

Adaptive Sampling for Efficient MPSoC Architecture Simulation

Melhem Tawk*, Khaled Z. Ibrahim⁺, Smail Niar*

*LAMIH, University of Valenciennes, 59313 Valenciennes Cedex 9, France

⁺Suez Canal University, Port-Said, Egypt

Abstract—Modern micro-architecture simulators are many orders of magnitude slower than the hardware they simulate. The use of multiprocessor architectures for supporting future mobile and embedded applications will exacerbate this slowness. In this paper, we focus on the usage of the sampling technique for simulation acceleration, in the case of design space exploration (DSE), considering MPSoC. Among the addressed issue is the formation of sampling intervals that are executed simultaneously by the different processors. We propose a technique that dynamically adjusts the size for simulation samples for multiprocessor activities overlaps. Experimental results show that with our method, the simulation can be speedup by a factor of up to 800 with a relatively small estimation error.

Keywords : Simulation, MPSoC, Acceleration, Sampling.

I. INTRODUCTION

Simulation plays an important role in the computer system design process. In embedded and mobile system area, where the time-to-market factor has a great importance, simulation technique has a vital position. In this paper, we focus on embedded system simulation, to explore the space of the possible system configurations at the micro-architectural level. This process is usually called *Design Space Exploration* (DSE). DSE involves evaluating performances (execution times) and power consumption of a large set of possible system configurations. The best configurations are then selected for a possible implementation in an ultimate design.

Nevertheless, the use of conventional cycle-accurate simulation tools in the case of *Multiprocessor System-on-Chip* (MPSoC) risk becoming problematic because of the increase in simulation time. Using our experimental MPSoC platform, we measured that even for embarrassingly parallel applications the simulation time can be multiplied by a factor of ten when the number of processors grows from 1 to 16. This increase is due to the architecture of the system which becomes more complex.

To tackle this problem, several techniques aiming at the reduction of simulation time have been proposed during the recent few years [2, 3, 4, 7, 8, 9, 10, 11, 12]. Application sampling is one of these techniques. In this approach one or several representative samples (or intervals) of the application are chosen. These samples are a subset of the

instruction count and represent the behavior of the whole application on a given hardware platform.

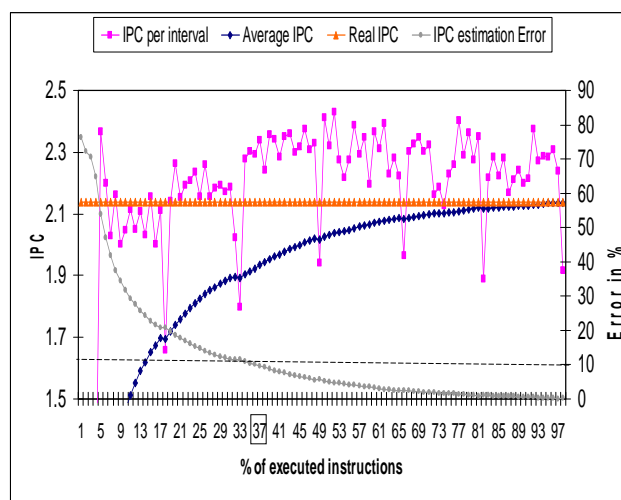


Figure 1. IPC per interval, Average IPC, Real total IPC and estimation error for fft on 4 processors

Figure 1, presents the advantage to use sampling for embedded application. In these experiments, we used an MPSoC configuration of 4 processors all running the fft benchmark. The interval length was set to a total of 50K instructions and we measured the *Instruction Per Cycle (IPC)* for each interval. That is, when the total number of instructions executed by all processors reaches 50K, the IPC for this interval is measured. The *average IPC* from the beginning of application till the corresponding point and the *real total IPC* are also given. This figure shows that: Firstly, the IPC per interval varies greatly and secondly simulating only one set of consecutive intervals of the application cannot produce an accurate estimation. For instance, to have an estimation error under 10%, at least 37% of the application needs to be simulated, giving merely a speedup of $1/0.37 = 2.7$.

While in the case of uniprocessor architectures, the application of sampling have been studied extensively, its use for multi-processor architectures spawns multiple problems. Among these problems, there is the determination of parallel phases executed simultaneously by

the different processors. In other words, while for uniprocessor architecture, phases to simulate can be determined *a priori*; this is no longer valid for multiprocessor architecture. The phase scheduling in the different processors can completely change from one configuration to another. Figure 2 illustrates this problem for two processors. In this figure, each application is decomposed into repetitive and similar phases of fixed length (c.f. section 4) [11]. The first application starts by alternating between two phases *a* and *b*, while the second application starts with a phase sequence *x x y y x z t*, where phases of the same symbol letter have similar behavior. While running each application individually (only one processor is active), intervals belonging to the same phase approximately require the same number of cycles (Figure 2.a). When these two applications coexist on two parallel processors (Fig. 2.b), phases affect each others. This leads to the deformation of phases.

Thus, it becomes impossible to determine *a priori* with which phase, in the other processor, a given phase will be executed. Due to shared resources (such as the shared bus and the shared memory), performances of the parallel applications are interdependent. This paper addresses the problem of *cycle-accurate bit-accurate* (CABA) simulation acceleration for MPSoC systems. It contributes to the existing work in the field by introducing a sampling technique that adaptively adjusts the size of the sample and the amount of instructions simulated in detail for the sake of maintaining low error in estimating the performance without tedious intervention of system designer.

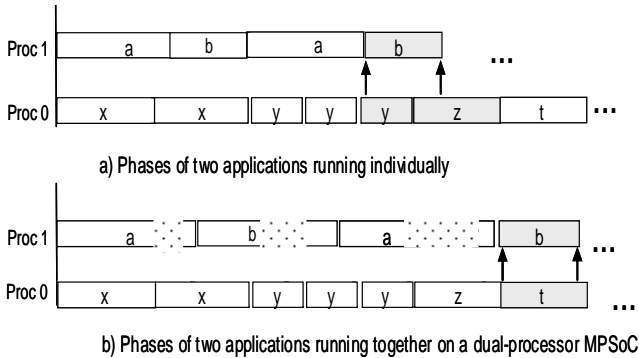


Figure 2. Simulation phases on a dual processor MPSoC. a) Individual execution: *b* seems to be in parallel with *y* and *z*. b) Simultaneous execution with possible contention (processor 1 is delayed in case of contention): *b* is in parallel with *t*.

The rest of this paper is organized as follows: Section 2 discusses related work and overviews phase clustering techniques for multiprocessor systems and contrasts them with our work. Section 3 details the sampling technique proposed in this study. Section 4 outlines the experimental setup. We conclude this paper in Section 5.

II. RELATED WORK

Exploration of large design space using simulation motivated many research proposals to address reducing the cost of simulation using sampling [2, 3, 4, 7, 8, 11]. Repetitive application phases are detected and performance is estimated based on sampling from different phases. SimPoint proposed by Sherwood et. al [11] provides a versatile tool to analyze applications. Application execution is divided into *intervals* either fixed [10] or variable number [6] of instructions. These intervals are examined to identify similarity in terms of the distribution of basic blocks. A group of similar intervals constitute a phase. Simulation acceleration is based on exploiting the fact that intervals from the same phase have almost identical behavior and simulating only one sample from each phase is sufficient for the whole application performance estimation. Recently, several works have been devoted to applying the sampling technique to *simultaneous multi-threaded* (SMT) architectures.

In [2] co-phase matrix has been used to sample all combination of phases (Figure 3.a). Phases partly overlap with different joint scenarios. Because phases are not generally homogeneous, overlapping the same couple of phases results in multiple performance estimates. Multiple samples of co-phase overlap must be simulated to achieve accurate performance prediction. The accuracy is dependent on the number of samples taken for each co-phase overlap entry. For a given simulation accuracy, it is difficult to determine the number of samples needed *a priori*.

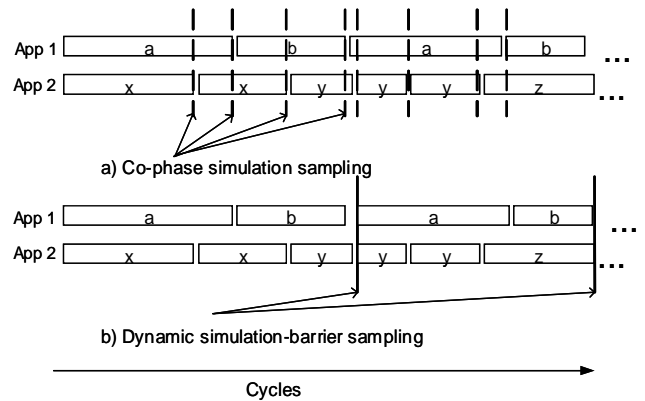


Figure 3. Simulation phases on a dual processor MPSoC. (a) Co-phase approach with multiple performance estimate per phase (b) Dynamic sampling with one performance estimate for each phase cluster.

MPSoC architectures differ from SMT at several levels: First, in their DSE, MPSoC involve a greater number of parameters to explore. Consequently, the DSE for these platforms requires a higher simulation acceleration factor. Second, the resource sharing degree in MPSoC is relatively lower in MPSoC than in SMT. These features allow to design different simulation acceleration methods.

Our proposal in contrast, dynamically forms strings of phases from each processor such that these strings of phases overlap sharply (figure 3.b). In this way, we obtain a good phase string boundary alignment. The string length is dynamically adapted to achieve overlap that has a repeating performance thus alleviating the need for multiple sampling.

Kanaujia et al. [7] propose a multi-core simulation methodology for homogeneous applications that does not share data. Simulation speedup is achieved by implementing detailed simulation of only one core and approximating the traffic from the remaining cores by a traffic analysis using a transactional module.

Namkung et al. [9] experienced the application of co-phase for multi-processor architectures with a large number of processors. They noticed that simulation speedup diminishes due to a rapid increase in the number of phase combinations. To reduce the number of simulation samples, they proposed synthesizing samples from similar phase combinations. Del Valle et al. [4] propose to speedup simulation by using FPGA-based emulation framework. Their approach enables a rapid extraction of a large range of statistics for three MPSoC components, i.e. processing cores, memory and interconnection. This approach demonstrated high accuracy and simulation speedup but requires expensive specific platforms (FPGA).

III. DYNAMIC ADAPTIVE SAMPLING FOR MPSOC

In this study, we target design space exploration for MPSoC, where multiple configurations are examined through simulation. We propose to use a sequence of phases (called *phase string*) for each application such that they overlap perfectly, as shown in Figure 3.b. These strings are separated by *simulation barriers*. The performance is estimated through simulation of overlaps of phase strings (called *clusters of strings CS*) that did not show up in simulation earlier. The proposed scheme comprises the following advantages:

- Formation of strings of phases is done dynamically. The system designer merely specifies a maximum acceptable wait time at the simulation barrier.
- The overlap of *phase strings* has a repetitive performance which alleviates the need for multiple sampling for the same overlap

The dynamic adaptation technique for forming *phase strings*, as detailed in the following section, has the potential for achieving reasonable accuracy across different design space configurations without the need of frequent tuning of simulation sampling percentages. Effectively, the sampling percentage is determined dynamically based on the number of phase strings that arise during simulation and not based on static sampling percentage that is determined a

priori. This dynamic adaptation of sampling is a key simplification in design space exploration that is needed most in enhancing time-to-market of embedded applications. In the next section, we detail our proposed technique to accelerate simulations.

A. First step: program tracing and phase identification

In this stage, we generate a phase-ID trace for each application individually. The obtained traces are neither dependent on the execution of the other processors which execute on parallel nor on a specific MPSoC architectural configuration. It depends solely on the application and its input data. Thus, this step is accomplished once for all the evaluated configurations in the DSE. Moreover, as a rapid functional simulation is used here, its execution time is relatively short. In a few minutes, a processor phase trace is obtained. Table 1 gives an example of two trace files generated in this step for an MPSoC of 2 processors (P0 and P1). In this table, the interval size has been set to 50K instructions.

TABLE 1
PHASE'S TRACE EXAMPLE FOR TWO PROCESSORS GENERATED BY PROFILING THE APPLICATIONS

Interval (in K Instructions)	P0 phases	P1 phases
0-50	a	x
50-100	b	y
100-150	c	z
150-200	d	w
200-250	a	w
250-300	b	x
350-400	f	y
...	...	z

B. Second step: generation and utilization of clusters

The second step uses all the phase traces that have been generated in the first step. For each processor, consecutive phases are combined together to form a *phase string*. The number of phases within the string is determined dynamically as will be detailed in the next subsections. A *cluster of strings (CS)* is generated dynamically and consists of P parallel phase strings, where P is the number of processor cores in the MPSoC. Each new CS is allocated an entry in the *cluster of strings table (CST)*. The key idea of our approach is that, clusters containing the same parallel phase strings will have the same behavior and thus will give the same performance. Table 2 summarizes the function of each of the above steps. The second step is further detailed in the following subsections.

B.1. Using simulation barriers for CS generation

At the end of every simulation interval, each processor estimates the maximum remaining cycles such that all other processors finish their intervals in progress.

As shown in Figure 4, the estimated remaining cycles of each processor depends on its IPC at this point of simulation and the total number of cycles (C) since the beginning of the CS. Due to the difference in the IPCs, the estimated remaining cycles (denoted as ΔC in Figure 4) will be different from one processor to the other. Thus, if the remaining cycles over the cycles of execution, is less than a given threshold for all processors, the processor stops simulation and waits at the simulation barrier. This threshold is denoted by TWSB for *Threshold Waiting at Simulation Barrier*. Otherwise, it proceeds with executing the following interval. If a processor decides to wait, this corresponds to a simulation-barrier and all the other processors will be obliged to stop simulation at the end of their current intervals. Pseudo-code for the proposed synchronization technique is shown in Figure 4.

TABLE 2
ADAPTIVE PHASE SAMPLING

1. **First step:** generate application individual phase-ID trace file using the SimPoint profiler.
2. **Second Step:** For each already simulated CS in CST, determine if it corresponds to the next set of strings in the phase's trace generated offline. Once a match is found the search is stopped.
3. If a match of cluster of strings (i.e. CS) is found in the CST, the simulation is fast forwarded and the performance is estimated based on the found match.
4. Else the simulation is performed with dynamic termination of cluster.
 - a. During the simulation, at the end of each interval, we check if it is possible to combine the strings of phases executed in parallel till that point in order to generate a new CS using simulation barrier criteria.
 - b. Upon termination, a new entry is allocated in the CST and performance is recorded.
5. Repeat the second step until end of phase-ID traces

In this figure *Interval* corresponds to interval length in instructions, I_{proc} and IPC_{proc} correspond respectively to the number of executed instructions and IPC for the processor $proc$ for the current interval. These statistics are available in the simulator at run time.

At the simulation barrier, we combine the strings of phases executed in parallel. Due to the waiting state of some processors at barriers, phase scheduling may be slightly different from the real situation (without barriers). This shift in scheduling can cause an estimation error additional to that due to phase classification in step 1. As the IPCs for each application executing in parallel are not the same, the number of intervals executed by each processor in a cluster can be different.

Figure 4 shows that the processors test at the end of each interval the constraint of the threshold. For instance, at the end of the interval b , P0 decides to wait, while at the end of interval a , x , and y the system decides to continue simulation.

B.2. Generating clusters of strings

As shown in figure 4, CSs are created dynamically depending on the way each processor interacts with the shared resources. CSs are separated by simulation barriers and each string of phases corresponds to phases that are executed by the corresponding processor.

```

 $\Delta C = 0$ 
for  $proc = 1$  to  $P$ 
  if ( $proc \neq my\_proc$ )
     $\Delta C = \max(\Delta C, (Interval - I_{proc}) / IPC_{proc})$ 
  Endif
Endfor

If  $\Delta C / C < TWSB$ 
  Start barrier synchronization for all  $proc$ 's
Endif

```

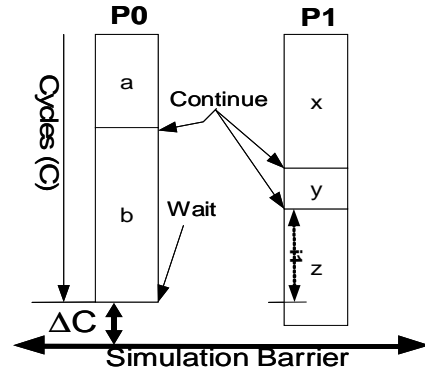


Figure 4. Simulation barrier for CS generation in step 4.a of TABLE 2.

The combination of phase's identifiers, which are executed in parallel, represents a unique CS identifier. Table 3 shows possible CSs for the application phases given in Table 1. The first generated CS is denoted as $\underline{a,b-x,y,z}$ and contains two strings of phases: a,b (for P0) and x,y,z (for P1). Two clusters of strings which contain the same phase strings will have the same performance.

As soon as a new CS is simulated, a new entry is added to the CST and its performance metrics (such as the number of *cycles* and the total consumed *energy*) are recorded in this table.

B.3. Using CST for simulation acceleration

As shown in table 2 (step 2), if a match is found between a CS in the CST and the next phases in the traces, the CS is skipped and the corresponding entry in CST is updated. The number of instructions that each processor must be fast-forwarded is also taken from the CST (see table 3). In this case a rapid functional simulation is used to move the program context to the end of the CS. Otherwise, if no match is found, then detailed CABA simulation is performed until a termination of the new CS using simulation barrier.

“Cold Start Hit” is used as warmup technique to start detailed simulation. A *warmup bit* is added to each cache block to signify the first time an entry is addressed. When starting the simulation of a CS, the first access to a cache block is assumed to be a hit. Since the miss rates for our benchmarks are generally very low, this simple method provides interesting performance.

In table 1 and table 3, after simulating the first two CSs, the cluster $a,b-x,y,z$ recurs. This cluster already exists in the CST so it is fast-forwarded by a rapid functional simulation. Thus P0 will be fast forwarded 100K instructions ($2*50K$) and P1 will be fast forwarded 150K ($3*50K$). We estimate the performance of the whole application based on CS information and their relative contribution to the whole program.

TABLE 3
CST CONTENT FOR THE EXAMPLE IN TABLE 1

Cluster	Inst. count P0 in Kinst.	Inst. count P1 in Kinst.	Cycles	Energy	Repetition
a,b-x,y,z	100	150	200	100	2
c,d-w,w	100	100	300	150	1

IV. EXPERIMENTAL RESULTS

To evaluate the usefulness of our adaptive phase sampling method for MPSoC DSE, several experiments have been conducted. For this purpose, five different benchmarks taken from the MiBench suite [5] and the H264 have been ported to our MPSoC platform.

Table 4, gives for each benchmark, the number of instructions per application and the number of phases found in the profiling (step 1 of table 2).

TABLE 4
BENCHMARK DETAILS

Category	Benchmark	Version	Number of instructions per application	Number of phases
Telecommunication	FFT		168 008 964	10
	FFT	Inverse	184 626 007	10
	Gsm	Encode	1 251 462 740	8
	Gsm	Decode	536 235 207	5
	Adpcm	Encode	949 890 274	4
	Adpcm	Decode	607 441 786	8
	H264		129 204 031	10
Security	Rijndael	Encode	332 082 997	3
	Rijndael	Decode	349 339 153	2
	Blowfish	Encode	311 616 420	2
	Blowfish	Decode	314 864 383	2

For each of these benchmarks, both the “encode” and the “decode” version of the benchmarks have been used. During the experiments, the number of processor cores has been varied from 2 to 12. Except for H264, where all the

processor execute the same application, for the other benchmarks half of the processors execute the encode version while the other half execute the decode version of the application. It is worth noting here that, sampling for independent applications is more difficult than for dependent (or communicating) applications, as in the former case communications and synchronizations help in determining the parallel phases and make sampling easier.

The simulation framework for these experiments is based on MPARAM [1]. This framework consists of ARM7 cores connected to shared RAM modules via a standard shared AMBA bus. All these components are described at cycle-accurate and bit-accurate (CABA) level using SystemC. The processor configuration is shown in Table 5.

We define *simulation speedup* as the ratio between the total number of instructions executed by all the processors over the number of instructions simulated in the detailed mode (or CABA mode).

TABLE 5
MPARM PROCESSOR CONFIGURATION

I-cache	8KB direct mapped, 1 cycle latency
D-cache	4KB 4-way set-associative, 1 cycle latency
Memory	64-cycle latency
Core	Arm version 7

During the experiments, we used 50K instructions as interval size. With such small size interval there is a high probability of detecting the phase behavior of the applications, leading to improved acceleration. The influence of the interval size on the method performances is left as a possible future extension.

When the total detailed CABA simulation is applied on a benchmark each run requires several hours of simulation, depending on the MPSoC configuration. For instance, a total detailed simulation with 8 processors needs about three days to finish on a 3GHz Pentium4 platform.

A. Effect of TWSB on simulation acceleration

Decreasing the value of TWSB decreases the probability for a given processor to generate a simulation barrier. As a consequence, the size of clusters in terms of number of phases increases. As a result, clusters become larger and the occurrences of the same CS decreases. Accordingly, simulation speedup decreases and estimation error decreases, as less waiting cycles are injected. In other words, TWSB controls the tradeoffs between estimation accuracy and simulation acceleration.

Figure 5 shows the simulation speedup and the estimation error for the six applications as a function of TWSB when the number of processor varies between 2 and 12 processors. For each application, the acceleration factor consistently increases with the value of TWSB. The three applications blowfish, rijndael and H264 have the best acceleration factor. Blowfish and rijndael exhibit very high

speedup factors (respectively 783 and 280) as these benchmarks generate small number of phases (2 for blowfish and 3 for rijndael).

In the case of H264, the same application is duplicated so processor workloads are almost similar. The generated CS for this application contains one phase per processor. During simulation, we observed 8 different CS which corresponds to the number of phases in H264. Accordingly the acceleration factor is constant for the entire configurations.

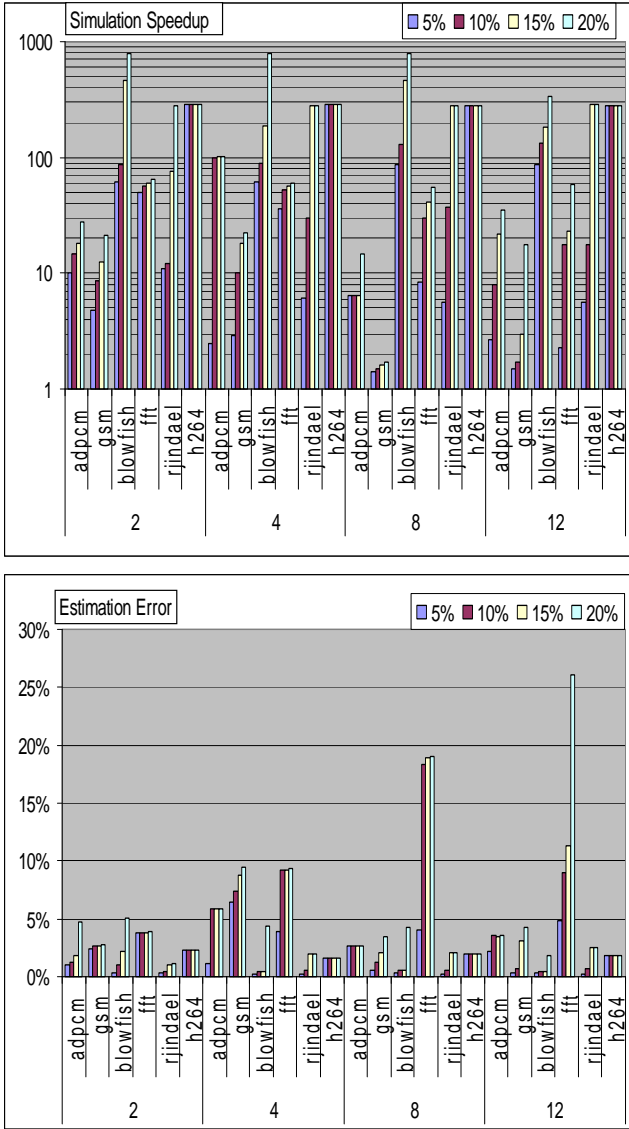


Figure 5. Experimental results with 6 benchmarks. The number of processor varies from 2 to 12 and TWBSB varies from 5% to 20%. Upper part gives the speed up. Lower part gives the corresponding estimation error.

The speedup factor of gsm using 8 processors decreases to 1.5. The reason of this weak performance is the high

number of phases and the aperiodicity of gsm decode and uncode. Consequently, the average size of clusters is very important with small number of repetition. A TWBSB of 60% for gsm on 8 processors (not shown in the figure) has allowed to reduce the average CS size and to increase the speedup to 97.7 without sacrificing the precision estimated (error is almost 4.5%). An automatic setting of TWBSB, according to the application behavior, is proposed as future extension of the work.

B. Effect of TWBSB on performance estimation

The error in IPC estimation is actually related to two sources of error. The first source is due to associating the IPC of each repeated cluster with the IPC of the first cluster that have been already simulated. Intervals identified as belonging to the same phase (using SimPoint) may have different performances.

The second source of error is the utilization of simulation barriers. As explained in section III, during simulation our method injects waiting cycles in the different processors before synchronization barriers. These wait cycles cause two problems: first, the estimated IPC is lowered because no workload instructions are committed during the wait; second, the overlap between the applications is slightly different from the overlap without simulation barriers.

Figure 5 (lower part) shows the error in estimating IPC as a function of the four TWBSB values for our six duplicated applications using up to 12 processors. We observe that for each architectural configuration, the estimation error increases with the TWBSB. In fact, when TWBSB increases, the number of simulation barriers and the number of injected waiting cycles increase.

In the case of *H264*, the error is constant with the variation of TWBSB on all the configurations because the total number of observed CS does not change with TWBSB variation, as explained in the previous subsection. We have the same situation for *fft*, the error stabilizes when the value of TWBSB is greater than 10%. When the TWBSB increases from 5% to 25% the *fft* error increases from 4% up to 9% on 4 processors, from 4% up to 18% on 8 processors and from 5% up to 26% on 12 processors.

For rijndael, the error varies slightly from 0% up to 1% when the TWBSB increases. As previously stated, this benchmark has a high regularity degree between intervals and thus generates a small number of phases. We observe that CSs have almost very close IPCs values. Due to lack of space, power consumption estimation errors are not shown here. Nevertheless, these errors are much smaller than errors for the IPC.

Our experiments indicate that using simulation barriers, the amount of detailed simulation needed is adjusted adaptively based on application's dynamic. For example on a 12 core configuration and a TWBSB of 5% (figure 5), the error percentage in estimating the IPC is less than 5%, while the detailed simulation ranges from 0.36% (or 1/280)

for H264 to 55% (or 1/1.8) for adpcm. In contrast, other sampling techniques suffer either from lost opportunities for saving if they set the sampling percentage to a fixed high value, or less estimation accuracy if they set the sampling percentage to a fixed low value.

V. CONCLUSION

In this work, we introduce a technique to adaptively adjust sampling intervals and sampling amount in order to maintain a low estimation error. We tackle the problem of variability in estimating performance due to the different scenarios of phase overlaps. In this proposal, we form strings of phases such that overlaps from the running applications superimpose with small difference in alignment boundaries. We simulate only new overlap scenarios.

We conducted simulations that show that our proposed technique adapts the amount of sampling as well as parts of the application to simulate based on the application's dynamic nature. In almost all the cases, the error percentage in estimating the IPC is kept low. For applications with a large number of phases and/or large differences of IPC between running applications, our scheme enlarges the amount of detailed simulation, and vice versa, in order to achieve an accurate estimate of performance. Experimental results show that the proposed method is able to produce a high simulation speedup making it highly applicable for large DSE in embedded systems. Future research will focus on several areas. First, we propose an automatic setting of TWSB. Using this technique, TWSB value will be varied function of the desired speedup, to provide the smallest possible estimation error. Second, more complex processor and network architectures will be integrated into the platform.

REFERENCES

- [1] Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing* 41, 2005.
- [2] Biesbroucky, M. V., Sherwood, T., and Calder, B. A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. *Symp. on Performance Analysis of Systems and Software*, 2004.
- [3] Biesbroucky, M. V., Eeckhout, L., and Calder, B. Considering All Starting Points for Simultaneous Multithreading Simulation. *Symp. on Performance Analysis of Systems and Software*, 2006.
- [4] Del Valle, P. G. et Al., Architectural Exploration of MPSoC Designs Based on an FPGA Emulation Framework, *Conf. on Design of Circuits and Integrated Systems* 2006.
- [5] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite, *Workshop on Workload Characterization*, Dec. 2001.
- [6] Lau, J., Perelman, E., Hamerly, G., Sherwood, T., and Calder, B. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *Symposium on Performance Analysis of Systems and Software*, Mar 2005.
- [7] Kanaujia, S., Papazian, I., Chamberlain, J., Baxter, J. FastMP: A Multi-core Simulation Methodology, *MOBS* 2006.
- [8] Ekman, M., Stenstrom, P.; Enhancing Multiprocessor Architecture Simulation Speed using Matched-Pair. *Symp. on Performance Analysis of Systems and Software*, 2005.
- [9] Namkung, J., Kozintsev, I., and Dulong, C. Phase Guided Sampling for Efficient Parallel Application Simulation. *Conference on Hardware/software codesign and system synthesis*, Oct 2006.
- [10] Nussbaum, S., and Smith, J. E. Statistical Simulation of Symmetric Multiprocessor Systems. *Simulation Symposium*, 2002.
- [11] Sherwood, T., Perelman, E., Hamerly, G., Calder, B. Automatically Characterizing Large Scale Program Behavior. *Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.
- [12] Wunderlich, R. E., Wensich, T. F., Falsafi, B., and Hoe, J. C. Smarts: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *ISCA* June 2003.