A Novel Evaluation Methodology to Obtain Fair Measurements in Multithreaded Architectures

Javier Vera[1] Francisco J. Cazorla[1] Alex Pajuelo[2]
Oliverio J. Santana[3] Enrique Fernandez[3] Mateo Valero[1,2]

[1]Barcelona Supercomputing Center, Spain. {javier.vera,francisco.cazorla}@bsc.es
[2]DAC, Universitat Politècnica de Catalunya, Spain. {mpajuelo,mateo}@ac.upc.edu.
[3]Universidad de Las Palmas de Gran Canaria, Spain. {ojsantana,efernandez}@dis.ulpgc.es

ABSTRACT

Nowadays, multithreaded architectures are becoming more and more popular. In order to evaluate their behavior, several methodologies and metrics have been proposed. A methodology defines when the measurements of a given workload execution are taken. A metric combines those measurements to obtain a final evaluation result. However, since current evaluation methodologies do not provide representative measurements for these metrics, the analysis and evaluation of novel ideas could be either unfair or misleading. Given the potential impact of multithreaded architectures on current and future processor designs, it is crucial to develop an accurate evaluation methodology for them.

This paper presents FAME, a novel evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME reexecutes all threads in a multithreaded workload until all of them are fairly represented in the final measurements taken from the workload. We compare FAME with previously used evaluation methodologies in architectural research scenarios. Our results show that FAME provides the most accurate measurements, becoming an ideal evaluation methodology to analyze novel design ideas implemented in multithreaded architectures.

1. INTRODUCTION

Recent technology advances have increased the number of available transistors for processor designers. However, the performance achievable by traditional superscalar processor designs has almost saturated due to the limitation imposed by instruction-level parallelism. As a consequence, computer architects have looked for new ways to use these available transistors in order to exploit more parallelism.

Thread-level parallelism has become a common strategy for improving processor performance. Since it is difficult to extract more instruction-level parallelism from a single program, multithreaded processors rely on using the additional transistors to obtain more parallelism by simultaneously executing several tasks. This strategy has led to a wide range of multithreaded processor architectures, from simultaneous-multithreaded processors (SMT) [13][14][21], in which most processor resources are shared among threads, to chip multiprocessors (CMP), in which every thread has their own dedicated processor resources, only sharing the highest levels of the memory hierarchy. Recent evolutions of these architectures have also generated CMP/SMT processors, *i.e.*, chip multiprocessors in which every core is a SMT [17].

To design these processors, the first steps commonly involve using simulation tools [6][21] to model their expected behavior. These simulators allow researchers to propose and test novel techniques that could be included in the final processor design.

In order to evaluate these new techniques, computer architecture researchers use benchmark suites [1][2], since they are representative of current and future applications that will be executed by the designed processor. In spite of the increasing trend to use truly parallel applications, they are currently less common in real machines than non-cooperative single-threaded applications. Therefore, computer architecture researchers frequently evaluate multithreaded processors using workloads composed by non-cooperative single-threaded applications, picked up from a benchmark suite, which perform non-related work and do not communicate each other.

However, as the complexity of the simulated processor grows, the simulator also becomes more complex, increasing the time required for completing benchmark simulations. As a consequence, the amount of time required to simulate a whole benchmark becomes unaffordable. The most common approach to reduce simulation time is to select a smaller segment of every benchmark that is representative of the whole benchmark execution [8][10][16][23]. This representative segment (from now onwards we will call it *trace*) will be used to feed the simulator with the data required to evaluate the processor model. Actually, traces can have many different forms. Traces containing the mainstream of the execution segment and the corresponding memory addresses can be as effective as traces containing just the data needed to initialize the architectural state of the processor (checkpoint) and restart execution at that point.

The generation of representative traces is an excellent way to reduce simulation time in traditional single-threaded superscalar processors. Nevertheless, using those single-thread traces in multithreaded processors is not straightforward. Multithreaded processors are able to execute several tasks in parallel, using separate hardware structures called thread contexts to physically keep all the data required and produced by every task during execution. As a consequence, combinations of traces must be considered to create execution workloads.

Working with several traces at a time involves an important decision, that is, to determine when simulation will finish. In a single-threaded processor, the simulator runs the full trace until completion. However, it is not so easy in a multithreaded processor simulator running a workload composed by several traces. Traces in a workload can execute at different speeds due to the different features of each program, as well as the availability of the shared resources. Therefore, they do not have to necessarily complete execution at the same time. We will explain this fact with an example. Let us assume a M-context multithreaded proces-

sor executing a 2-thread workload (being M greater than or equal to 2). The execution of this workload occurs as depicted in Figure 1. Both threads execute at different speeds and thus they do not have to finish at the same time. Therefore, we can divide the execution of the workload into two phases. Firstly, there is a *multithreaded period* in which both threads are being executed. Secondly, after the first thread finishes (Thread 0 in Figure 1), there is a *single-threaded period* in which the remaining thread executes alone until completion. If the multithreaded period is too short, then the potential of the multithreaded processor is only exploited during a small interval of time. As a consequence, the *total execution time* becomes an inaccurate metric for multithreaded processors. We have found that, when executing 2-thread workloads composed by SPEC benchmarks, a 2-context SMT simulator spends almost one third of the time executing a single thread.

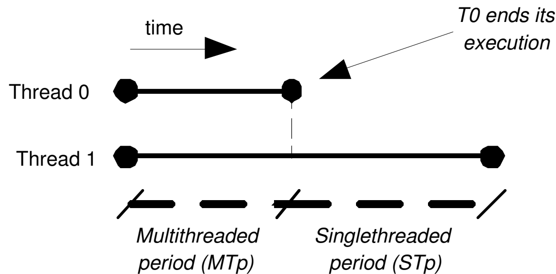


Figure 1: Execution of a 2-thread workload in a M-context multithreaded processor ($M \geq 2$).

In general, the execution of an N-thread workload [1] involves N periods of N, N-1, N-2,... and 1 thread respectively. Only measurements obtained from the period with N running threads are representative. Periods having less running threads should not be taken into account since the results could be inaccurate and misleading [2].

In this paper, we analyze several simulation methodologies that have been used to face this problem during the last years. These methodologies suggest how simulation should be performed and, in particular, they determine when workload simulations have to finish. However, we show that these methodologies are not completely fair, since they cannot ensure that the trace of every benchmark is fully executed, and thus it is not possible to assure that the measurements obtained are representative of the whole program behavior.

To face this problem, we present FAME, a new simulation methodology for the evaluation of multithreaded processors. Our methodology aims to ensure that a representative trace of every benchmark in a workload is executed, allowing to do fair comparisons between different techniques and processor setups. FAME can be used with any of the state-of-the-art tools for obtaining a representative trace of a given program [8][10][16][23]. As a case study, we have selected to apply FAME to SMT processors, modeled with the SMTsim simulation tool, using the SPEC benchmark suite to create workloads. Nevertheless, there is no loss of generality. FAME is applicable to any multithreaded architectural design, since all them present identical evaluation problems. Overall, our results show that FAME provides more accurate measurements than previously used methodologies.

[1]We assume that the number of threads in a workload is smaller than or equal to the number of available hardware contexts in a multithreaded processor.

[2]This is a common characteristic of all the evaluation methodologies.

The structure of the paper is as follows. Section 2 describes our evaluation environment. Section 3 analyzes the most commonly used methodologies, exposing their major drawbacks. The FAME methodology is described in Section 4. Section 5 compares FAME with the current methodologies described in Section 2. The related work is referenced in Section 6. Finally, we present our concluding remarks in Section 7.

2. EXPERIMENTAL ENVIRONMENT

This section describes the research scenario in which existing evaluation methodologies as well as FAME will be compared. As previously said, this is a case study. FAME can be applied to any multithreaded design.

2.1 Simulation Framework

To evaluate FAME in a state-of-the-art experimental environment, we use an SMT simulator derived from smtsim [21] (see configuration parameters in Table 1). As an illustration of applicability of the FAME methodology we have selected two well-known fetch policies: *icount* [21] and *stall* [19]. The icount fetch policy prioritizes those threads with fewer instructions in the processor pipeline. The stall fetch policy uses the same heuristic, but it also detects whenever a thread has a pending long-latency memory access. When this situation is detected, stall prevents the thread from fetching more instructions until the memory access is resolved, avoiding unnecessary over-pressure over the shared resources.

Table 1: Baseline configuration.

Parameter	Value
Pipeline depth	12 stages
Number of contexts	2 and 4
Default fetch policy	stall
Fetch/Issue/Commit Width	8
Queue Entries	80 int, 80 fp, 80 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	320 integer, 320 fp
(shared)ROB size	512 entries
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way assoc.
Return Address Stack	256 entries
Icache, Dcache	64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	2048 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access
Main memory latency	300 cycles
TLB miss penalty	160 cycles

We feed our simulator with traces collected from the whole SPEC2000 benchmark suite [2] (excluding *facerec*, *fma3d* and *sixtrack*, since we were unable to collect traces for them) using the reference input set. Benchmarks were compiled with the Compaq/Alpha C V5.8-015 compiler on Compaq UNIX V4.0 with all the optimizations enabled. Each trace contains 300 million representative instructions, which were selected using the SimPoint [15] tool to analyze the distribution of basic blocks. Using these benchmarks, we have generated 2-thread workloads with all the possible 2-thread combinations, leading to a total number of 276 workloads. Note that, if a workload is composed by threads A and B, since the behavior or a thread is independent of the context in which it is executed, the workload with threads B and A is not generated.

3. EVALUATING MULTITHREADED PROCESSORS

Measuring the performance of multithreaded processors is a difficult task. As a consequence, several *methodolo-*

Table 2: A possible classification of current simulation methodologies.

Finaliz. Moment → Trace duration ↓	First	Last	Fixed Instructions		
			100 mill	200 mill	1 bill
Fixed Length	[7]				
Variable Length	X	[24]	[22]	[11]	[9]

gies and *metrics* have been proposed in the literature. A methodology defines how simulation is performed and when the measurements are taken. Later, a metric combines those measurements to obtain a final result of the performance of the evaluated processor. Popular metrics include the IPC Throughput, the Weighted Speedup [18] and the Harmonic Mean [12]. The final result is commonly based on two inputs: the IPC achieved by each thread in a workload and the IPC of each thread when it is run in isolation.

Two main parameters define the behavior of a simulation methodology. These parameters are the *trace duration* and the *finalization moment*.

Trace duration: Researchers frequently use the SimPoint tool [15] to select a representative trace of S instructions from the whole program. We differentiate two kinds of traces, *fixed length traces* and *variable length traces*. If we use a fixed length trace and, when running a multithreaded simulation, it is required to execute more than S instructions, the trace is re-executed from the beginning. If we use a variable length trace schema, instructions beyond the trace of S instructions are executed as needed until the workload simulation ends. The first drawback of this strategy is that it is not possible to know beforehand the total number of instructions to execute beyond S, since it depends on the processor setup and the other threads in the workload. Therefore, a completely accurate upper bound of the number of required instructions cannot be obtained. A second drawback is that there is no warranty that the instructions after the interval provided by SimPoint are representative of the program. That is, if we start from the initial instruction given by SimPoint and we execute more than S instructions, there is no warranty that the executed trace is representative of the original program. Due to these two drawbacks, we use fixed length traces in our study, which is according to the SimPoint philosophy.

Finalization moment: In order to fairly evaluate the performance of an SMT processor, measurements should be obtained while all threads in a given workload are running. However, the threads in a workload can be executed at different speeds, and thus they do not have to finish at the same time. Consequently, the evaluation methodology should determine what to do whenever any thread finalizes its execution. All current simulation methodologies can be classified based on the finalization moment. This classification, shown in Table 2, includes the *First* methodology, the *Last* methodology, and the *Fixed Instructions* methodology.

3.1 Current Evaluation Methodologies

The First methodology finalizes workload simulation when any thread of the workload ends its execution [7]. The main drawback of this methodology is that only one thread in the workload is executed until completion, and thus it cannot be ensured that the remaining threads execute completely, losing representativity in the final result.

The Last methodology finalizes workload simulation when all the traces have been run until completion. When any thread, excluding the last one, finishes its execution, it can either reexecute (fixed length traces) or continue execution beyond that point (variable length traces) [24] while the other threads are still executing. The main drawback of this methodology is that the total number of evaluated instructions can vary from an evaluation to another one. Since the execution speed of the different threads depends on the processor parameters, any variation can cause all threads to be executed at different speeds. As a consequence, it cannot be ensured that the amount of executed instructions is the same for different simulations with different parameter values, and thus comparisons between them may be inaccurate.

The Fixed Instructions methodology is based on the idea of executing the same amount of instructions in every simulation. The simulation finalizes whenever the total number of executed instructions reaches a fixed threshold. This threshold is usually determined per thread, that is, the simulation of a workload with N threads will finalize when the total number of executed instructions is N times the threshold. Typical values for this threshold range from 100-million instructions [22] and 200-million instructions [11] to 1-billion instructions [9]. However, the Fixed Instructions methodology is also unable to ensure that a representative part of every benchmark is being executed, since workload simulation ends in an arbitrary point (whenever the total number of executed instructions is reached). Even worse, despite the total number of instructions is the same, the mix of executed instructions may change. As an example, imagine that two different instruction fetch policies must be compared, IF1 and IF2 in a 2-context SMT processor. IF1 always prioritizes instructions belonging to the first context and IF2 always prioritizes instructions belonging to the second one. The simulation finishes when N instructions from both threads are executed. When both simulations end, they have executed the same number of instructions but these instructions are not the same: most instructions belong to the first thread for IF1 and most instructions belong to the second thread for IF2. Therefore, since the executed instructions are not the same, the comparison between IF1 and IF2 is not fair regardless of the metric used.

3.2 Analysis of Current Methodologies

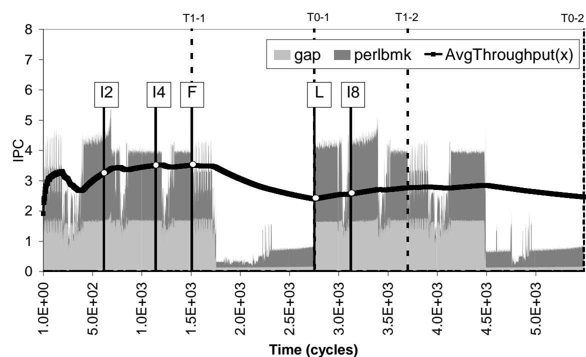
To show the behavior of current evaluation methodologies, we analyze the three most currently used methodologies for evaluating the performance of multithreaded processors: First (F), Last (L), and Fixed Instructions (I). We analyze three versions of the latter: 200-million fixed instructions (I2), 400-million fixed instructions (I4), and 800-million fixed instructions (I8).

As an example, Figure 2 shows the obtained results for these methodologies using our SMT simulator configuration and a 2-thread workload composed by the benchmarks *perlbmk* and *gap*. The simulation ends when both threads have executed at least twice.

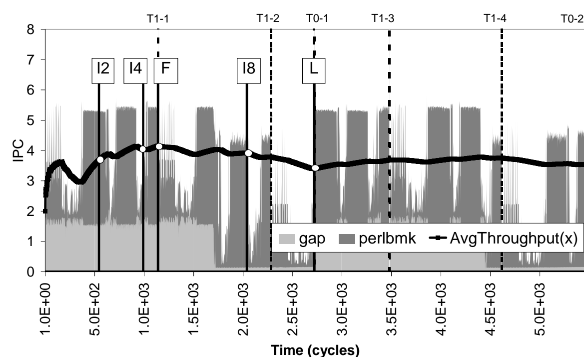
We provide data for two different fetch policies: the icount policy [21] in Figure 2(a) and the stall policy [19] in Figure 2(b). In Figure 2, the y-axis shows processor performance (IPC) and the x-axis represents execution time. The light-grey bars show the instant IPC of *gap*, that is, the IPC achieved by this benchmark each cycle. Likewise, the dark-grey bars show the instant IPC of *perlbmk*. In every cycle, the sum of both bars represents the instant throughput, *i.e*, the sum of the instant IPC of both threads.

Furthermore, the black horizontal line represents the average instant throughput until a time instant, that is, the average value of the instant throughput for every cycle from the beginning of the workload execution until the current time instant. The white circles over the black line show the final throughput reported by every methodology.

Finally, the vertical dashed lines show the time instant at



(a) Results with icount



(b) Results with stall

Figure 2: IPC of *gap* and *perlbmk* when executed together on the SMT simulator.

Table 3: Improvement of stall over icount with the different methodologies.

Methodology →		I2	I4	F	L	I8
IPC Throughput	icount	3.25	3.51	3.51	2.40	2.56
$IPC_{gap} + IPC_{perl}$	stall	3.68	4.04	4.15	3.40	3.92
stall Improvement(%)→		13.09	15.11	18.22	41.82	53.08

which every instance of a thread finishes. Above each line we add a legend in the form $Tx - y$, in which x indicates the thread and y indicates the number of times thread x has been executed. The vertical solid lines show the cycle in which the workload simulation ends according to each experimental methodology.

The main observation that can be drawn from Figure 2 is that every methodology provides different throughput values. It is summarized in the second (icount) and third (stall) rows of Table 3. It should be taken into account that researchers use simulation to evaluate the performance of a design enhancement relative to a baseline design. In the experiment of Figure 2, we can measure the performance improvement of stall with icount as baseline (shown in the last row of Table 3). Although stall improves the performance of icount for all methodologies, the speedup varies depending on the methodology used. If the $I2$ methodology is used, stall only achieves 13% performance improvement. But if measurements are taken using the $I8$ methodology, stall performance improvement arises to 53%. That is, depending on the evaluation methodology the stall improvement over icount varies up to 40%. Such a wide range of variation makes difficult to estimate the impact of any proposal and may cause misleading conclusions when a multithreaded processor enhancement is evaluated.

As discussed in previous sections, this problem is due to the fact that current methodologies cannot ensure fully representativity of every thread of the workload, which can lead to unfair comparisons between different simulator setups. Table 4 summarizes these drawbacks by showing the number of times every thread has been completely executed and the percentage of instructions executed in the last repetition for each methodology when using the stall fetch policy (results for icount are similar). The total amount of executed instructions varies from one evaluation methodology to another one. For example, in the case of the $I8$ methodology, T0 executes once completely and then executes 60% instructions from a second repetition. The same happens with T1, but in this case the percentage of instructions executed in the second repetition is 77%. Another example is the L methodology: T0 executes once and T1 execute once and 63% of the second repetition. This data clearly shows

Table 4: Number of full executions for each methodology and percentage of instructions executed of the current execution.

	Thread number	Methodology I2	I4	F	L	I8
Number of full executions	T0	0	0	0	1	1
	T1	0	0	1	1	1
% of instructions in current execution	T0	26	61	82	0	60
	T1	36	75	0	63	77

that the mix of instructions is different in every case what could make the comparison of results misleading.

4. THE FAME METHODOLOGY

Current simulation methodologies do not ensure that all threads in a workload are faithfully represented in the simulation results. To alleviate this problem, we propose a new methodology called FAME. The main objective of our methodology is to obtain representative measurements of the actual processor behavior. In order to do it, it is necessary to ensure that all threads in a given workload are faithfully represented whenever the workload finalizes execution. Intuitively, all threads in a given workload are accurately represented if all the instructions from every thread have been completely executed the same number of times when the workload execution ends. This is the case in single-threaded processors, in which every workload has just one thread that is executed once. However, in a multithreaded processor, every thread runs at a different speed depending on the dynamic program phases and the availability of shared resources and thus it is difficult that all threads finish at the same time. Furthermore, it is not likely that all instructions execute the same number of times.

4.1 Trace Reexecution

The objective of FAME is to determine how many times a program in a workload should be reexecuted for being faithfully represented. In order to determine it, FAME analyzes the behavior of every program in isolation. In this paper we assume that the behavior of each thread in a workload executed in multi-thread mode remains similar to the behavior in single-thread mode as the code signatures do not change.

Depending on the particular methodology features, the execution of each thread in a workload may be stopped at any point, and the IPC value provided by the methodology will be the average IPC value until that point. This average

IPC would be fully representative of the thread execution if it is similar to the final IPC value, that is, the average IPC value at the end of the whole thread execution. Therefore, the FAME methodology forces each thread to be executed enough times so that the difference between the obtained average IPC and the final IPC is below a particular threshold.

The basis of FAME can be better explained using a synthetic example. Light-grey bars in Figure 3(a) show the instant IPC of our synthetic application, that is, the IPC on each particular cycle of its entire execution when run in isolation. The black line shows the evolution of the average IPC of the application along its execution. The average IPC value for a given execution cycle is calculated as the average value of the instant IPC from the beginning of the program execution until that particular cycle. Thus, the final IPC would be equal to the average IPC value at the end of program execution.

It becomes clear that the average IPC converges towards the final IPC value. Figure 3(b) shows the difference between the average IPC and the final IPC. As expected, this difference is a decreasing function. The more instructions executed by a thread, the more representative its average IPC is.

Figure 3(c) shows the instant IPC and the average IPC during three reexecutions of the application. In addition, Figure 3(d) shows the difference between the average IPC and the final IPC during the three reexecutions. The difference is still a decreasing function, but it is important to note that it is not monotone. This means that the difference would be very small in a given cycle, but it may increase again in the subsequent cycles. Therefore, if the goal is to obtain representative measurements, thread execution cannot be stopped at any point.

One could think that the solution is to finalize program execution when a full application repetition has been executed, since the average IPC is always equal to the final IPC at the end of any repetition. However, a multithreaded processor is able to execute more than one application at once. Although simulation can be stopped at the end of a repetition for one of the threads, it is likely that this point is not the end of a repetition for the other threads, and thus the other threads will not be accurately represented. The actual solution comes from the observation that, although the difference between the average and the final IPC does not decreases monotonically, the maximum difference in a reexecution is lower for every new executed repetition. That is, it is a decreasing monotone function. Thus, if we execute enough repetitions of a thread, the maximum difference will reach a value small enough to consider that the average IPC is representative of the full benchmark behavior. For this reason, our methodology reexecutes all threads several times, until the difference is upper-bounded by a given threshold.

Figure 3(d) shows the difference between the average and the final IPC as our synthetic program is reexecuted. The highest difference values are obtained in the first repetition due to the warm-up period of the thread. The difference decreases along with the thread execution, reaching zero when the first repetition finishes. Indeed, the difference is always zero at the end of every thread repetition, since the average IPC is always equal to the final IPC at those points.

It can be observed in Figure 3(d) that the average IPC of the first repetition is not representative of the average IPC behavior in following repetitions due to the warm-up period. For this reason, we discard the first repetition. It can also be observed that the difference between the average and the final IPC presents similar behavior for all repetitions excluding the first one. Furthermore, the instruction and the cycle in which the difference achieves its higher value is always the same for all repetitions.

Let $InstMax_2$ be the instruction in the second repetition that reaches the maximum difference between the average IPC and the final IPC value within that repetition. Let also $CycleMax_2$ be the cycle in which that instruction is executed. Since the instruction and cycle in which the application reaches the maximum difference is always the same for all repetitions from the second one onwards, we can compute the number of instructions and cycles that should be executed to reach $InstMax_i$ and $CycleMax_i$ for every repetition i. This calculation is performed with formulas 1 and 2, in which $TotalInst$ and $TotalCycle$ are the total number of instructions and cycles of the program on each repetition.

$$InstMax_i = (TotalInst * (i-2)) + InstMax_2 \quad (1)$$

$$CycleMax_i = (TotalCycle * (i-2)) + CycleMax_2 \quad (2)$$

These equations make possible to compute the maximum difference value for any thread repetition beyond the second one without needing to actually execute it. In other words, executing two repetitions is enough to calculate the maximum difference value for any number of additional repetitions, greatly reducing the simulation time required to obtain these values. Thus, the maximum difference value from the beginning of the first repetition, can be calculated using equation 3.

$$DiffMax_i = \left| \frac{InstMax_i}{CycleMax_i} - FinalIPC \right| \quad (3)$$

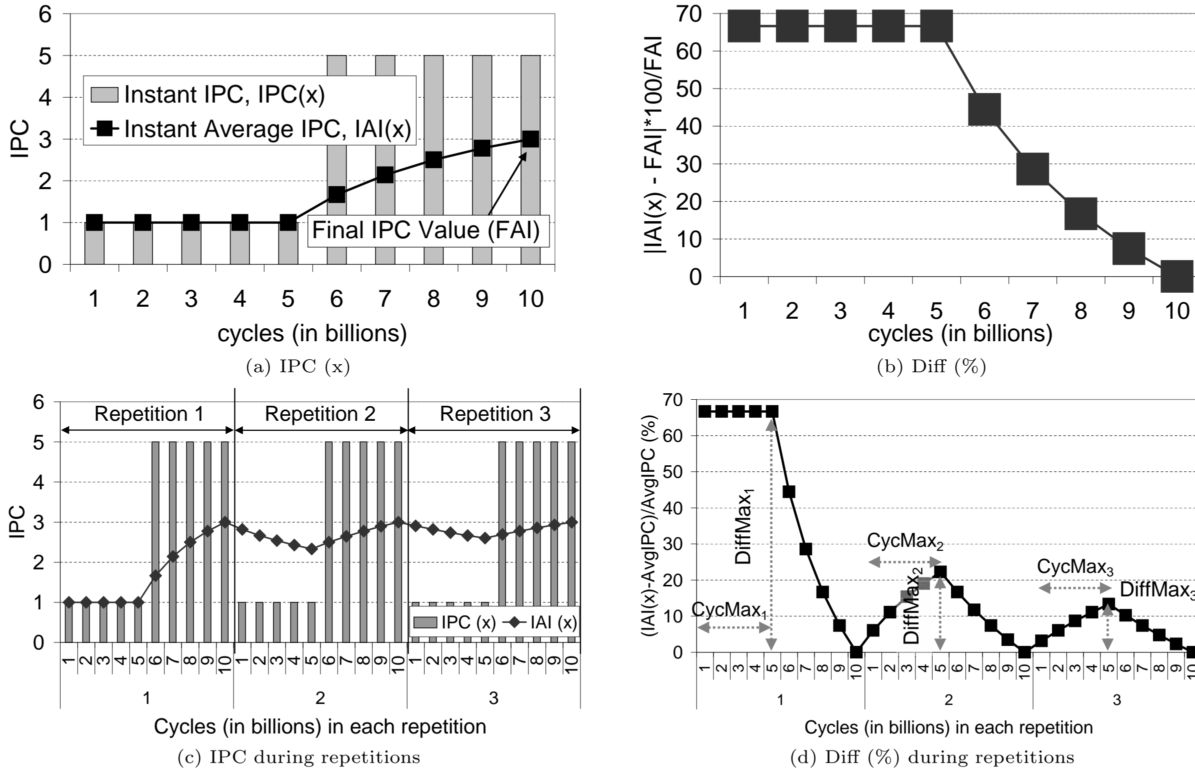
From equation 3 we can deduce a formula to calculate the minimal number of repetitions required to ensure representativity of a thread. Since it is not possible to achieve perfect representativity, we define a threshold value that indicates the maximum difference between the average IPC and the final IPC that is acceptable to consider that the average IPC value obtained is representative of the full thread execution. We call this threshold the *Maximum Allowable IPC Variance* (MAIV).

In order to obtain representative results, simulation will not finalize until all threads have reached the point where the maximum difference between the average IPC and the final IPC is smaller than a chosen MAIV. From this point onwards, simulation can be stopped at any time. The minimal number of repetitions required to fulfill a given MAIV requirement can be obtained working out the value of i from equations 1, 2 and 3. The result is shown in Equation 4, which states how to calculate the minimal number of repetitions required from the data obtained during the second repetition.

4.2 Applying FAME

As with previous simulation methodologies, the first step to apply the FAME methodology is obtaining a representative trace of every benchmark. We have selected the SimPoint tool [15] to generate them. Once traces are obtained, we simulate two repetitions of every trace in isolation. Periodically, we sample the IPC of the application obtaining the IPC during execution.

Figure 4 shows the instant IPC of *perlbmk* (a) and *wupwise* (b). For this experiment we sample the IPC of each benchmark every 15,000 cycles. As before, the light-grey bars and the black line represents the instant IPC and the



(a) IPC (x)

(b) Diff (%)

(c) IPC during repetitions

(d) Diff (%) during repetitions

Figure 3: Instant IPC and average IPC of T0.

$$i \geq \left\lceil \frac{(CycleMax_2 - 2 * TotalCycles) * (FinalIPC * (1 + MAIV)) - InstrMax_2 + 2 * TotalInst}{TotalCycles * (FinalIPC * (1 + MAIV)) - TotalInst} \right\rceil \quad (4)$$

average IPC of the given benchmark respectively. The final IPC is the average IPC at the end of the simulation. Figure 4(a) shows an scenario in which the instant IPC of the application (*perlbmk*) varies noticeably. On the other hand, Figure 4(b) shows a scenario in which the instant IPC of the application does not vary significantly (*wupwise*). Intuitively, in order to fulfill a given MAIV, it would be necessary to reexecute more times *perlbmk* than *wupwise*, since its average IPC presents more variability. From this information we obtain $CycleMax_2$ and $InstMax_2$, and compute the number of re-executions, i, required to satisfy a given MAIV.

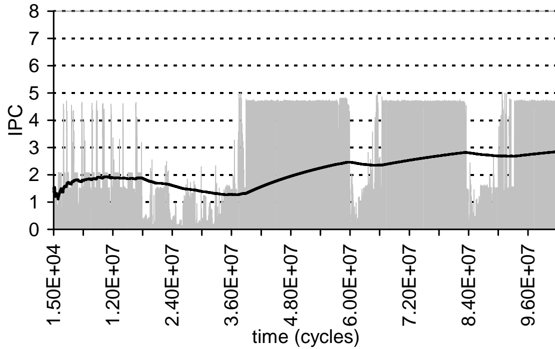
Table 5 shows the minimal number of reexecutions required for both SpecInt and SpecFP with MAIV values ranging from 20% to 1%. The lower the MAIV value is, the higher accuracy required, and thus, in most cases, the more repetitions are needed. For example, if a MAIV value of less than 1% is required, some benchmarks (*gap*, *gcc*, *apsi* and *galgel*) have to be reexecuted more than 30 times to be accurately represented in the workload. It is also noticeable that when the MAIV requirements are relaxed (20%) only 2 repetitions are needed in most of the SPECs. This is the minimal number of reexecutions done by FAME, since the first repetition is discarded.

Once the traces and the minimal number of repetitions are obtained, workload simulations can begin. Workload simulation will not finalize until every thread in the workload has been executed, at least, as many times as the minimal number of repetitions required for accurate representativity. If any thread reaches this minimal number of repetitions before the rest of the threads, it will reexecute once and again until all threads fulfill their requirements. This is not a problem for representativity, since the maximum difference between the average and the final IPC can only decrease. When all threads have been reexecuted at least the corresponding minimal number of times, workload execution can be stopped at any point, since we can ensure that the results are representative. For example, if the workload composed by *gcc* and *perlbmk* and a MAIV of 1% is required, *gcc* and *perlbmk* must be reexecuted at least 33 and 22 times respectively. If *perlbmk* finishes first, the simulator must reexecute it once and again to keep the complete workload executing, that is, to maintain a fair scenario for the execution of the other thread. Once both benchmarks reach the minimal number of repetitions, simulation finalizes.

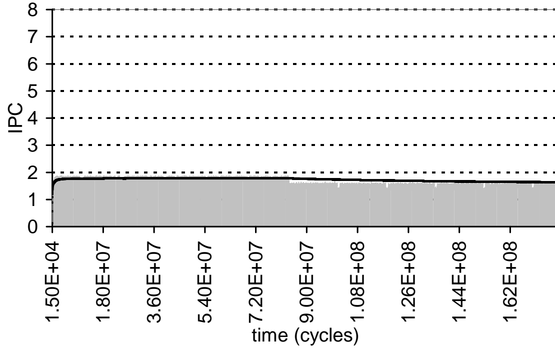
It is interesting to note that, when a thread is reexecuted, we flush the data of this thread from the memory hierarchy. This flush procedure is done to prevent the processor from unfairly taking advantage of the warming-up of structures. Indeed, real operating systems do so. In every context switch the TLB is invalidated and thus, the memory hierarchy is flushed. Nevertheless, we have found that, for our experimental setup, the initialization part is a negligible percentage of the total execution time and it does not vary the results. The difference between flushing and not flushing is less than 0.01% for all cases.

5. ANALYSIS OF EVALUATION METHODOLOGIES

In order to show that FAME is more accurate than currently used methodologies, we calculate the error of every methodology respect to a given baseline. We have checked that 50 repetitions are enough for all programs to reach a



(a) perlbmk



(b) wupwise

Figure 4: Instant and average IPC of two simulated benchmarks with different behavior.

Table 5: Number of repetitions required in the baseline architecture.

Bench.	MAIV(%)				
Name	20	10	5	2	1
bzip2	2	2	2	4	7
crafty	2	2	2	2	2
eon	2	2	2	2	2
gap	3	4	8	18	35
gcc	3	4	7	17	33
gzip	2	2	3	5	9
mcf	2	2	2	2	2
parser	2	3	4	9	16
perl	2	3	5	11	22
twolf	2	2	2	2	2
vortex	2	2	2	4	7
vpr	2	2	2	2	2

(a) Spec CPU INT

Bench.	MAIV(%)				
Name	20	10	5	2	1
ammp	2	2	2	2	2
applu	2	2	2	5	8
apsi	3	4	8	18	36
art	2	2	2	2	2
equake	2	2	2	2	3
galgel	3	4	7	16	31
lucas	2	2	2	2	4
mesa	2	2	2	4	7
mgrid	2	2	2	3	4
swim	2	2	3	6	11
wupw.	2	2	2	3	5

(b) Spec CPU FP

steady state in which the IPC of the following executions is nearly constant. Hence, our baseline will be the IPC of every workload after executing at least 50 times every thread.

5.1 Error of the Methodologies

In a first experiment, we measure per-thread IPC. If per-thread IPC is accurate, our FAME methodology can be used to study any metric, like throughput, since per-thread IPC is the only variable parameter used to compute these metrics. We calculate the error of every thread in a workload for every methodology using formula 5, in which $T_iIPC_{baseline}$ is the IPC of thread i for the baseline, and $T_iIPC_{workload}$ is the IPC of thread i reported by the methodology under study.

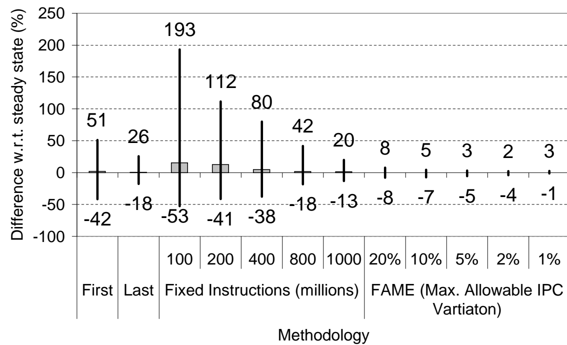
$$ErrorT_i = \frac{T_iIPC_{baseline} - T_iIPC_{workload}}{T_iIPC_{baseline}}(\%) \qquad (5)$$

Figure 5 shows the average error of every methodology respect to the baseline. Data is presented for thread 0, Figure 5(a), and thread 1, Figure 5(b), of every workload. For example, thread 0 in the workload composed by *gap* and *perlbmk* is *gap*, and thread 1 is *perlbmk*. For every methodology, we show the average error (grey bars) and the maximum positive and negative errors. Both figures present different results because we do not simulate any particular workload combination more than once (e.g. if we simulate *gap*+*gcc*, then we do not simulate *gcc*+*gap*).

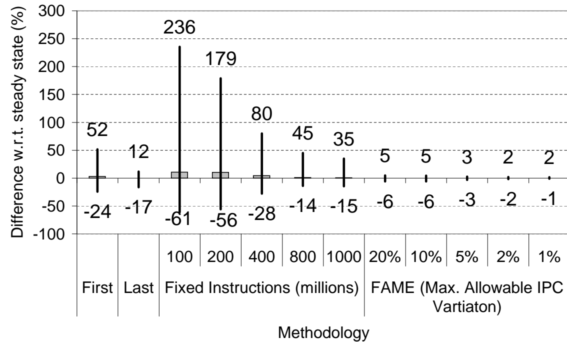
The main observation from Figure 5 is that all current methodologies present a noticeable average error. The 100 and 200-million Fixed Instruction methodologies achieve the worst results, presenting an average error over 15%. These noticeable error values are due to the fact that one or both threads are not accurately represented in the final result of the workload, as described in previous sections. Even the 1-billion Fixed Instructions methodology, which provides the lowest errors of all current methodologies, presents an average error of 5%, having a maximum positive error of 35% and a maximum negative error of -15% for thread 1. Such a large range of variation in the error value should not be tolerated, since it would lead to inaccurate and misleading conclusions when evaluating multithread processors.

Figure 5 also shows the average, maximum positive and maximum negative errors of FAME with MAIV values ranging from 20% to 1%. When MAIV is 20%, the average error is below 0.5%, having less than +/-8% variation, what makes FAME the methodology that presents the lowest error. Therefore, the measurements obtained with FAME are more representative of the final results than the ones obtained with any other methodology. As it can be expected, the lower the MAIV value is, the lower the error obtained by FAME, since the higher precision is required. On the other hand, a lower MAIV value also involves that a higher number of iterations are needed, increasing execution time. It depends on the researcher to decide about this trade-off, but it should be noted that even using a 5% MAIV value will require much lower execution time than the baseline execution, just having a negligible +/- 5% of maximal error. We do really believe that a MAIV of 5% is a good trade-off between representativity and simulation overhead.

These results show that FAME provides accurate results per each separate thread. Moreover, since per-thread performance is the only variable parameter used by most multi-threaded performance metrics, FAME is also able to provide

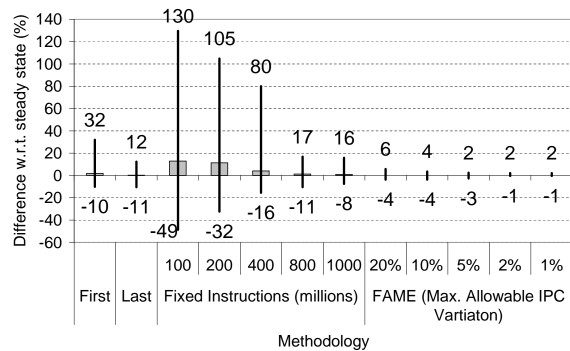


(a) Thread 0

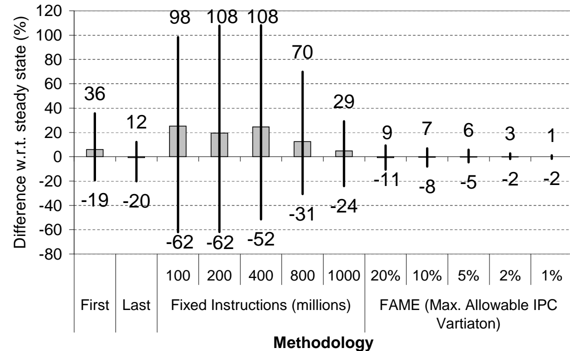


(b) Thread 1

Figure 5: Error of the different methodologies for the 2-thread workloads



(a) 2-context SMT



(b) 4-context SMT

Figure 6: Error of the different methodologies using the IPC throughput as metric

accurate results per any of them. For instance, Figure 6(a) shows the global errors of the methodologies taking into account the throughput metric. Again, FAME is the methodology that obtains the lowest errors, ranging from -3% to 3% and from -4% to 6% when the MAIV constrain is relaxed (20%).

To show that FAME also alleviates the representativity problems in other scenarios, we test all the methodologies using 4-thread workloads (Figure 6(b)). In this case, only the 6 benchmarks with the highest IPC variability (*gcc*, *parser*, *perlbmk*, *gap*, *galgel* and *apsi*) are used to compose workloads, leading to a total of 126 4-thread workloads[3]. Again, FAME is the methodology that presents the lowest errors, ranging from +9% to -11% when the MAIV constrain is relaxed (20%) and from 1% to -2% when the more accurate 1% MAIV is required. The best results from current methodologies are obtained by Last, which has maximum errors ranging from +29% to -24%.

Figures 6(a) and 6(b) also show that, as the number of simulated instructions is increased, the accuracy error decreases. For example, in the 2-thread configuration (Figure 6(a)) the 200-million Fixed Instructions methodology presents an error interval from 105% to -32%, whereas the 1-billion Fixed Instructions methodology leads to an error interval ranging from 16% to -8%. It can be observed that, in the 4-thread configuration (Figure 6(b)), the interval of error has increased to 108%, -62% for the 200-million methodology and to 29%, -34% for the 1-billion methodology. The problem with these methodologies is that we cannot fix *a priori* the number of instructions to simulate in order to obtain a low error, since this number depends on both the simulator setup and the number and mix of threads in every workload. In contrast, our FAME methodology presents a much more stable behavior regardless of the configuration, which is a desirable characteristic for any methodology.

[3]Using SPEC2000 benchmarks, there are 14950 possible 4-thread workload combinations, which would make simulation time unaffordable.

5.2 Execution time of FAME

As we showed, FAME is the methodology that presents the lowest error. But there is another important factor that cannot be disregarded. This is simulation time. Table 6 shows the simulation time (in minutes) needed by each methodology in our environment for two-thread and the four-thread workloads. It is also included the baseline (steady state) where every thread reexecutes at least 50 times. Notice that these numbers highly depend on the simulator (smtsim [20]) and the execution environment.

Table 6: Mean execution time in minutes

Speedup	Workload	
Methodology	2-thread	4-thread
Steady_State	5247.50	9184.00
First	105.00	182.69
Last	139.22	243.56
I100	13.98	12.30
I200	27.89	24.46
I400	55.83	49.00
I800	134.58	98.49
I1000	166.44	140.98
MAIV_20	252.95	450.11
MAIV_10	295.97	532.46
MAIV_5	425.03	779.16
MAIV_2	979.06	1804.54
MAIV_1	1853.20	3424.94

As expected the Fixed Instructions methodologies spend

less time to complete than the other methodologies since they execute the lowest number of instructions. Note that for 4-thread workloads execution time is faster than for 2-thread as the IPC for 4-thread workload is higher than for 2-thread workloads. On the other hand, FAME presents affordable times to obtain the lowest errors. For the 4-thread workloads, the execution time of FAME is clearly improved respect to the baseline, only needing less than 5% of the baseline execution time for the MAIV_20. From Table 6, it is easy to see than MAIV_10 and MAIV_5 are good tradeoffs between simulation error and time.

As ongoing work, we are improving FAME to substantially reduce the execution time detecting the number of reexecutions dynamically.

6. RELATED WORK

Since executing complete programs is not possible in current simulation frameworks, computer architecture researchers have been forced to select representative parts of the programs in order to get simulation results in a reasonable time. The SimPoint tool [15] [16] has become a popular mechanism to obtain those representative traces. Program execution is broken into consecutive intervals whose code signatures are represented using basic block vectors. Intervals with similar code signatures are grouped into a same program phase. From each phase, a representative interval is selected for simulation, since all the others are expected to have very similar behavior. Therefore, it is possible to extrapolate the behavior of the full program without needing to completely simulate it.

6.1 Previous Methodologies and Metrics

Accurate simulation can be performed for mono-thread architectures by using representative traces selected using SimPoint or any other technique with the same proposal, such as the state-reduction technique [8], the statistical slice classification technique [10], and SMARTS [23]. However, these techniques are not enough to get accurate evaluation results for multithreaded architectures due to performance variability phenomena in multithreaded workloads. This problem was identified for truly parallel cooperative workloads in [3], which also presents a statistical method to overcome this problem.

Opposite to [3], our work is focused on non-cooperative workloads composed of independent threads. Several methodologies and metrics have been proposed in the literature for measuring the performance of multithreaded processors executing non-cooperative workloads. On the one hand, evaluation methodologies determine how to take measurements from a workload. Current methodologies can be grouped into three sets according to the point in which simulation ends: when the first thread finishes [7], when the last thread finishes [24], and when a fixed amount of instructions have been simulated [12] [9]. On the other hand, metrics determine how to compute a representative value from the measurements obtained using an evaluation methodology. The most commonly used metrics are throughput [21], harmonic mean [12], and weighted speedup [18].

FAME is an evaluation methodology that provides more accurate measurements than any of the aforementioned methodologies. These accurate measurements can be later used to compute throughput, harmonic mean, weighted speedup, or any other possible metric. Like previous evaluation methodologies, FAME is absolutely independent on the technique used to select representative parts of program execution. In particular, the results presented in this paper have been obtained using SimPoint to select a single representative interval per program. Although it is stated in [5] that using single-interval SimPoint does not provide accurate enough intervals for multithreaded simulation, we consider it would be not necessarily true. The poor accuracy obtained for single-interval SimPoint in [5] would be due to lack of representativity in the selected interval, but it would also be due to the fact that this interval is not reexecuted enough times. FAME determines how many times an interval should be reexecuted to provide accurate results and thus it would solve the latter problem. We are currently working on applying FAME in conjunction with multiple-interval SimPoint, since this is an interesting topic for future research.

Furthermore, it is interesting to note that FAME is applicable to all types of multithreaded processors, including chip multiprocessors. A clear example is the IBM Power5 processor, which is a chip multiprocessor containing two 2-thread simultaneous multithreaded cores. Since Power5 architects were aware of the performance variability problem in multithreaded workloads, they evaluated their design using 4-thread workloads containing the same application replicated four times [17]. We have found that, when we execute a workload containing a single program replicated several times, the performance variability is almost negligible regardless the evaluation methodology used. However, using just this type of workload limits the variety of analysis and evaluation that can be done. FAME would have allowed evaluating the Power5 processor using any arbitrary workload, since it is a more general methodology.

6.2 The Co-phase Matrix

The co-phase matrix [5][4] is an evaluation methodology having an objective close to FAME. This methodology uses multiple-interval SimPoint to identify program phases and selects a representative interval per phase. In that paper authors use 5-million intervals and on average each program has 27 phases. Once phases are identified for all programs in a multithreaded workload, a matrix is populated with information for all possible combinations of phases, one per program in the workload, which can be run together during multithreaded execution. Data for each phase combination is gathered a detailed simulation that finishes when a thread executes 3.5-million instructions.

Multithreaded simulation is done analytically after the co-phase matrix is built. Given the starting point of the threads in the workload, it is possible to determine the starting combination of phases. The co-phase matrix contains enough information to estimate the performance of all threads during the execution of this combination of phases. Moreover, it allows determining the point at which one of the threads will change from one execution phase to another, which means that the combination of phases also changes. Looking again at the co-phase matrix, it is possible to repeat the process for the new combination of phases, and so on until the performance of the whole workload execution is estimated.

In [4] the authors use the static methodology proposed in [5]. The static method is repeated for different starting points for each thread in the workload, until the results sta-

[4]Before describing the co-phase matrix approach [5][4], we would like to emphasize that at the time we sent this paper to evaluation, we did not know the existence of [4]. During the evaluation process we received the recommendation to extend the related work including a description of [4]. Here, we make a qualitative comparison of FAME and [4]. Nevertheless, as future work we will make a quantitative comparison of both methods.

tistically converge for a given level of confidence. In other words, in [4] hundreds to thousands I1000 samples from different starting thread locations are averaged using the static co-phase approach to statistically estimate performance over the entire program's execution. Our FAME approach, as the approach in [4], is focused looking for statistical convergence and providing an accurate simulation, but assumes a single starting point position in each trace being simulated.

7. CONCLUSIONS

To guarantee the resemblance between the real world and the research environment in multithreaded architectures is mandatory the use of an appropriate measuring methodology. The evaluation of the capabilities of a multithreaded processor using a given workload requires taking measurements when all the threads in that workload are running. However, the execution speed of every thread in a workload varies according to the particular thread features and the availability of shared resources, which makes some threads finalize execution before others. This fact forces researchers to define, firstly, when the workload execution finalizes and, secondly, when measurements are taken. However, the methodologies currently used to define these features cannot ensure that these results are representative. Even worse, since thread speed also depends on the processor features, any change in the processor setup would vary the mix of executed instructions from every thread, and thus two results obtained using two different processor setups are not comparable.

To deal with these problems we propose FAME, a novel evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME is mainly based on representative trace reexecution since, when a benchmark is reexecuted enough times, its average IPC value converges to a representative result. Therefore, once all benchmarks in a workload are executed a required number of times, it is possible to stop workload simulation at any arbitrary point, since representativity is ensured.

As a case study, we apply FAME to a well-known SMT simulation tool. We have shown that FAME achieves better accuracy than previously proposed evaluation methodologies. In addition, any metric can use the measurements obtained with FAME, since a methodology just dictates how to take measurements and not how to use them. Even more, since the main difference among multithreaded designs is the amount of shared resources, all of them present the same evaluation problems, making FAME directly applicable to SMT processors, CMP processors, and even CMP/SMT processors in both research and real scenarios. In particular, as a future work, we will apply FAME to a real scenario in which an actual processor, such as the Pentium 4, is the case study.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2004-07739-C02-01, the HiPEAC European Network of Excellence, an Intel fellowship and an IBM fellowship. The authors would like to thank Jaume Abella, and Beatriz Otero for their comments. We also would like to thank our shepherd Brad Calder for his help during the camera ready of this paper.

8. REFERENCES

[1] http://www.nas.nasa.gov/software/npb/.
[2] http://www.specbench.org/.
[3] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. *Proceedings of the 9th HPCA*, 2003.
[4] M. V. Biesbrouck, L. Eeckhout B. Calder Considering All Starting Points for Simultaneous Multithreading Simulation *Proceedings of ISPASS*, 2006.
[5] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. *Proceedings of the ISPASS*, 2004.
[6] D. Burger and T. Austin. The simplescalar tool set, version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.
[7] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in SMT processors. *Proceedings of the 37th MICRO*, 2004.
[8] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. *Proceedings of the ICCD*, 1996.
[9] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. *Proceedings of the 36th MICRO*, 2003.
[10] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *In IEEE Workshop on Workload Characterization*, 2000.
[11] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. *Proceedings of the IPDPS*, 2001.
[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *Proceedings of the ISPASS*, 2001.
[13] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.
[14] M. J. Serrano, R. Wood, and M. Nemirovsky A Study of Multistreamed Superscalar Processors. Technical Report #93-05, University of California, Santa Barbara, 1993.
[15] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *10th PACT*, 2001.
[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatic characterizing large scale program behavior. *10th ASPLOS*, 2002.
[17] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.
[18] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *SIGMETRICS*, 2002.
[19] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *34th MICRO*, 2001.
[20] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism *Proceedings of the 22nd ISCA*, 1995.
[21] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd ISCA*, 1996.
[22] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. *37th MICRO*, 2004.
[23] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *30th ISCA*, 2003.
[24] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. *Proceedings of CASES*, 2005.