On the Problem of Evaluating the Performance of Multiprogrammed Workloads

Francisco J. Cazorla, *Member*, *IEEE*, Alex Pajuelo, Oliverio J. Santana, Enrique Fernández, and Mateo Valero, *Fellow*, *IEEE*

Abstract—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 for 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 new evaluation methodology aimed to fairly measure the performance of multithreaded processors executing multiprogrammed workloads. FAME reexecutes all programs in the workload until all of them are fairly represented in the final measurements taken. We compare FAME with previously used methodologies showing that it provides more accurate measurements, becoming an ideal evaluation methodology to analyze proposals for multithreaded architectures.

Index Terms—Multithreaded processors, multiprogrammed workloads, evaluation methodologies.

1 Introduction

MULTITHREADED architectures are able to execute several applications simultaneously, breaking the limitations of instruction-level parallelism and boosting processor performance. The continuous evolution of multithreaded architectures has lead to designs including simultaneous multithreaded processors (SMT) [1], [2], [3], chip multiprocessors (CMP), and even hybrid CMP/SMT processors composed of separate SMT execution cores like the IBM POWER5 and POWER6 and the Intel i7.

In spite of the increasing trend to use truly parallel applications, multiprogrammed workloads are still commonly used to evaluate CMP/SMT processors. Computer architecture researchers frequently evaluate their proposals for multithreaded architectures using workloads composed of independent single-thread applications like SPEC CPU [4]. In this paper, we focus on evaluation methodologies for multithreaded architectures executing multiprogrammed workloads composed of noncooperative single-thread applications that perform nonrelated work and do not communicate each other.

For fully evaluating a wide range of scenarios, workloads comprised of benchmarks with different behaviors should be used. However, it is important to notice that working with different programs running simultaneously involves an important decision: to determine when the execution of the multiprogrammed workload will finish. In a single-threaded processor, the full program is run until completion, but it is not so trivial in a multithreaded processor running several programs at the same time. Applications in a workload execute at different speeds due to the particular features of each program and the competition for the shared resources. Consequently, it is unlikely that they complete execution at the same time.

This fact has a negative impact on the accuracy of the evaluation of multithreaded processors. For example, assume a two-thread workload running in a multithreaded processor with two execution contexts. Both threads start executing at the same time and share the processor resources during a period of time. Since these threads run at different speeds, one of them will end its execution before the other. Therefore, after the two-thread period, there is a one-thread period in which a single thread is executed alone. This is an undesirable side effect, since the full potential of this multithreaded processor is only exploited during the two-thread period and, thus, only results from the two-thread period should be considered.

Several evaluation methodologies have been proposed to face this problem. These methodologies determine how to perform executions to assure that all the programs in the workload are still running when the measures are collected. For instance, in the previous example, the second thread may be finalized when the first thread ends or the first thread may be reexecuted until the second thread ends. However, with all these methodologies, it cannot be assured that the measurements obtained are actually representative of the whole program behavior.

Our approach to overcome this problem is FAME [5], a novel methodology for the evaluation of multithreaded processors executing multiprogrammed workloads. FAME is a flexible methodology that can be applied to any multithreaded architecture, including SMT, CMP, and CMP/SMT. The benefits of our proposal are shown through results from real SMT and CMP processors (Intel Pentium 4 and Dual Core). FAME is also applicable to research simulators [5]. Overall, our results show that FAME provides more accurate measurements than previously used methodologies.

2 Evaluation Methodologies

In order to characterize multithreaded processors, we must distinguish between three orthogonal concepts: *measurements*, *methodologies*, and *metrics*. Measurements are objective data collected from the execution of a workload in a given processor. Common examples are IPC and power consumption. A methodology defines how the workload execution is performed and when the measurements are taken. These measurements are later combined into metrics such as IPC throughput, weighted speedup [6], and harmonic mean [7].

In this paper, we focus on evaluation methodologies. Evaluation methodologies are critical because they determine the representativeness of the taken measurements. The key idea is that if the measurements are inaccurate due to the selection of a wrong evaluation methodology, the precision of the metrics, and hence the system characterization, will degrade even if the metrics are supposed to provide fairness.

In order to fairly evaluate the performance of SMT or CMP processors, 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

- *F.J. Cazorla is with the IIIA-CSIC (Spanish National Research Council), Spain, and the Barcelona Supercomputing Center, Edificio Nexus II, Jordi Girona 29, 08034 Barcelona, Spain. E-mail: francisco.cazorla@bsc.es.*
- *A. Pajuelo is with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, C6-205, Campus Nord UPC, C/Jordi Girona, 1-3, 08034 Barcelona, Spain. E-mail: mpajuelo@ac.upc.edu.*
- *O.J. Santana and E. Fernández are with the Departamento de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus Universitario de Tafira, 35017 Las Palmas de Gran Canaria, Spain. E-mail: {ojsantana, efernandez}@dis.ulpgc.es.*
- *M. Valero is with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, C6-205, Campus Nord UPC, C/Jordi Girona, 1-3, 08034 Barcelona, Spain, and the Barcelona Supercomputing Center, Edificio Nexus II, Jordi Girona 29, 08034 Barcelona, Spain. E-mail: mateo@ac.upc.edu.*

Manuscript received 2 Sept. 2009; revised 13 Jan. 2010; accepted 15 Jan. 2010; published online 26 Feb. 2010.
Recommended for acceptance by M. Yousif.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-09-0451.
Digital Object Identifier no. 10.1109/TC.2010.62.

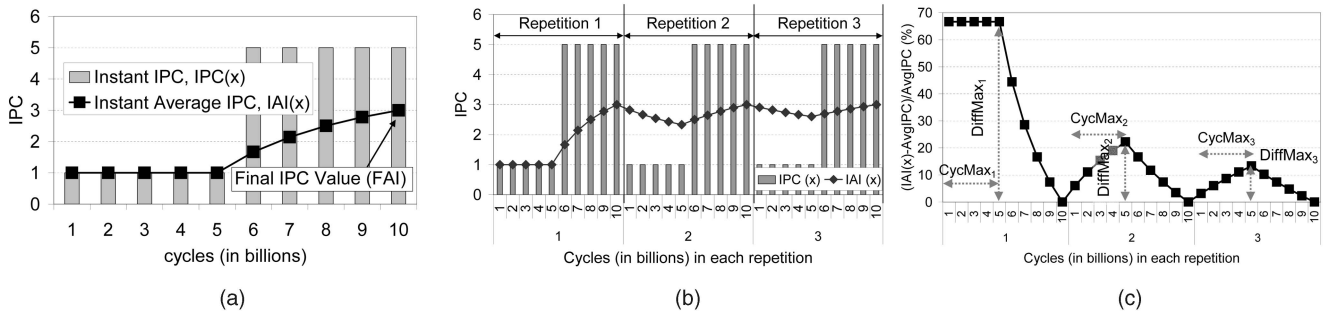

Fig. 1. Instant IPC, average IPC, and difference between both in a synthetic program during three repetitions. (a) IPC (x). (b) IPC during repetitions. (c) Difference (percent) during repetition.

should determine what to do whenever any thread finalizes its execution. All current evaluation methodologies can be classified based on its *finalization method*. This classification includes the First, the Last, and the Fixed Instructions methodologies.

The *First* methodology finalizes the execution of a workload when any program of the workload ends its execution [8]. The main drawback of this methodology is that only one program in the workload is executed until completion, and thus it cannot be ensured that the remaining programs execute completely, losing representativeness in the final result.

The *Last* methodology finalizes a workload execution when all the programs have run until completion. 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 programs depends on the processor parameters, any variation can cause all programs to be executed at different speeds. As a consequence, it cannot be ensured that the amount of executed instructions is the same for different executions 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 evaluation. The execution finalizes whenever the total number of executed instructions reaches a fixed threshold. This threshold is usually determined per thread, that is, the execution 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 [9] or 200-million instructions [10] to 1-billion instructions [11].

However, the Fixed Instructions methodology is also unable to ensure that a representative part of every benchmark is being executed, since workload execution 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 two-context SMT processor. IF1 always prioritizes instructions belonging to the first context and IF2 always prioritizes instructions belonging to the second one. The execution finishes when N instructions from both threads are executed. When both executions end, they have ran 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 The FAME Methodology

Current evaluation methodologies do not ensure that all programs in a workload are faithfully represented in the final 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. FAME determines how many times a program in a workload should be reexecuted for being faithfully represented, that is, to ensure that the difference between measurements collected at that point and real values is below a particular threshold. Executions are then run until all programs have been executed the minimum number of times required to be representative. Once this number is reached for all programs, execution can end at any point, since representativeness is guaranteed for all programs.

The basis of FAME can be better explained using a synthetic example. Light-gray bars in Fig. 1a show the instant IPC of a 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 is clear that the average IPC converges toward the final IPC value.

Fig. 1b shows the instant IPC and the average IPC during three reexecutions of the application. In addition, Fig. 1c shows the difference between the average IPC and the final IPC during the three reexecutions. It is clear that the average IPC converges toward the final IPC value. Even if that difference is a decreasing function, it is important to note that it is not monotonic. 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, program 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 execution can be stopped at the end of a repetition for one of the programs, it is likely that this point is not the end of a repetition for the other programs, and thus the other programs could be not accurately represented. The actual solution comes from the observation that although the difference between the average and the final IPC does not decrease monotonically, the maximum difference in a reexecution is lower for each new executed repetition, that is, it is a decreasing monotonic function. Thus, if we execute enough repetitions of a program, 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 programs several times, until the difference is upper bounded by a given threshold.

Fig. 1c 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 cold-start IPC calculation of the program. The difference decreases along with the

program execution, reaching zero when the first repetition finishes. The difference is always zero at the end of every program repetition, since the average IPC is always equal to the final IPC at those points. It can be observed in Fig. 1c that the IPC behavior of the first repetition is not representative of the IPC behavior in following repetitions due to the cold-start effect. 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. Indeed, the instruction and the cycle in which the difference achieves its highest value are 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 (1) and (2), in which $TotalInst$ and $TotalCycle$ are the total number of instructions and cycles of the program on each repetition.

$$InstMax_i = (TotalInst * (i-2)) + InstMax_2, \tag{1}$$

$$CycleMax_i = (TotalCycle * (i-2)) + CycleMax_2. \tag{2}$$

These equations make possible the computation of the maximum difference value for any program 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 execution time required to obtain these values. Thus, the maximum difference value from the beginning of the first repetition can be calculated using (3).

$$DiffMax_i = \left| \frac{InstMax_i}{CycleMax_i} - FinalIPC \right|. \tag{3}$$

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

In order to obtain representative results, executions 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, execution can be stopped at any time. Equation (4) states how to calculate the minimal number of repetitions required to fulfill a given MAIV requirement. This equation is obtained working out the value of i from (1)-(3)

$$\begin{aligned} i \geq \lceil [(CycleMax_2 - 2 * TotalCycles) * (FinalIPC * (1 + MAIV)) \\ - InstrMax_2 + 2 * TotalInst] / \\ [TotalCycles * (FinalIPC * (1 + MAIV)) - TotalInst] \rceil. \end{aligned} \tag{4}$$

4 Experimental Framework

FAME can be applied to both simulation environments and real processors. In this paper, we focus on the latter. An evaluation of FAME for simulation environments can be found in [5].

To prove the suitability of FAME in real scenarios, we apply the methodology to both SMT and CMP processors representative of current trends of multithreaded designs. As SMT processor we use a 4.3 GHz Intel Pentium processor with Hyperthreading Technology [3] and 512 MBytes of DDRAM at 400 MHz. As CMP processor, we use a 2 GHz Intel Xeon 5130 Dual Core [3] processor with 8 GBytes of DDRAM (1,333 MHz FSB). In both processors, we boot a Fedora Core 3 with GNU linux kernel 2.6.11 patched with perfectr-2.6.18 to allow the access to the performance monitoring counters from any privilege level of execution. The operating system is booted at runlevel one to reduce as much as possible the interferences generated by multiuser/multitasking processing. Video, audio, and communication hardware capabilities are disabled. The whole SPECcpu2000 benchmark suite was compiled with all optimizations enabled using gcc 3.4.2 and Intel Fortran 9.0. Benchmarks were executed until completion with the reference input set. In the SMT processor, benchmarks belonging to a workload are executed in different hardware contexts (hardware threads). In the dual core processor, the benchmarks are executed in different cores.

Workloads were generated with all the possible combinations of the SPECcpu2000 benchmarks. In this sense, it is interesting to note that our workloads are composed of noncooperative single-threaded applications that perform nonrelated work and do not communicate each other. Using multithreaded workloads composed of independent applications is still a frequently used technique by computer architecture researchers. We consider that evaluation methodologies for parallel applications are a really interesting topic for future research, but it is out of the scope of this work.

It should also be noted that in order to determine the required duration for each thread, FAME analyzes the behavior of every program in isolation. Our assumption here is that the behavior of each program in isolation is similar to its behavior when executed as part of a workload, since the code signatures do not change [5].

In our experiments, we measure per-thread IPC. It can be proven mathematically that the maximal error that can be obtained with any metric, like throughput, weighted speedup or harmonic mean, for a given workload is lower-or-equal to the maximal error incurred in measuring the per-thread IPC. Therefore, all metrics will suffer from lower error than the error in per-thread IPC.

In order to correctly measure the performance of a multithreaded processor, it would be desirable that the baseline performance was obtained with the measurements taken when the processor reaches a *steady state* since, in this state, the variation of performance is negligible. In our real processor environment, we have measured that the steady state is reached when every program is reexecuted, at least, 20 times in a workload. Hence, we calculate the error of every thread in a workload for every methodology using (5), in which $T_iIPC_{steady\_state}$ is the IPC of thread i for the baseline, and $T_iIPC_{methodology}$ is the IPC of thread i reported by the methodology under study

$$ErrorT_i = \frac{T_iIPC_{steady\_state} - T_iIPC_{methodology}}{T_iIPC_{steady\_state}}(\%). \tag{5}$$

5 Analysis of the Methodologies

In [5], we analyzed the behavior of existing evaluation methodologies. Our results show that in the case of simulation environments, current methodologies cannot ensure full representativeness of every program in a workload, which can lead to unfair comparisons between different simulator setups. These representativeness and fairness problems are also present in real multithreaded processors [3], [12], [13].

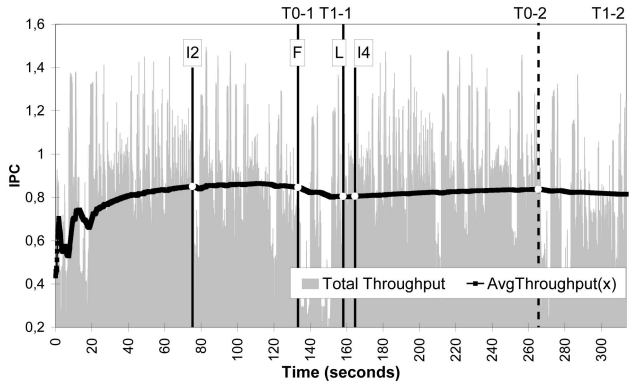

Fig. 2. IPC of gap and gcc when executed together on an Intel Pentium 4.

We made a similar experiment on our real processor environment. Fig. 2 shows the performance throughput of the gcc and gap benchmarks when they are executed together on a Pentium 4 processor. The light-gray bars show instant throughput, that is, the sum of the instant IPCs of both benchmarks. 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 vertical solid lines show the cycle in which the workload execution ends according to each experimental methodology. The white circles over each black line show the final throughput reported by that methodology. Finally, the vertical dashed lines show the time instant at which every instance of a program finishes. Above each line, we add a legend in the form $Tx-y$, in which x indicates the program and y indicates the number of times a program x has been executed.

We observe that the real throughput value varies depending on the used methodology. The results point out that the lowest value is 0.8 (L and I_{400} methodologies) and the highest value is 0.85 (I_{200} methodology), which shows that using different methodologies involves obtaining different results.

Fig. 3 shows the instant IPC of crafty and vpr sampled every 100 milliseconds. The benchmark vpr is a representative example of an application in which the instant IPC varies noticeably, while the benchmark crafty is a representative example of an application in which the instant IPC does not vary significantly. As before, the light-gray bars and the black line represent the instant IPC and the average IPC of the given benchmark, respectively. The final IPC is the average IPC at the end of the evaluation. Intuitively, in order to fulfill a given MAIV, it would be necessary to reexecute more times vpr than crafty, since its average IPC presents more variability. From this information, we obtain $CycleMax_2$ and $InstMax_2$ and compute the number of reexecutions i required to satisfy a given MAIV.

Table 1 shows the minimal number of reexecutions required for both SpecInt and SpecFP with MAIV values ranging from 20 to one percent for (a) the Pentium 4 and (b) the Intel Dual Core. The lower the MAIV value is, the higher accuracy required, and thus, in most cases, the more repetitions are needed. For

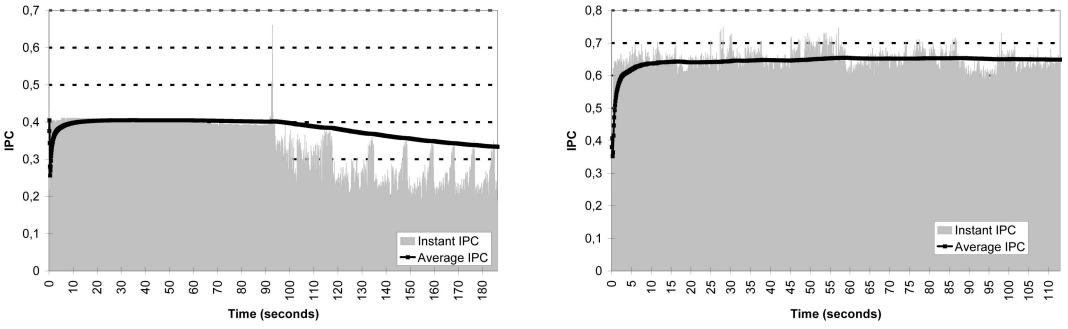

Fig. 3. Instant and average IPC of two benchmarks executed in a Pentium 4. (a) vpr; (b) crafty.

TABLE 1
Number of Repetitions Required for Every SPEC2K Benchmark

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

Spec CPU INT

Bench. Name	MAIV(%) 20	10	5	2	1
ammp	1	1	1	2	3
applu	1	1	1	1	1
apsi	1	1	1	1	1
art	1	1	1	1	1
equake	1	1	2	4	7
facerec	1	1	1	1	1
fma3d	1	1	1	1	1
galgel	1	1	1	1	1
lucas	1	1	1	1	1
mesa	1	1	1	1	1
mgrid	1	1	1	1	1
sixtrck	1	1	1	1	1
swim	1	1	1	1	1
wupwise	1	1	1	1	1

Spec CPU FP

(a)

Bench. Name	MAIV(%) 20	10	5	2	1
bzip2	1	1	2	2	3
crafty	1	1	1	1	1
eon	1	1	1	1	1
gap	1	1	2	5	5
gcc	1	1	2	3	7
gzip	1	1	1	1	3
mcf	1	1	2	5	7
parser	1	1	1	1	1
perl.	1	2	3	4	8
twolf	1	1	1	1	1
vortex	1	1	1	1	1
vpr	1	1	2	5	10

Spec CPU INT

Bench. Name	MAIV(%) 20	10	5	2	1
ammp	1	1	1	1	2
applu	1	1	1	1	1
apsi	1	1	1	1	1
art	1	1	1	1	1
equake	1	1	2	4	8
facerec	1	1	1	1	1
fma3d	1	1	1	1	1
galgel	1	1	1	1	1
lucas	1	1	1	1	1
mesa	1	1	1	1	1
mgrid	1	1	1	1	1
sixtrck	1	1	1	1	1
swim	1	1	1	1	1
wupwise	1	1	1	1	1

Spec CPU FP

(b)

(a) In the Intel Pentium 4. (b) In the Intel Dual Core.

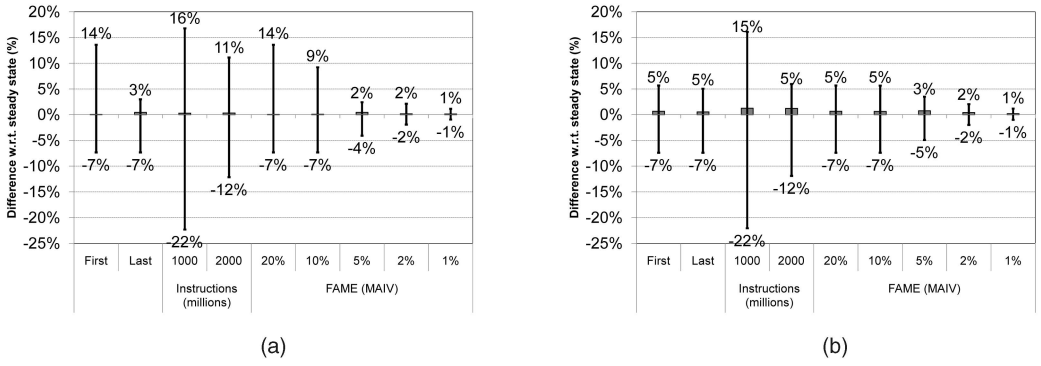

Fig. 4. Per-thread error in an Intel Pentium 4 with Hyperthreading. (a) Pentium 4 hardware context 0. (b) Pentium 4 hardware context 1.

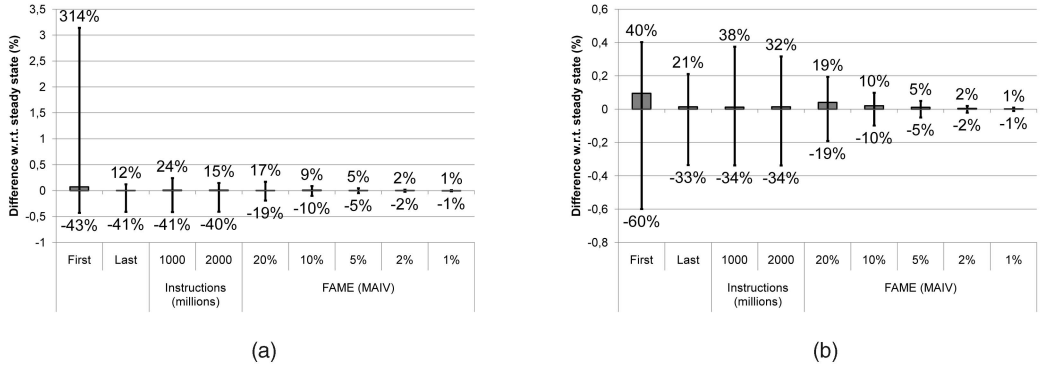

Fig. 5. Per-thread error in a Dual Core Intel Xeon 5130. (a) Core 0; (b) Core 1.

example, if a MAIV value of less than one percent is required, some benchmarks (gap, gcc, mcf, perl, vpr, and equake) have to be reexecuted more than five times to be accurately represented in the workload. It is also noticeable that when the MAIV requirements are relaxed (five percent), only one repetition is needed in most of the SPECs.

Once the minimal number of repetitions is obtained, workload executions can begin. Workload will not finalize until every program in the workload has been executed, at least, as many times as the minimal number of repetitions required for accurate representativeness. If any program reaches this minimal number of repetitions before the rest of the programs, it will reexecute once and again until all programs fulfill their requirements. This is not a problem for representativeness, since the maximum difference between the average and the final IPC can only decrease. When all programs 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, in the workload composed by gcc and vpr, requiring a MAIV of one percent, gcc and vpr must be reexecuted at least seven and ten times, respectively. If gcc finishes first, it must be reexecuted 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, execution finalizes.

To use FAME in real scenarios, every benchmark in a workload should be reexecuted until the desired MAIV is reached. To do that, FAME requires access to the performance counters in the processor, so that the IPC of every benchmark in the workload can be computed whenever one of them finishes its execution.

The memory hierarchy is flushed before each program's reexecution because the OS allocates a new process address to execute another instance of the same application. Thus, the thread memory footprint corresponding to the program reexecuted is erased by the OS. In simulation environments, when a program 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 found that, for our simulation environment [5], the initialization part (the first instructions executed after a context switch) 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 percent for all cases.

6 Accuracy Evaluation

In order to show the benefits of FAME, this section provides a comparison between FAME and previous methodologies in real processors (in [5], we provided a detailed comparison of FAME against those methodologies in simulation environments). Data are presented in terms of per-thread IPC, since accurate per-thread IPC measures involves accurate measures for all currently used metrics.

Figs. 4 and 5 show the accuracy results for an Intel Pentium 4 and a Dual Core 5130 Xeon processors, respectively. These figures show the average error of every methodology with respect to the baseline (gray bars) and the maximum positive and negative errors.

The main observation from Figs. 4 and 5 is that all current methodologies present a noticeable average error. The First and Last methodologies present a significant error due to the fact that when these methodologies finalize the execution of a workload, it cannot be ensured that all programs are fairly represented in the final result. For the I_x methodologies, we observe that the more the executed instructions the less the error is, but at the cost of increasing evaluation time. FAME is the methodology that presents the lowest error, and thus 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

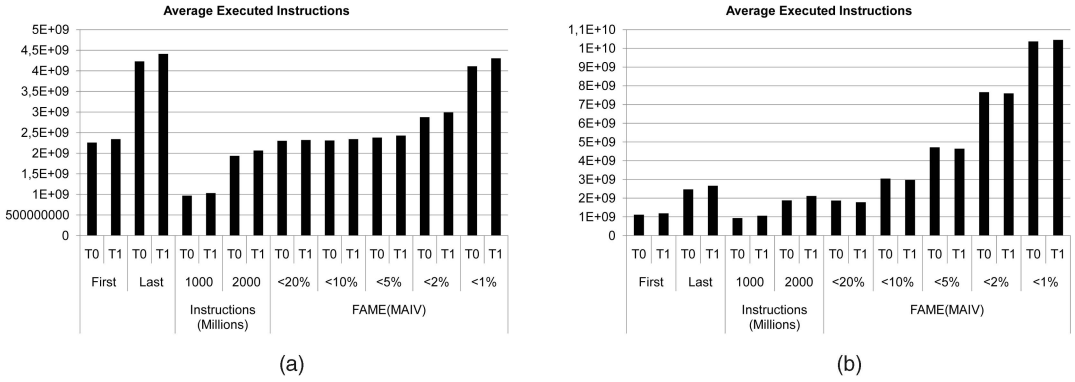

Fig. 6. Average executed instructions per context. (a) Pentium 4 scenario. (b) Dual Core scenario.

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.

This shows that there is a clear trade-off between the number of instructions a methodology executes and the error it obtains. It depends on the researcher to decide about this trade-off, but it should be noted that even using a five percent MAIV value will require much lower execution time than the baseline execution, just having a negligible $+/-5$ percent of maximal error. We do really believe that a MAIV of five percent is a good trade-off between representativeness and evaluation overhead.

The key point here is that FAME adapts the finalization moment of a workload to the behavior of the programs that compose that workload. Hence, if a program presents an invariant IPC, FAME executes few instructions for a lower error. For example, when executing the workload $eon + twolf$ in the Intel Pentium 4 scenario, in which both threads have a plain IPC, FAME with a MAIV five percent executes only 7.1 billions of instructions and leads to an error of 0.05 percent. The $Last$ methodology obtains the same error but executing $2.5\times$ more instructions. On the other hand, programs with higher IPC variance need to be reexecuted several times in order to ensure a fair measurement. For example, in the workload $eon + perlbmk$ in the Dual Core scenario, $perlbmk$ has a high IPC variance what makes FAME (MAIV five percent) execute $2\times$ more instructions than the $I_{2,000}$ methodology to obtain the same error (0.6 percent).

Fig. 6 shows the average number of instructions executed with every methodology per hardware context (T0 and T1), Fig. 6a for the SMT configuration and Fig. 6b for the CMP configuration. We observe that in the SMT scenario, FAME always executes less instructions than the $Last$ methodology and nearly the same amount of instructions than the $I_{2,000}$ methodology for MAIVs of 20, 10, and five percent. FAME (MAIV two percent) only executes 50 percent more instructions than the $I_{2,000}$ methodology. In the CMP scenario (Fig. 6b), FAME (MAIV 20 percent) obtains, on average, less error (0.11 percent) than the $Last$ (0.2 percent) and the $I_{2,000}$ (0.36 percent) methodologies executing 25 and seven percent less instructions, respectively.

7 Related Work

Several methodologies and metrics have been proposed for measuring the performance of multithreaded processors executing noncooperative workloads. On the one hand, evaluation methodologies determine how to take measurements from a workload. On the other hand, metrics compute a representative value from the measurements obtained using an evaluation methodology.

In this paper, we have evaluated the First, the Last, and the Fixed Instructions methodologies, which have been already explained in previous sections. FAME is an evaluation methodology that provides more accurate measurements than any of the aforementioned methodologies.

Commonly used metrics are throughput [2], harmonic mean [7], and weighted speedup [6]. According to [14], both harmonic mean and weighted speedup have system-level meaning in multiprogram environments, while throughput has not. In particular, harmonic mean represents user-perceived performance and weighted speedup represents system-perceived performance, so combining both metrics provides insight into the trade-off between single-program performance and overall system performance. In this paper, we have focused on per-thread IPC instead of on any particular metric because, as mentioned before, per-thread IPC is the only variable parameter used to compute any of these metrics. In other words, showing accuracy for per-thread IPC demonstrates accuracy for any of these metrics.

The most related work to FAME is the cophase matrix [15]. A qualitative comparison between FAME and cophase matrix for simulated environments can be found in [5]. In real scenarios, the cophase matrix cannot be easily applied. The main problem to port cophase to a real processor scenario is the implementation of the checkpointing mechanism needed to start the execution of one phase of a thread. This means that an operating system should provide a mechanism to restore the whole memory image of a process in a given point. On the other side, cophase makes a fast-forward of 1.5M instructions per thread in a phase to warm up memory structures, which is impossible to perform in a real processor. Furthermore, if cophase is modified to avoid this fast-forward by executing instructions, it cannot be ensured that both threads in a phase reach, at the same time, the segment of code to evaluate.

8 Conclusions

Research in multithreaded processors is becoming more and more popular due to current industrial trends. In these researches, an appropriate evaluation methodology is mandatory to guarantee the accuracy of the results. However, current methodologies do not ensure the representativeness of the obtained results and, even worse, they do not ensure that a fair comparison can be done between different experiments.

FAME is a novel evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME is mainly based on representative program reexecution. The basic idea behind our proposal is that, when a program is reexecuted enough times, its average performance converges to a representative result. Therefore, once all programs in a workload are executed a required number of times, representativeness is assured.

Any metric can use the measurements obtained with FAME because a methodology just dictates how to take measurements and not how to use them. Furthermore, 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 simulation and real scenarios.

As a case study, we apply FAME to real SMT and CMP processors. In both cases, FAME achieves better accuracy than traditional evaluation methodologies, proving that FAME is a worthwhile contribution for the evaluation of future multithreaded processors.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain under contracts TIN2007-60625 and CSD2007-00050, the HiPEAC European Network of Excellence, the Barcelona Supercomputing Center, and an Intel fellowship. The authors would like to thank Javier Vera, Jaume Abella, and Beatriz Otero for their valuable help during the development of this work.

REFERENCES

[1] M. Serrano, R. Wood, and M. Nemirovsky, "A Study of Multistreamed Superscalar Processors," Technical Report #93-05, Univ. of California, 1993.
[2] 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," *Proc. 23rd Int'l Symp. Computer Architecture,* 1996.
[3] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, pp. 4-15, 2002.
[4] http://www.specbench.org/, 2010.
[5] J. Vera, F. Cazorla, A. Pajuelo, O.J. Santana, E. Fernández, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques,* 2007.
[6] A. Snavely, D. Tullsen, and G. Voelker, "Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreaded Processor," *Proc. Int'l Conf. Measurements and Modeling of Computer Systems,* 2002.
[7] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software,* 2001.
[8] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "Dynamically Controlled Resource Allocation in SMT Processors," *Proc. 37th Int'l Symp. Microarchitecture,* 2004.
[9] E. Tune, R. Kumar, D. Tullsen, and B. Calder, "Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy," *Proc. 37th Int'l Symp. Microarchitecture,* 2004.
[10] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec, "Boosting SMT Performance by Speculation Control," *Proc. 17th Int'l Parallel and Distributed Processing Symp.*, 2003.
[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. 36th Int'l Symp. Microarchitecture,* 2003.
[12] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner, "POWER5 System Microarchitecture," *IBM J. of Research and Development*, vol. 49, no. 4, pp. 505-521, 2005.
[13] http://opensparc-t1.sunsource.net/, 2010.
[14] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro,* vol. 28, no. 3, pp. 42-53, May 2008.
[15] M. Biesbrouck, T. Sherwood, and B. Calder, "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation," *Proc. Int'l Symp. Performance Analysis of Systems and Software,* 2004.