sampling-based techniques have been used to reduce simulation time while maintaining good accuracy. There are primarily 3 classes of sampling techniques [12]:

- *Representative sampling techniques* attempt to identify a sample or a group of samples in the simulated code that can be held representative for the entire execution. Perelman et al. [6] use the K-means algorithm to group similar simulation points into clusters and then calculate a centroid for each cluster. The "closest" sample point to the centroid is considered representative for the entire cluster.
- *Periodic sampling techniques* select portions of the simulated code at periodic intervals. SMARTS [7] samples a large number of very small (in terms of number of instructions) execution points which are later simulated in detail. An important characteristic of SMARTS is that it allows to trade off result confidence against speed of execution. Building on their work with SMARTS, Wenisch et al. developed another

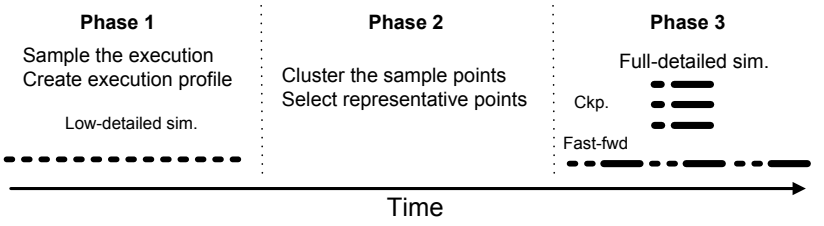

Fig. 1. The FASTA flow

statistical sampling methodology called SimFlex [13]. Using a different approach for warming the CPU units, SimFlex reduces the total simulation time. In addition, SimFlex can be employed for multiprocessor throughput applications like those in the TPC-C OLTP or Specweb99 benchmark suites.

- *Random sampling techniques* try to combine the results from N randomly selected simulation points to produce the overall result. To improve the accuracy of this technique Conte et al. [14] suggest the use of longer warm-up periods and increasing the number of instructions in each sample.

For very large input sets even functional simulation (i.e. fast-forwarding) is too time consuming. In such cases, a technique called *direct-execution* can be employed [15]. The host machine is used to execute the application and checkpoint the execution. This checkpoint is then transferred to the simulator. In order for this technique to work, the host and simulated machines must use the same ISA or cross-compilation needs to be employed before the transfer.

Simulation can be also accelerated by varying the abstraction level or by exploiting parallel hardware. Rico et al. [16] argue that the level of abstraction used by most current simulators can be misleading for some studies like early processor design or high-level explorations. They introduce *TaskSim*, an architecture simulator designed for many-core platforms that can provide several levels of modeling abstraction. Another way to reduce simulation time is to distribute the simulation across multi-core or multi-machine environments. Graphite[17] and SlackSim[18] are two parallel simulation tools.

3. FASTA

In this paper, we propose a simple methodology for *representative sampling*, called *FAst Simulation of TBP Applications* (FASTA). Like Perelman et al. [6], our technique exploits the periodic behavior of parallel applications and identifies representative sections for each interval of the execution. There are 3 main differences between our approach and SimPoint: (1) we target the simulation of parallel application thus (2) we use a different work-related progress metric than the traditional CPI/IPC and (3) our methodology is not architecture independent.

In this study, we used the simulated number of cycles (which we converted to simulated execution time) as a metric for profiling and validating FASTA simulations against full-detailed simulations. This metric aggregates the impact of all components in a simulation platform (execution units, memory hierarchies, interconnects etc) and is the most used metric for measuring overall performance of a system [19]. However, if other metrics are of interest (cache accesses, number of executed instructions etc.), FASTA can be easily adapted to reflect those. It is just a matter of what data you chose to record during the sampling phase.

Fig. 1 gives an overview of the flow of our approach. To better understand how FASTA works and how different parameters can affect the results, we summarize each phase in the next subsections.

3.1. Phase 1: Sampling

We gather the simulated number of cycles for every *s* completed tasks and create an execution profile. Section 4 discusses in detail why we use the number of completed tasks as a metric in our work. The value of *s* has an impact on the granularity of the profile: a low value will lead to a more detailed profile while a high value to a coarser one. A trade off is needed because simulating with a low *s* value requires more time while a coarser profile can miss patterns in the execution.

3.2. Phase 2: Clustering and Representative Point Selection

In Phase 2, we use the *K-means* algorithm on the resulting sample population. *K*-means is an well known iterative clustering algorithm [20]. The main input parameter for this algorithm is the number of cluster to be created (*K*). *K* affects both the accuracy and the potential speedup of the FASTA simulation.

In our first study about efficient simulation of TBP application [21], we explored the characteristics of clustering that can impact FASTA's accuracy. We also introduced a selection method for the initial values of the centroids called *Initialization aware* that we use in the current paper. A detailed presentation of our *K-means* implementation can be found in [21] as well.

The algorithm outputs (1) the coordinates of the centroids, (2) a set of weights for each cluster and (3) a mapping of each data point to a cluster. The "closest" sample point to its respective centroid is selected as representative for each cluster.

3.3. Phase 3: Detailed Simulation

In Phase 3, we start the full-detailed execution of the representative samples using the checkpoints or fast-forwarding. When using checkpoints, we can start the simulations concurrently because each checkpoint is independent of the others.

In the fast-forwarding approach, we emulate the execution up to the representative sample point where we switch to full-detailed simulation. Then, we simulate the representative sample and dump the measurements before we fast-forward to the next representative point. However, switching back and forth between simple and full-detailed mode adds an overhead that affects speedup. Also, since some buffers are cleared when switching from detailed to simple simulation (TLBs for example) accuracy can also be impacted.

For both approaches, the detailed results are weighted according to the output of *K*-means and an estimate of the complete execution in detailed mode is created.

We experimented with 3 different methods to generate the checkpoints:

- *Checkpoint Representative Sample Points (CRSP)*: After the representative sample points are selected, a second low-detail simulation is started and checkpoints are created only for the representative sample points (see Fig. 1).
- *Checkpoint All Sample Points (CASP)*: Generate a checkpoint for each sample point during Phase 1.
- *Checkpoint at Intervals (CI)*: This approach is a trade-off between CRSP and CASP. During Phase 1, checkpoints are generated every X sample points, where X can be defined by the user. After selecting the representative sample points, the nearest preceding checkpoint is used to restart the simulation and create a new checkpoint for the representative sample.

All the checkpoints have been created using the built-in functionality of our simulator. We did not try to improve in any way this functionality, neither for storage nor for speed. Both [22] and [23] present ways in which checkpoint size and restoration time can be reduced. We consider that such work, though interesting and important in the simulation field, was beyond the scope of our research. We kept focus on validating our methodology rather than trying to optimize it.

4. The challenges

There are 2 main challenges to be addressed when employing sampling for reducing the simulation time of multi-threaded applications:

- sampling using a metric that is proportional to forward progress of execution
- the change in the instruction stream of a parallel application from one execution to another as races resolve differently on different runs

Table 1. BOTS input workloads

	FFT	Fib	nQueens	Sort	SparseLU	Strassen
Input	2^{25} floats	30'th element (no cut-off)	14x14 board	2^{25} integers	200x200 sparse matrix of 25x25 blocks	2048x2048 matrix

To address the first issue, we use *the number of completed tasks* as a work-related progress metric. We count the tasks collectively for all working threads. In this way we eliminate the problem of spin locks being recorded as false progress of the execution.

The second issue is much more difficult to deal with. Wenisch et al. conclude that it is unclear how to sample general multiprocessor applications so they focus their SimFlex methodology on throughput parallel applications [13]. Because in our approach we are trying to estimate the results of a detailed simulation using results from a very simple one, we need to assess the variability of the thread interleaving. To do that, we have experimented with 2 low detailed simulation modes. The first one models a simple CPU with instantaneous access to memory (Atomic mode) while the second one models the same CPU but with delays in accessing the memory system (Timing mode). By comparing the Atomic and Timing execution profiles, we can reason about the impact the memory model has on interleaving the threads. We also assess how the difference in CPU model affects the profiles, by comparing the results of the 2 low detailed simulation modes with the detailed ones.

5. Methodology

We use the cycle-accurate GEM5 simulator [24] in full-system mode with the 2.6.27 Linux kernel. We simulate both the application and the operating system (OS) because TBP libraries rely on the OS to manage shared memory and provide a thread and process abstraction. Our simulated platforms use 2-, 4- and 8-core CPUs, with a clock frequency of 1GHz and a 2 level cache hierarchy with a split L1 private cache (64 KB) and a shared L2 (2 MB). As mentioned before, for Phase 1 we have experimented with 2 simulation modes. The *Atomic* mode models a simple 5-stage pipeline In-Order CPU that can accesses the memory hierarchy without any resource contention or queuing delay. The *Timing* mode models the same simple CPU but its access to memory is realistic and includes delays. The simulations in Phase 3 are performed using a detailed Out-of-order CPU model.

In this work, we use a subset of the Barcelona OpenMP Task Benchmark Suite (BOTS) [25]. We changed the default OpenMP parallelization to a Wool [4] implementation. We also customized this library to signal GEM5 every time a task is completed. In this way, the simulator is able to keep record of each completed task and to implement the sampling for Phase 1. Table 1 lists the benchmark input sets we used in our experiments. Using these inputs, the full detail simulation of some benchmarks running on the 8-core platforms lasted more than 18 days. For this reason we decided not to increase input sets with the core count.

6. Results

In the next subsections we present our accuracy, speedup and parameter variation results. Our design space is defined on 3 parameters: the sample size (ranging from $s = 500$ to $s = 2000$ samples), the number of clusters (ranging from $K = 4$ to $K = 10$ clusters) and the core count (2-, 4- and 8-core systems). We investigate the impact the interleaving of threads has on profiling the applications and ultimately on the accuracy of the FASTA results. We experiment with both checkpointing and fast-forwarding in order to determine the best speedup that we can achieve for our simulations.

6.1. Accuracy and speedup results

In order to quantify the error of the FASTA results we define the *performance* as the total number of tasks divided by estimated simulated time. We use the complete full-detail simulation of each benchmark as a baseline. The accuracy error is calculated as $E = (P_{FASTA} - P_{Full})/P_{Full}$ where P_{FASTA} is the estimated performance of the FASTA simulation and P_{Full} is the performance of the baseline.

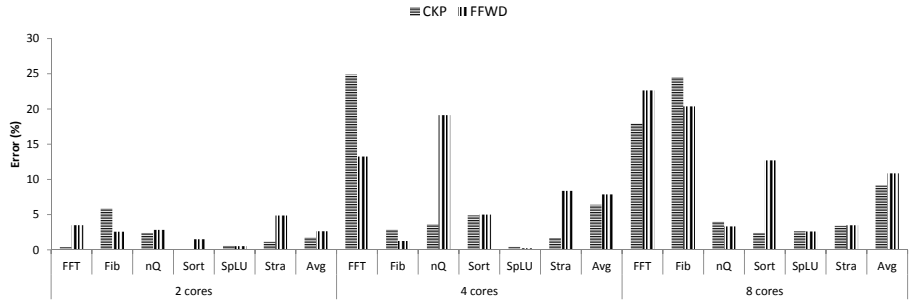

Fig. 2. The accuracy of FASTA simulations

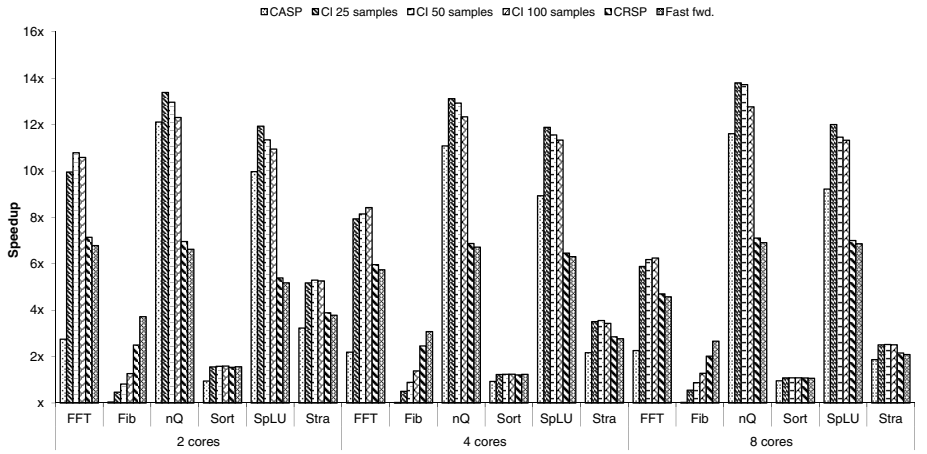

Fig. 3. The speedup of FASTA simulations

Fig. 2 presents the accuracy results for all benchmarks for $s = 1000$ completed tasks and $K = 6$ clusters across all test platforms. More results on the full ranges of both s and K are presented in Section 6.2. As the average results show, the accuracy tends to decrease from 2- to 8-core systems. This trend can be partly explained by the fact that we did not increase our workloads with the number of cores. Because of this, the clusters are less differentiated for 8 cores, they can merge together and the selection of representative samples is erroneous. SparseLU and Strassen are affected by this type of errors. However, higher errors (like those of FFT and Fib) are due to the way threads interleave on different runs, this being discussed further in Section 6.3.

As mentioned in Section 3.3, using the fast-forward approach during Phase 3 can impact the accuracy of the results, not only the speedup. This is a short-coming of our experimental framework and not a limitation of the methodology. However, the clearing of buffers when switching back and forth between simulation modes is not the only cause for error. Thread interleaving has an impact here as well and is discussed in more details in Section 6.3.

Fig. 3 presents the potential speedups that can be achieved by using FASTA when compared to a complete full detailed execution of the benchmarks. These results are calculated for $s = 1000$ completed tasks and $K = 6$ clusters. Based on our results, we can conclude that for benchmarks with large input sets like FFT, Sort and Strassen, the checkpointing overhead is larger so CASP takes longer than fast-forwarding. If we use CRSP or CI then it will be generally faster than fast-forwarding. For Fib, nQueens and SparseLU checkpointing is always faster than fast-forwarding. It is also worth mentioning that checkpointing methods require storage space. FFT, Sort and Strassen need on average 156 GB, 176 GB and 115 GB respectively for a single simulation using the CASP method.

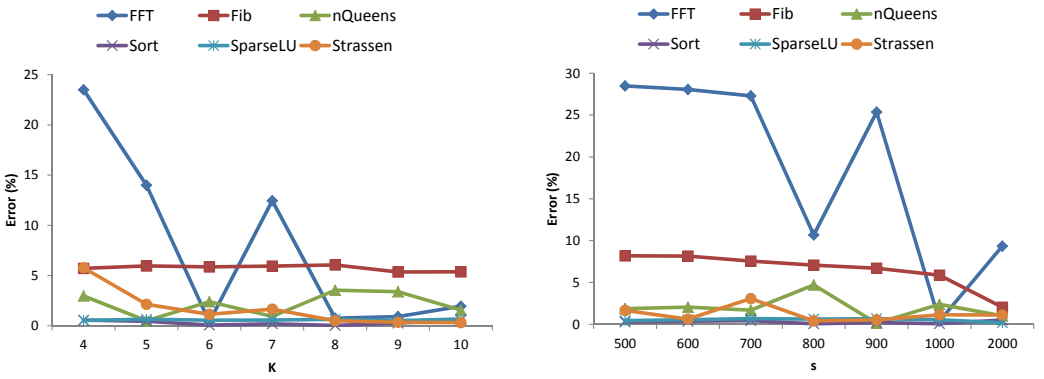

Fig. 4. The parameter analysis: the variation of K (s = 1000) and the variation of s (K = 6) - 2 cores

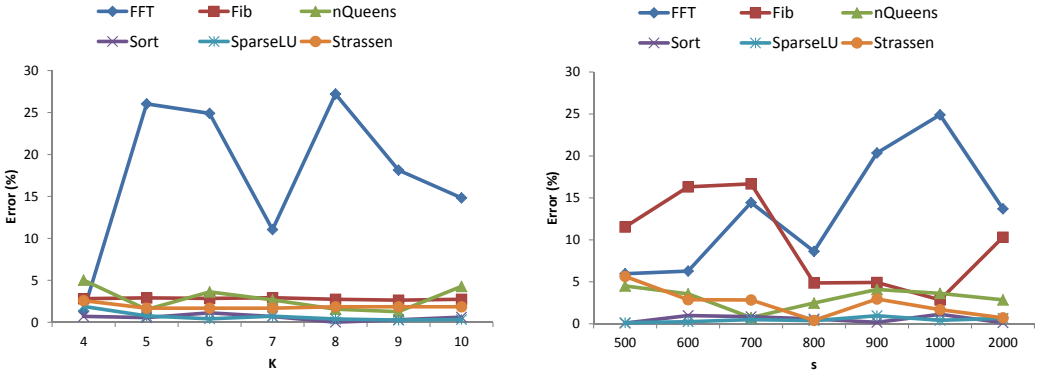

Fig. 5. The parameter analysis: the variation of K (s = 1000) and the variation of s (K = 6) - 4 cores

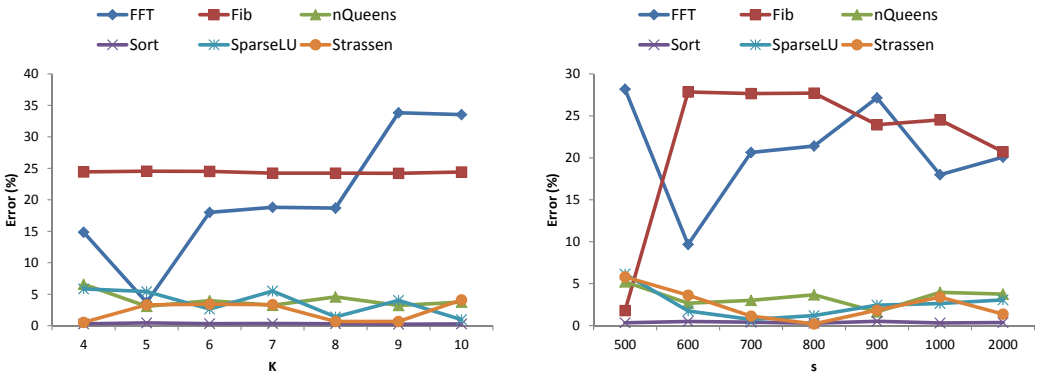

Fig. 6. The parameter analysis: the variation of K (s = 1000) and the variation of s (K = 6) - 8 cores

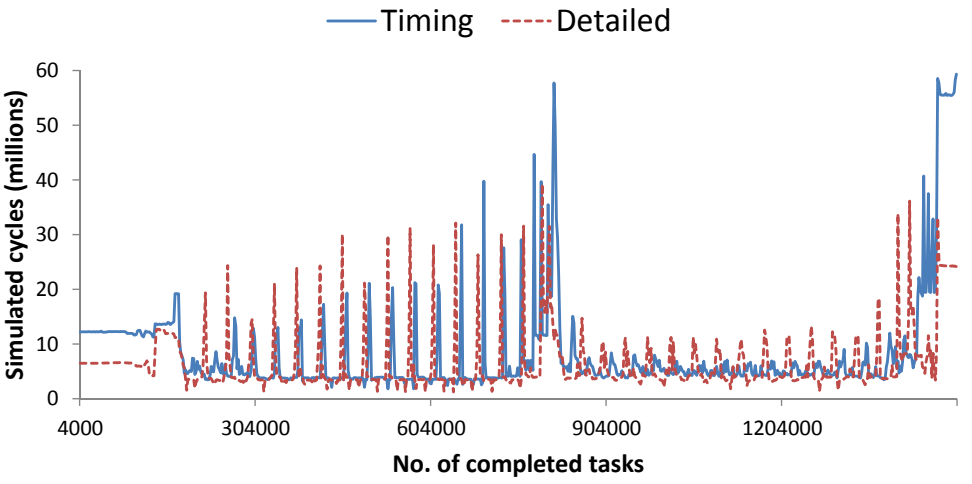

Fig. 7. FFT - Timing mode profile vs. Detailed mode profile

6.2. Parameter analysis

Fig. 4 — 6 show part of our results for the parameter study. Because of space limitation, we only present 2 sets of results for each test platform: the variation of K for $s = 1000$ and the variation of s for $K = 6$.

For reasons discussed in Section 6.3, it is very hard to clearly quantify the effect of s or K on the accuracy of FASTA. However, for our simulations, we observed that a profile granularity of 800 - 1000 points gives the best accuracy results. This means, that knowing the total number of tasks an application will execute, we could calculate the value of s.

For a low value K, several patterns in the profile merge together in a cluster which results in a poor estimation. Generally, the higher the K the better the FASTA estimation is. However, we need to consider that K determines the number of samples that are simulated in detail during Phase 3. If we simulate too many samples during this phase, the potential speedup of FASTA decreases. So the idea is to select a low as possible K without affecting the accuracy of the estimation. In our experiments a value of 5 or 6 for K would yield a good accuracy level.

6.3. Thread interleaving

Given a random sample point in an application's profile, we were expecting to see different values of the simulated execution time when we simulate that application in different modes (Atomic, Timing or Detailed). What we also found was that a sample point will not contain the same work (the same tasks) across the 3 simulation modes. This can be seen as a "shift" of the profile spikes in Fig. 7. Because of this "shift", the clusters and representative points selected in Phase 2 from a low-detail simulation profile do not map onto the Detailed profile. This behavior causes the FASTA estimations to be erroneous. This happens because threads do not interleave the same way for different simulation modes.

FFT calculates the discrete Fourier transform using the Cooley-Turkey algorithm and its execution is divided in 3 main phases. Because of the recursive approach used, tasks form a binary tree in each of the algorithm's phases. As the execution progresses, all sample points end up containing tasks from each of the 3 phases. The "shift" visible in Fig. 7 is caused by races in accessing the data and the way the dependencies among tasks are handled. The estimation error varies from an average of 7.7% in the 2-core simulations to an average of 20.2% in the 8-core ones.

Fib, nQueens, Sort and Strassen are also implemented recursively and show the same behavior. However, how prominent the profile "shift" is differ from benchmark to benchmark. We found that the form of the application's task tree is closely correlated with the "shift". Fib and FFT have a binary task tree (2 children for every parent) and they are the ones affected by the largest profile "shift". Sort generates a task tree with 4 children for every

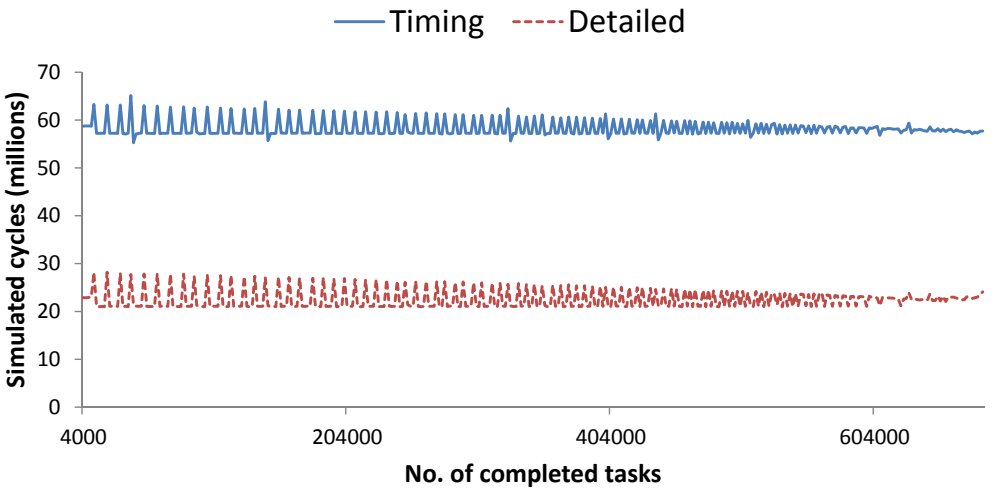

Fig. 8. SparseLU - Timing mode profile vs. Detailed mode profile

parent and it is far less affected than the previous two benchmarks. For Strassen the "shift" is even less significant since its task tree has 7 children for every parent. The number of parent tree-node translates into the number of situations when the task stream can change. The higher the number of parent nodes, the higher the chances that race conditions will determine a different task stream for different executions.

In the case of benchmarks with more than 1 type of tasks (FFT and Sort), resource contention is another cause of changes in the task stream. This can be seen by analyzing an Atomic mode profile against a Timing mode one. Since the two simulation modes use the same simple CPU model, the profile "shift" is caused only by races in accessing the different memory models. By overlapping Timing mode profiles and Detailed mode ones, we could see yet another type of resource contention: the ones for CPU resources. Due to space limitation we could not include any such figures in this paper.

We also observed that the profile "shift" becomes more significant with the core count. In the 2-core experiments we recorded no such behavior, while for the 4-core ones profiles differed in some cases. The 8-core profiles are affected the most. This is due the fact that the number of races for resources (software or hardware) increase with the number of worker threads. So the number of situation when the task stream can change is greater for a higher core count.

There is 1 benchmark that was not affected by the profile "shift" (see Fig. 8). *SparseLU* calculates the LU matrix factorization. The algorithm divides the input array into smaller blocks on which computation is performed. This is not a recursive algorithm and only 1 thread spawns all tasks. Because there is only 1 parent task and all children tasks perform the same work, race conditions do not change the task stream of this benchmark's execution. The average estimation error ranges from 0.5% for the 2-core test system to 3.7% for the 8-core one.

The fast-forwarding approach in Phase 3 is also affected by the thread interleaving problem, even though it is technically a single execution. If an application has a deep task tree (like FFT or Fib), then switching back and forth between 2 simulation modes will cause a similar "shift" of the representative samples as the one seen in Fig. 7. In the case of SparseLU, using fast-forwarding is almost as accurate as using checkpointing (see Fig. 2).

7. Conclusion

In this paper, we have introduced FASTA, a simple methodology for reducing simulation time of TBP applications running on multi-core platforms. In a 3-phase approach, FASTA samples and profiles the execution (Phase 1), identifies representative sample points through clustering (Phase 2) and simulates in detail the representative points (Phase 3). The results in Phase 3 are weighted and an estimate of the full-detail execution is created.

We investigated the challenges of using a sampling based methodology in a multi-threaded environment. To adapt to this environment, we propose the number of completed tasks as a progress metric for sampling. Our experiments over a large design space show that FASTA estimations error can be below 4% for a certain class of appplications. At the same time FASTA simulations can achieve up to 12x speedup. Applications that have a deep task tree, show increasing estimation errors with the core count. The reason for these errors is the interleaving of threads that causes a different task stream for each simulations mode. As a result, the representative samples calculated from a low-detail simulation profile are not representative for the detailed simulation. We also determined that the level in which the task stream changes between runs is correlated with the number of parent-nodes in an application's task tree. Our parameter analysis shows that for $K = 6$ clusters, in most cases the estimation error of FASTA is below 5%. These results are encouraging and they recommend FASTA for future research.

References

[1] S. Fuller, L. Millett, Computing Performance: Game Over or Next Level?, Computer 44 (1).

[2] Cilk++: A quick, easy and reliable way to improve threaded performance, `http://software.intel.com/en-us/articles/intel-cilk-plus/`.

[3] D. Leijen, W. Schulte, S. Burckhardt, The design of a task parallel library, in: Proc. of the 24th Conference on Object Oriented Programming, Systems Languages and Applications, 2009.

[4] K.-F. Faxén, Wool - A work stealing library, SIGARCH Computer Architecture News 36 (5).

[5] Intel Threading Building Blocks, `http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf`.

[6] E. Perelman, G. Hamerly, B. Calder, Picking statistically valid and early simulation points, in: Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques, 2003.

[7] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, Statistical sampling of microarchitecture simulation, ACM Trans. Model. Comput. Simul. 16.

[8] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, D. M. Hawkins, Characterizing and Comparing Prevailing Simulation Techniques, in: Proc. of the 11th International Symposium on High-Performance Computer Architecture, 2005.

[9] A. Alameldeen, D. Wood, IPC Considered Harmful for Multiprocessor Workloads, IEEE Micro 26 (4).

[10] A. J. KleinOsowski, D. J. Lilja, MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research, IEEE Comput. Archit. Lett. 1 (1).

[11] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, D. J. Sorin, Simulating a $2M Commercial Server on a $2K PC, Computer 36 (2).

[12] J. J. Yi, D. J. Lilja, Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations, IEEE Trans. Comput. 55 (3).

[13] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, J. C. Hoe, SimFlex: Statistical Sampling of Computer System Simulation, IEEE Micro 26 (4).

[14] T. M. Conte, M. A. Hirsch, K. N. Menezes, Reducing State Loss For Effective Trace Sampling of Superscalar Processors, in: Proc. of the International Conference on Computer Design, VLSI in Computers and Processors, 1996.

[15] S. Dwarkadas, J. R. Jump, J. B. Sinclair, Execution-driven simulation of multiprocessors: address and timing analysis, ACM Trans. Model. Comput. Simul. 4 (4).

[16] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, M. Valero, On the simulation of large-scale architectures using multiple application abstraction levels, ACM Trans. Archit. Code Optim. 8 (4).

[17] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A distributed parallel simulator for multicores, in: IEEE 16th International Symposium on High Performance Computer Architecture, 2010.

[18] J. Chen, M. Annavaram, M. Dubois, SlackSim: a platform for parallel simulations of CMPs on CMPs, SIGARCH Comput. Archit. News 37.

[19] J. L. Hennessy, D. A. Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 2006.

[20] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability, 1967.

[21] A. C. Iordan, M. Jahre, L. Natvig, Towards Efficient Simulation of Task Based Parallel Application, in: Norsk informatikkonferanse, 2012.

[22] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, J. C. Hoe, Simulation sampling with live-points, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2006.

[23] K. C. Barr, H. Pan, M. Zhang, K. Asanovic, Accelerating Multiprocessor Simulation with a Memory Timestamp Record, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2005.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, The GEM5 Simulator, SIGARCH Comput. Archit. News 39.

[25] A. Duran, X. Teruel, R. Ferrer, X. Martorell, E. Ayguade, Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, in: Proc. of the International Conference on Parallel Processing, 2009.