Measuring the Performance of Multithreaded Processors

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 fact, many processor vendors have already shipped processors with multithreaded features. Regardless of this push on multithreaded processors, still today there is not a clear procedure that defines how to measure the behavior of a multithreaded processor.**

This paper presents FAME, a new evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME can be used in conjunction with any of the metrics proposed for multithreaded processors like IPC throughput, weighted speedup, etc. The idea behind FAME is to reexecute all threads in a multithreaded workload until all of them are fairly represented in the final measurements taken from the workload. Then these measurements will be combined with the corresponding metric to obtain a final value that quantifies the performance of the processor under consideration.

I. Introduction

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 programs. This strategy has led to a wide range of multithreaded processor architectures like SMT [4][8], CMP, or combinations of both. Currently, many of the main processor vendors have some multithreaded processor. Some examples are the Intel Pentium 4 [3] that is a dual-threaded SMT, the IBM Power5 [5] that is a dual core processor where each core is a 2-context SMT, and the Sun Niagara T1 [11] that has eight 4-context fine-grain multithreaded cores.

A. The problem of evaluating multithreaded processors

In spite of the increasing trend to use truly parallel applications, they are still less common in current multithreaded machines than single-threaded applications like SPEC CPU [12]. Therefore, computer architecture researchers frequently evaluate their proposals for multithreaded architectures using workloads composed by single-threaded applications, i.e. SPECrate consists on executing simultaneously several copies of the same benchmark. Furthermore, it is interesting to notice that for fully evaluating a wide range of scenarios, workloads composed by benchmarks with different behaviors should be used.

Working with several different programs running simultaneously involves an important decision, that is, to determine when the execution of the multi-program workload will finish. In a single-threaded processor, the full program is ran until completion. However, it is not so easy in a multithreaded processor running a workload composed by several programs. Applications in a workload can execute at different speeds due to the different features of each one, as well as the availability of the shared resources in the processor. Therefore, the usual case is that they do not complete execution at the same time.

We will explain this fact with an example. Let us assume an M-context multithreaded processor executing a 2-program workload (being M greater than or equal to 2). The execution of this workload occurs as depicted in Figure 1. Both applications 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 applications are being executed. Secondly, after the first application finishes (Application 0 in Figure 1), there is a *single-threaded period* in which the remaining application 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* may become an inaccurate metric for multithreaded processors. For instance, in an Intel Pentium 4 processor running all 2-thread combinations from SPEC2000 (see details in Section 4), the single-threaded periods represent, on average, 40% of the total execution time.

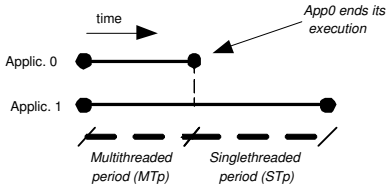


Fig. 1. Example of the execution of a 2-program workload in a M-context multithreaded processor ($M \geq 2$).

In general, the execution of an N-program workload involves N periods of N, N-1, N-2,...and 1 program respectively. A common characteristic of all the evaluation methodologies we have analyzed is that only measurements obtained from the period with N running applications are representative. Periods having less running applications than the maximum available should not be taken into account since the results could be inaccurate and misleading.

B. Proposed Solutions

In order to quantify the behavior of multithreaded processors, several *methodologies* and *metrics* have been proposed. Metrics for measuring the performance of a processor compute a value (result) for each workload that quantifies the performance of that processor when running the workload. This value is based on two inputs. On the one hand, the IPC achieved by each program in the workload, which we call ($IPC_{MT_1}, IPC_{MT_2}, \ldots IPC_{MT_N}$) for a workload of N applications. On the other hand, the IPC of each program when it is run in isolation, which we call ($IPC_{alone_1}, IPC_{alone_2}, \ldots IPC_{alone_N}$). Thus, a metric is a function $f\,(IPC_{MT_i}, IPC_{alone_i})$ where $0 \leq i \leq N$. For example, the IPC throughput is defined as $\sum_{i=1}^{N} IPC_{MT_i}$, the weighted speedup [6] is defined as $\frac{1}{N}\sum_{i=1}^{N}(IPC_{MT_i}/IPC_{alone_i})$, and the harmonic mean [2] as $N \times \left(\sum_{i=1}^{N} IPC_{alone_i}/IPC_{MT_i}\right)^{-1}$.

A methodology defines when the measurements for a given workload execution are taken. In this paper, we analyze several methodologies that have been used during the last years to measure the performance of both real multithreaded processors and simulated multithreaded processors. The main task of a methodology is to determine when the programs in a workload have to finish. We will show that previous methodologies cannot ensure that every benchmark is fully represented, and thus it is not possible to assure that the measurements obtained are representative of the whole program.

To face this problem, we present FAME, a new simulation methodology for the evaluation of multithreaded processors. Our methodology aims to ensure that every program in a workload is executed, allowing to do fair comparisons between different techniques and processor setups. As a case study, we have selected to apply FAME to a real SMT processor (Intel Pentium 4). However, FAME can be applied also to simulation environments and any other multithreaded processors. Our results show that FAME provides more accurate measurements than previously used methodologies.

II. Current Methodologies

In order to fairly evaluate the performance of an SMT processor, measurements should be obtained while all programs in a given workload are running. However, the programs 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 program finalizes its execution. Current simulation methodologies can be classified as follows:

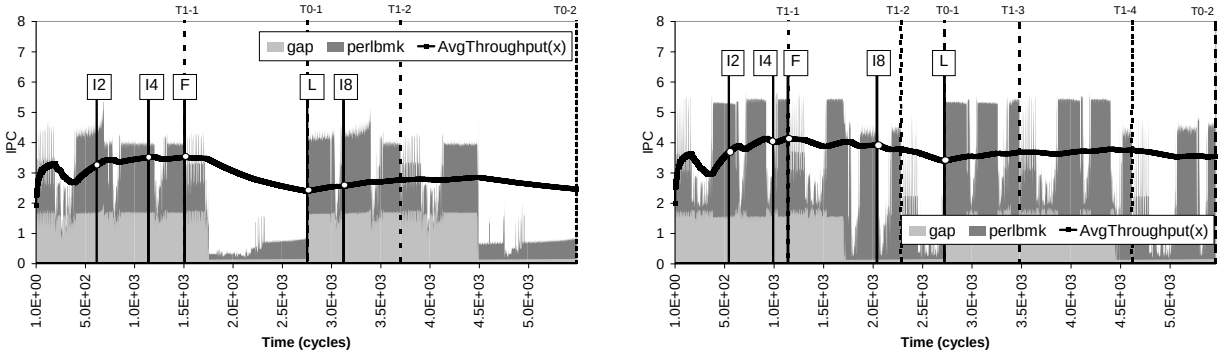
The *First* methodology finalizes the simulation of a workload when any program of the workload ends its execution [1]. The main drawback of this methodology is that only one program in the workload is executed until completion, and thus it cannot be ensured that the remaining programs execute completely, losing representativity in the final result.

The *Last* methodology finalizes workload simulation when all the programs have been run until completion. When any program ends, excluding the last one, it is reexecuted [10] while the other programs 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 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 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 program, that is, the simulation of a workload with N programs will finalize when the total number of executed instructions is N times the threshold. 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.

To show the behavior of current evaluation methodologies, we analyze these three methodologies. Without lost of generality, we have used a multithreaded simulator to collect information about these methodologies. Our simulator is a fairly parametrized 2-thread SMT processor. We implemented the methodologies 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).

Figure 2 shows the obtained results for these methodologies using our SMT simulator setup and a 2-thread workload composed by the benchmarks *perlbmk* and *gap*. The simulation ends when both programs have executed at least twice. We provide data for two well-known fetch policies: *icount* [8] and *stall* [9]. Icount, in Figure 2(a), prioritizes those programs with fewer instructions in the processor pipeline. The stall fetch policy, in Figure 2(b), uses the same heuristic, but it also detects when a program has a pending long-latency memory access. When this situation is detected, stall prevents the program from fetching more instructions until the memory access is resolved, avoiding unnecessary over-pressure over the shared resources.



(a) Results with icount

(b) Results with stall

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

In Figure 2, the y-axis shows processor performance (IPC) and the x-axis represents execution time. The light-gray bars show the instant IPC of *gap*. Likewise, the dark-gray bars show the instant IPC of *perlbmk* (To obtain the instant IPC we use a sampling period of 15K cycles). In every sample, the sum of both bars represents the instant throughput, *i.e*, the sum of the instant IPC of both programs. 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 and the vertical solid lines show the cycle in which the workload simulation ends according to each experimental 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 $Tx - y$, where x indicates the program and y the number of times a program x has been executed.

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 I(a). 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 measure the performance improvement of stall with icount as baseline (shown in the last row of Table I(a)). 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, the 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.

TABLE I

BEHAVIOR OF CURRENT METHODOLOGIES.

Methodology →		I2	I4	F	L	I8
IPC Throughput $IPC_{gap} + IPC_{perl}$	icount	3.2	3.5	3.5	2.4	2.6
	stall	3.7	4.0	4.1	3.4	3.9
stall Improvement(%)→		13.1	15.1	18.2	41.8	53.0

(a) Improvement of stall over icount using different methodologies.

	Th.	Methodology				
		I2	I4	F	L	I8
Number of full executions	T0	0	0	0	1	1
	T1	0	0	1	1	1
% of instructions (current execution)	T0	26	61	82	0	60
	T1	36	75	0	63	77

(b) number of full executions and percentage of instructions executed of the current execution

As discussed in previous sections, this problem is due to the fact that current methodologies cannot ensure fully representativity of every program of the workload, which can lead to unfair comparisons between different processor setups. For example, if we want to compare the effect of the L2 cache, which can be deactivated through the BIOS, in a Pentium 4 or if we want to determine the real performance improvement of the SMT capability of the Pentium 4.

Table I(b) summarizes these drawbacks by showing the number of times every program is 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 number 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 that the mix of instructions in every case is different, and thus, any comparison done may be misleading.

We made a similar experiment on our real processor

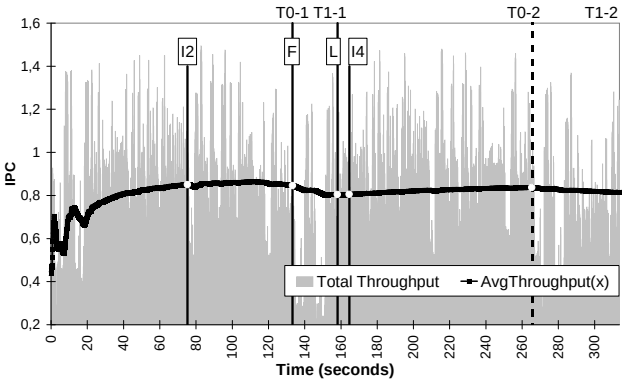


Fig. 3. IPC of *gap* and *gcc* when run together on an Intel Pentium 4

TABLE II
NUMBER OF FULL EXECUTIONS AND PERCENTAGE OF INSTRUCTIONS EXECUTED OF THE CURRENT EXECUTION

	Program number	Methodology			
		I2	F	L	I4
# of full executions	T0	0	1	1	1
	T1	0	0	1	1
% of current execution	T0	56	0	19	24
	T1	48	84	0	4

environment, obtaining the same trends. Figure 3 shows the performance throughput of the *gcc* and *gap* benchmarks when they are executed together on a Pentium 4 processor. The light-grey bars show instant throughput, that is, the sum of the instant IPCs of both benchmarks. The real throughput value varies depending on the used methodology. The lowest value is 0.8 (L and $I4$ methodologies) and the highest value is 0.85 ($I2$ methodology), which shows that using different methodologies involves obtaining different results. Table II(b) summarizes the drawbacks of current evaluation methodologies for the Pentium 4 environment. It shows the number of times each program has been completely executed and the percentage of instructions executed in the current repetition. As in the previous case the total amount of executed instructions varies from one evaluation methodology to a different one and the mix of instructions is different.

III. THE FAME METHODOLOGY

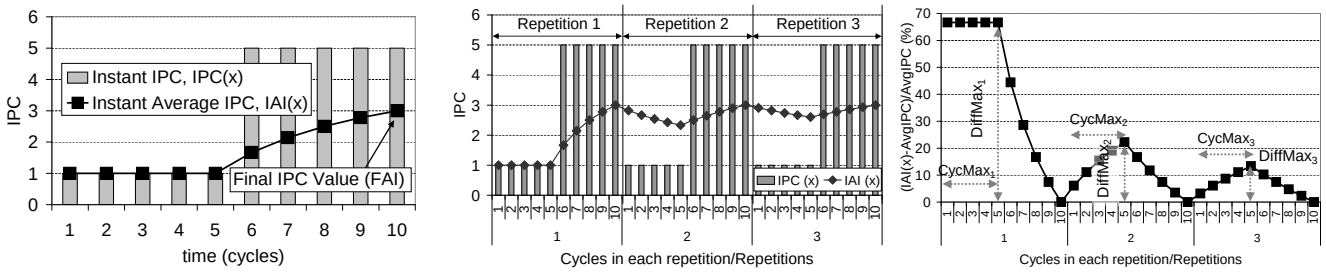
Current simulation methodologies do not ensure that all programs in a workload are faithfully represented in the simulation results. To alleviate this problem, we propose a new methodology called FAME. The main objective of our methodology is to obtain representative measurements of the actual processor behavior. In doing so, FAME determines 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 trace in isolation. In this paper we assume that the behavior of each program in a workload executed in multithread mode remains similar to the behavior in single-thread mode because the code signatures do not change. Notice, that if this assumption does not hold incurred errors will be high.

Depending on the particular methodology features, the execution of each program 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 program execution if it is similar to the final IPC value, that is, the average IPC value at the end of the whole program execution. Hence, FAME forces each program 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 4(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 is clear that the average IPC converges towards the final IPC value.

Figure 4(b) shows the instant IPC and the average IPC during three reexecutions of the application. In addition, Figure 4(c) shows the difference between the average IPC and the final IPC during the three reexecutions. It is clear that the average IPC converges towards the final IPC value. Even if that difference is a decreasing function, 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, 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 decreases monotonically, the maximum difference in a reexecution is lower for each new executed repetition. That is, it is a decreasing monotone 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



(a) IPC (x) (b) IPC during repetitions (c) Diff (%) during repetitions

Fig. 4. Instant IPC, average IPC, and *difference between both* of a synthetic program during 3 repetitions.

benchmark behavior. For this reason, our methodology reexecutes all programs several times, until the difference is upper-bounded by a given threshold.

Figure 4(c) 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 Figure 4(c) 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 higher value is always the same for all repetitions.

The first step to apply FAME is to run two repetitions of every program in isolation. Periodically, we sample the IPC of the application obtaining the IPC during execution. 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 III shows the minimal number of repetitions required per benchmark with MAIV values ranging from 20% to 1%. Once the minimal number of repetitions are obtained, workload simulations can begin.

Workload simulation 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 representativity. 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 representativity, 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.

TABLE III
NUMBER OF REPETITIONS REQUIRED FOR EVERY SPEC2K BENCHMARK ON THE INTEL PENTIUM 4.

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

(a) Spec CPU INT

Bench. Name	MAIV(%) 20	10	5	2	1
ammp	1	1	1	1	1
applu	1	1	1	4	7
apsi	2	3	7	17	35
art	1	1	1	1	1
equake	1	1	1	1	2
galgel	2	3	6	15	30
lucas	1	1	1	1	3
mesa	1	1	1	3	6
mgrid	1	1	2	5	10
swim	1	1	2	5	10
wupw.	1	1	1	2	4

(b) Spec CPU FP

IV. ANALYSIS OF EVALUATION METHODOLOGIES

To evaluate FAME in a real processor we use a 3GHz Intel Pentium 4 processor (model 531) with Hyperthreading Technology and 512 MBytes of DDRAM at 400 Mhz. The operating system is a Fedora Core 3 with gnu Linux kernel 2.6.11 patched with perfctr-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 1 to reduce as much as possible the interferences generated by multiuser-multitasking processing. Video, audio and communication hardware capabilities are disabled. Gcc 3.4.2 and the Intel Fortran Compiler 9.0 were used to compile the whole SPEC2000 benchmark suite with all optimizations enabled. Benchmarks are executed until completion with the standard reference input set. The SMT workloads were generated with all the possible combinations of 2 applications from SPEC2K, leading to 351 2-thread combinations.

In order to correctly measure the performance of a multithreaded processor, it would be desirable that the baseline performance is obtained with the measurements taken when the processor reaches a *steady state* because, in this state, the variation of performance is negligible. We measured that the steady state is reached when every program is reexecuted, at least, 20 times in a workload. Following reexecutions do not affect the results.

We measure per-thread IPC. If per-thread IPC is accurate, our FAME methodology can be used to study any metric, like throughput, weighted speedup or harmonic mean, since per-thread IPC is the only variable parameter used to

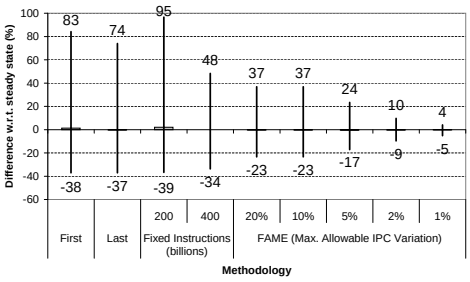


Fig. 5. Error of the different methodologies for the Pentium 4 processor

compute these metrics. We calculate the error of every thread in a workload for every methodology using the next formula, 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.

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

FAME is the methodology with the lowest error as shown in Figure 5. The worst results come from the 200-billion instruction methodology (errors range from 95% to -39%). There is a clear trade-off between the number of instructions a methodology executes and the error it obtains. Ideally, we would like to have a methodology that requires executing few iterations, while leading to a reduced error. Regarding this topic, lowest MAIV errors are achieved by an affordable execution time increase. For instance, the execution time for MAIV 20% and 10% is the same that in the Last methodology. MAIV 5%, 2% and 1% increase the execution time by 5.3%, 9.6% and 15,2% respectively.

Finally, note that the error of a methodology is independent of the metric used. Even if some metrics, like weighted speedup, are used to provide *fairness*, the results of these metrics depend on the accuracy of measurements. If measurements are wrong the results obtained by a metric are likely wrong.

V. Related Work and Conclusions

Choosing an accurate evaluation methodologies is crucial for measuring the performance of multithreaded processors. For instance, The IBM Power5 (2 cores and 2-threads per core) was evaluated using 4-thread workloads containing the same application replicated four times [5]. Since all the threads in the workload are the same program, they finalize execution almost simultaneously, which means that the error is negligible regardless the evaluation methodology used. We have found that, when we execute workloads containing a single program replicated several times, the duration of the single-threaded period is negligible, 0.03%). However, using just this type of workload limits the variety of the analysis and the evaluation that can be done. FAME could allow evaluating the Power5 processor using any arbitrary workload, since it is a more general methodology.

Another evaluation of a real SMT processor is presented in [7], where heterogeneous workloads are executed 12 times to guarantee, at least, 3 complete executions of a thread of every job. It is not explained how the number of repetitions are obtained and, since this number depends on both the simulator setup and the number and mix of threads in every workload, this methodology cannot be extrapolated to other environments. The point of FAME is that we can fix *a priori* the minimal number of repetitions per benchmark in a workload to ensure the correctness of measurements.

FAME achieves better accuracy than previously proposed evaluation methodologies, such as First, Last, and Fixed Instructions. 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 real scenarios (as presented in this paper) and simulation scenarios (an architectural simulator is used instead of a real processor).

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. The authors would like to thank Jaume Abella and Beatriz Otero for their technical comments.

References

[1] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in SMT processors. *MICRO*, 2004.
[2] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *ISPASS*, 2001.
[3] 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.
[4] M. J. Serrano, R. Wood, and M. Nemirovsky, A Study on Multistreamed Superscalar Processors. *Technical Report 93-05*, University of California Santa Barbara, 1993.
[5] 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.
[6] A. Snavely and D.M. Tullsen and G. Voelker. Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreaded Processor. *SIGMETRICS* 2002.
[7] N. Tuck and D. M. Tullsen Initial Observations of the Simultaneous Multithreading Pentium 4 Processor PACT 2003
[8] 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.
[9] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *MICRO*, 2001.
[10] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. *Proceedings of CASES*, 2005.
[11] http://opensparc-t1.sunsource.net/
[12] http://www.specbench.org/.