On the Problem of Minimizing Workload Execution Time in SMT Processors

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain. francisco.cazorla@bsc.es

Peter M.W. Knijnenburg
University of Amsterdam, The Netherlands. peterk@science.uva.nl

Rizos Sakellariou
University of Manchester, UK. rizos@cs.man.ac.uk

Enrique Fernandez
University of Las Palmas of Gran Canaria, Spain. efernandez@dis.ulpgc.es

Alex Ramirez
Universitat Politecnica de Catalunya and BSC, Spain. aramirez@ac.upc.edu

Mateo Valero
Universitat Politecnica de Catalunya and BSC, Spain. mateo@ac.upc.edu

***Abstract*—Most research work on (Simultaneous Multithreading Processors) SMTs focuses on improving throughput and/or fairness, or on prioritizing some threads over others in a workload. In this paper, we discuss a new problem not previously addressed in the SMT literature. We call this problem Workload Execution Time (WET) minimization. It consists of reducing the total execution time of all threads in a workload. This problem arises in parallel applications, where it is common for a single master thread to spawn several child jobs. The master job cannot continue until all child jobs have finished. Reducing the overall execution time is important to speedup the application. This paper is a first step in analyzing this problem. First, we analyze the WET provided by the best fetch policies aimed at improving throughput/fairness. We demonstrate that these policies achieve less than optimum performance. We show that, on average, for the workloads evaluated in this paper, there is space for improvement of up to 18 percentage points. It follows that novel mechanisms trying to reduce WET are required to speedup parallel applications.**

I. Introduction

Current processors take advantage of Instruction Level Parallelism (ILP) to execute several instructions from a single stream in parallel. However, there is only a limited amount of parallelism available in each thread, mainly due to data and control dependences [25]. As a result, hardware resources added to exploit this limited amount of ILP may be utilized only occasionally, thus significantly degrading the performance/cost ratio of these processors. A solution to overcome this problem is to share hardware resources between different threads. There exist different approaches to resource sharing, ranging from chip multiprocessors [19] to high performance SMTs [20][23][26]. In the former case, only the higher levels of the cache hierarchy are shared.

Current trends in computer architecture show that many future processors will have some form of multithreading. This is mainly due to the fact that SMT processors have a good performance/cost and performance/power consumption ratios, which makes them suitable for many types of computing systems. In high-performance systems we have processors like the Intel Pentium4 Xeon [18], the IBM Power5 [14] and the Sun Niagara T1 [1]; while in real-time embedded systems we have the Imagination Technologies Meta processor [16] and the Infineon TriCore 2 [6]. In embedded systems, in which processors must be low in cost, obtaining as much performance as possible from each resource is desirable. Hence, a viable option is a simultaneous multithreading (SMT) processor, which shares many resources between several threads for a good cost-performance tradeoff.

Given this trend in processor design, many researchers have focused their efforts on SMTs and, in particular, on policies that deal with the different objectives that the Operating System may impose on an SMT. We can distinguish two main areas of interest.

Throughput and fairness: many papers on SMT attempt to increase the total throughput and/or fairness [17] in SMT processors. In order to achieve this objective, the proposed policies focus on those situations in which the shared resources of an SMT may be used inefficiently by threads, that is, after branch mispredictions and loads that miss in the outer cache levels. In the former case, those threads that are supposed to experience a branch misprediction are stalled [15] until that branch is resolved. In the latter case, the policies act on threads that miss either in the L1 cache or in the L2 cache [7][8][9][22].

Prioritization and predictable performance: other papers show that, even when throughput and fairness are acceptable objectives, in many systems the OS may need to impose additional objectives, like prioritizing some threads in a workload [13][21] or, more restrictively, to ensure that certain threads in a workload execute at a minimum IPC speed, that is, they can finish before a given deadline [10][11].

In parallel programming environments, a common situation is that a given job spawns (or 'forks') several child jobs (threads). Every child job carries out some work and

1-4244-1058-4/07/$25.00 ©2007 IEEE

subsequently terminates. The parent job cannot continue until all the child jobs have finished. Much work has been done to balance the load of the different jobs between the different execution units available. These proposals assume that the execution units are single threaded and have similar resources. However, if multithreaded execution units are used, a different objective needs to be addressed by each of these units; namely, execute all threads assigned to each unit as fast as possible. If we assume that child jobs are executed on an SMT processor, then this last requirement represents another target objective that a system executing parallel programs can request from an SMT processor. This problem consists of reducing the total execution time of a given workload, that is, reducing the time to execute all threads in an entire workload. We call this problem *Workload Execution Time minimization* or *WET minimization*.

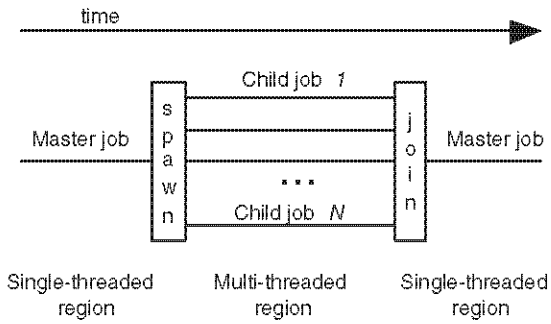
In this paper, we evaluate the WET provided by the best known policies designed to increase throughput and fairness. We also show that current metrics used to measure throughput and fairness are not adequate, and it is necessary to define new metrics. Finally, we provide some early suggestions and ideas on how we could reduce the WET.

This paper is structured as follows. Section 2 describes and motivates the problem presented in this paper. Section 3 describe our experimental environment. Section 4 shows the results achieved with the different mechanisms. Finally, we draw some conclusions in Section 5.

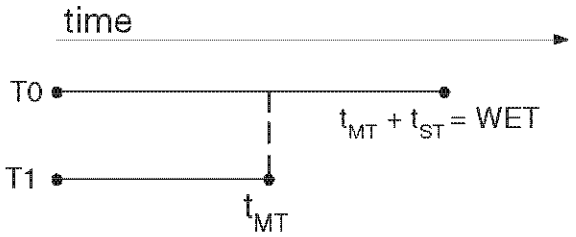
II. Workload execution time minimization: a new requirement for SMTs

The problem addressed in this paper originates from parallel execution environments where a situation as shown in Figure 1(a) is quite common. In this situation, a master job spawns several child jobs. The master job cannot continue until all the spawned jobs have finished, e.g. *barrier* and *collective* functions. If we assume that spawned jobs will be executed on an SMT, we have the following problem: given a workload with N threads, minimize the time required to execute all these threads. Note that if the number of spawned threads is larger than the number of available contexts in the SMT, another policy is needed to assemble workloads for the SMT. This problem is called the *workload selection problem* [13]. However, this problem is beyond the scope of the present paper.

The problem mentioned above occurs in parallel execution environments at different granularities. By *granularity* we mean the number of instructions executed before communication between processes is required. At one extreme, we may have loop parallelization where child jobs execute blocks of iterations, each block containing (in the order of) thousands of instructions. At a larger granularity, we have parallelization using OpenMP [4] or MPI [2] where large sections of code are running in parallel. For example, using Paraver [3], it has been shown that 10 iterations of



(a) Thread spawning: a common situation in parallel execution environments



(b) WET for a workload with two threads

Fig. 1. Thread spawning and WET for a workload with 2 threads

the NAS BT [5] benchmark execute approximately 3 billion instructions. If this program is parallelized using 4 child jobs, each child job executes 0.75 billion instructions. In this case, traces show that from 44 million to 77 million instructions are executed between points where synchronization is required. Parallelization may also occur at the subroutine or at the task level. In this paper, we assume parallelization with a large granularity, which means that each child job executes several millions of instructions.

A. Workload Execution Time (WET)

Workload Execution Time (WET) is defined as the amount of time required to execute all threads in a given workload. That is, the time until the last thread in the workload finishes. For workloads with 2 threads, T_0 and T_1, this interval can be split in two periods, as shown in Figure 1(b). The first period is called the *Multi-Threaded (MT) period* and the second period is called the *Single-Threaded (ST) period*. During the multi-threaded period, both threads are executed in multi-threaded mode until the fastest thread ends. During the subsequent single-threaded mode, the remaining thread is executed in isolation until it ends.

Let $t_x^{isolated}$ be the time that thread x requires to be executed on the SMT in isolation, that is, its execution time when it is run alone on the machine. Then the possible values of WET are in the interval:

$$\left[\max\left(t_{T_0}^{isolated}, t_{T_1}^{isolated}\right), \ldots, t_{T_0}^{isolated} + t_{T_1}^{isolated}\right]$$

The following observations can be made.

- The minimum time required to execute a workload equals $t_{slowest}^{isolated}$, where $slowest$ denotes the slowest thread in the workload, thread T_0 in Figure 1(b). That is, we execute the workload so that the slowest thread is not delayed with respect to the time it requires when it is run in isolation. In addition, the fastest thread is executed using the resources not used by the slowest thread. We assume that between synchronization points threads do not share the same address space as it is the case for example in MPI applications. That is, a thread cannot 'prefetch' data to another thread, and hence the minimum time to execute a thread, T_i, in multi-threaded mode is $t_{Ti}^{isolated}$.
- If we assume that the SMT always provides performance improvement, that is, executing several threads in SMT mode is more effective than executing them in sequential mode, then the upper bound of WET equals $t_{T_0}^{isolated} + t_{T_1}^{isolated}$.

Current fetch and resource allocation policies are designed either to increase throughput/fairness or to provide certain guarantees to a given high priority thread. In both cases, a common characteristic of existing research is that during the evaluation of proposed techniques, the number of running threads remains constant during the whole execution. In the former case, the simulation ends when the fastest thread ends and at that point throughput/fairness is computed. In the latter case, when the high priority thread ends, the simulation ends. If any of the low priority threads ends earlier, it is re-executed.

In the evaluation of our solution to the problem addressed in this paper, the situation is different as the number of running threads can change dynamically. For a workload of N threads, the number of running threads can change from N to 0 as threads gradually finish. This requires that policies dynamically need to adapt the resource allocation for the running threads as other threads finish. As a result, the problem of minimizing the workload execution time addressed in this paper is *not* equivalent to the problem of maximizing throughput.

B. WET additional remarks

In some scenarios child (spawned) threads execute the same number of instructions in order to reduce synchronization cost. In this case the problem of WET minimization is just a matter of evenly split shared hardware resources between threads so that the execute at the same speed.

However, there are also scenarios in which some threads do more work than others. Even more, in those situations in which child threads have the same number of instructions the number of L2 caches misses of each thread may vary what makes some threads take longer to execute than others. For example, let say that we are making some computations with a matrix a we divide the matrix into sub-matrices so that each child thread work on each sub-matrix. Even if the different sub-matrices have the same size, the values in each are different what may change the computation done by each thread. In this case, the WET minimization is a more complex problem.

On the other hand, in a normal situation once one thread finishes that context is free for other threads. For example, T1 in the situation depicted in figure 1(b) finishes before T0, so the Operating System schedules a new task in that context. In that case the WET minimization problem is even worse because T0 executes with another thread till its finalization what increments its execution time. In this paper, for simplicity reasons we assume that ones the first thread in a workload finishes this context is not used by the OS to schedule a new task.

TABLE I
BASELINE CONFIGURATION

Processor Configuration	
Pipeline depth	12 stages
Number of contexts	2
Fetch/Issue/Commit Width	8
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256 integer, 256 fp
ROB size(per thread)	256
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way
RAS	256 entries

Memory Configuration	
Icache, Dcache	64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	512 Kbytes, 8-way, 8-bank, 64-byte lines, 10 cycle access
Memory latency	100 cycles
TLB miss penalty	160 cycles

III. EXPERIMENTAL SETUP

In this section, we discuss the simulator we have used in this study as well as the metrics and benchmarks we used for that purpose.

To evaluate the performance of different policies, we use a trace driven SMT simulator derived from smtsim [24]. The simulator consists of our own trace driven front-end and an improved version of smtsim's back-end. The simulator allows executing wrong path instructions by using a separate basic block dictionary that contains all static instructions. Table I shows the main parameters of the simulated processor. This processor configuration represents a standard and fair configuration of a 2-context SMT according to state-of-the-art papers.

In this paper, we compare the WET of different fetch policies oriented toward throughput improvement. For this initial, proof-of-concept study, we did not use parallel programs. Instead, we used independent single-threaded SPEC

CPU 2000 benchmarks. We assume that the 'fork' operation has already been performed by the master job, and that these independent programs are the child jobs. In an MPI environment we assume that what we execute is the code between to communication points. If the number of spawned threads is greater than the number of available contexts in the SMT, two in our case, a *job balancer* balances the jobs over the available processing units. For each experiment, we consider two jobs that have to be executed as fast as possible. The simulation ends when both threads have finished.

A key parameter in our simulations is the ratio between the execution time of threads when executed in isolation. We call this measure *execution time ratio* or *ETratio*. This ratio is defined in Equation (1).

$$ETratio = \frac{t_{T0}^{isolated}}{t_{T1}^{isolated}} \tag{1}$$

where ET_{thread_x} denotes the execution time of $thread_x$ when run in isolation.

We use the integer and floating point SPEC CPU 2000 benchmarks. We have built 48 different workloads based on their ETratios. We have split workloads into 2 different groups, depending on this parameter. Without loss of generality, in all experiments we assume that T_0 has a longer execution time than T_1. For the first group, the ETratios of the threads in a workload are in the range $[1, \ldots, 2)$. This means that their execution times do not differ significantly. In the second group, the range of ETratios is $[2, \ldots, \infty)$. In this group, the difference between the execution times of both threads is higher. Based on these parameters, we built the workloads shown in Tables II(a) and II(b). In order to obtain different ETratios, we vary the number of instructions a thread has to execute.

A. Metrics

Obviously, the ultimate metric we use is execution time. For example, if we want to know the execution time improvement that a policy P_A achieves over a policy P_B we have to compute

$$ET_{reduction} = \left(1 - \frac{ET_{PA}}{ET_{PB}}\right) \cdot 100\% \tag{2}$$

In some situations, it may happen that the ETratio is very low due to the characteristics of the threads under consideration. For example, assume that we have a workload with two threads, T_0 and T_1, whose execution times when executed in isolation is 100 million and 1 million cycles, respectively. The maximum variation (in millions of cycles) of the WET is given by

$$\left[\max\left(t_{T_0}^{isol}, t_{T_1}^{isol}\right) \leq WET \leq \left(t_{T_0}^{isol} + t_{T_1}^{isol}\right)\right] \tag{3}$$

In this case, $[\max(100, 1) \leq WET \leq 100 + 1] = [100 \leq WET \leq 101]$ and hence the *maximum improvement* that the policy P_A may achieve over the policy P_B is 1%. This could hide the real improvement of a policy over other policies. This effect is more profound when ET_{P_A} is much greater ET_{P_B}, or *vice versa*, that is, as the difference $|ET_{P_A} - ET_{P_B}|$ increases.

IV. Results

In this section, we first evaluate some existing approaches. Next, we discuss the relation between throughput and WET. Finally, we discuss how much space for improvement existing approaches leave.

A. Existing approaches

Here, we show the results obtained from the best known fetch and resource allocation policies that improve throughput/fairness. We have evaluated the following policies: icount [23], dwarn [8], stall [22], flush [22], flush++ [7], and dcra [9].

The *Icount* policy prioritizes threads with fewer instructions in the pre-issue stages, and presents good results for threads with high ILP. However, SMT has difficulties with threads that experience many loads that miss in L2. When this situation happens, then icount does not realize that a thread can be blocked on an L2 miss and will not make forward progress for many cycles. Depending on the amount of instructions dependent of the blocked load, many processor resources may be blocked and the total throughput suffers from a serious slowdown.

Stall is built on top of *icount* to avoid the problems caused by threads with a high cache miss rate. It detects that a thread has a pending L2 miss and prevents the thread from fetching further instructions to avoid resource abuse. However, L2 miss detection already may be too late to prevent a thread from occupying most of the available resources. Furthermore, it is possible that the resources allocated to a thread are not required by any other thread, and so the thread could very well continue fetching instead of stalling, producing resource under-use.

Flush is an extension of stall that tries to correct the case in which an L2 miss is detected too late by deallocating all the resources of the offending thread, making them available to the other executing threads. However, it is still possible that the missing thread is being punished without reason, as the deallocated resources may not be used (or fully used) by the other threads. Furthermore, by flushing all instructions from the missing thread, a vast amount of extra fetch and power is required to redo the work for that thread.

Flush++ is based on the idea that stall performs better than flush for workloads that do not put a high pressure on resources, that is, workloads with few threads that have high L2 miss rate. Conversely, flush performs better when a workload has threads that often miss in the L2 cache, and hence the pressure on the resources is high. flush++ combines flush and stall: it uses cache behavior of threads

TABLE II
WORKLOAD DESCRIPTION

Workloads		$IPC_{T_x}^{isolated}$		Instr. (Millions)		ET-ratio
Th. 0	Th. 1	Th. 0	Th. 1	Th. 0	Th. 1	
bzip2	bzip2	4.111	4.111	300	300	1.00
crafty	perl	2.87	3.665	300	300	1.28
gap	gcc	2.054	2.569	300	300	1.25
vortex	eon	2.233	3.141	300	300	1.41
bzip2	applu	4.111	2.729	300	200	1.00
gzip	mgrid	2.855	2.051	300	150	1.44
mesa	vortex	3.871	2.233	300	100	1.73
wupwise	gap	3.092	2.054	300	200	1.00
twolf	twolf	1.286	1.286	300	300	1.00
vpr	twolf	1.254	1.286	300	300	1.03
vpr	parser	1.254	1.614	300	300	1.29
mcf	twolf	0.147	1.286	57	300	1.66
lucas	mcf	1.037	0.147	300	33	1.29
twolf	art	1.286	1.565	300	300	1.22
vpr	equake	1.254	0.905	300	200	1.08
lucas	parser	1.037	1.614	200	300	1.04
twolf	eon	1.286	3.141	208	300	1.69
vpr	vortex	1.254	2.233	300	280	1.91
gzip	twolf	2.855	1.286	300	100	1.35
parser	perl	1.614	3.665	150	300	1.14
mgrid	twolf	2.051	1.286	300	150	1.25
galgel	vpr	2.305	1.254	300	150	1.09
parser	art	1.614	1.565	300	150	1.94
swim	gcc	1.273	2.569	180	300	1.20

(a) Execution ratio $[1, \ldots, 2)$ (range 1)

Workloads		$IPC_{T_x}^{isolated}$		Instr. (Millions)		ET-ratio
Th. 0	Th. 1	Th. 0	Th. 1	Th. 0	Th. 1	
vortex	perl	2.233	3.665	300	200	2.46
gcc	bzip2	2.569	4.111	300	200	2.40
vortex	gzip	2.233	2.855	300	100	3.84
gap	crafty	2.054	2.87	300	70	5.99
gcc	applu	2.569	2.729	300	100	3.19
vortex	wupwise	2.233	3.092	300	200	2.08
fma3d	eon	2.305	3.141	300	200	2.04
apsi	crafty	2.943	2.87	300	100	2.93
twolf	vpr	1.286	1.254	300	100	2.93
vpr	parser	1.254	1.614	300	100	3.86
twolf	twolf	1.286	1.286	300	100	3.00
mcf	parser	0.141	1.614	160	300	6.10
swim	twolf	1.273	1.286	300	80	3.79
art	parser	1.565	1.614	300	80	3.87
mcf	equake	0.141	0.905	200	300	4.28
vpr	lucas	1.254	1.037	300	100	2.48
twolf	bzip2	1.286	4.111	300	200	4.80
vpr	eon	1.254	3.141	300	100	7.51
mcf	gcc	0.141	2.569	134	300	8.14
parser	crafty	1.614	2.87	300	200	2.67
vpr	mesa	1.254	3.871	300	200	4.63
galgel	twolf	2.305	1.286	300	80	2.09
perlbmk	apsi	3.665	2.943	300	100	2.41
fma3d	gap	2.305	2.054	300	100	2.67

(b) for execution ratio $[2, \ldots, \infty)$ (range 2)

to switch among flush and stall in order to provide better performance.

Unlike previous policies, *dwarn* does not squash instructions in the pipeline. Furthermore, it adapts to pressure on resources reducing resource underuse. Dwarn uses L1 data cache misses as indicators of a possible L2 miss. Threads experiencing an L1 data cache miss are given lower fetch priority than threads with no data cache misses. The key idea is to prevent the damage before it occurs, instead of waiting until an L2 miss is produced, when probably some damage has already been done.

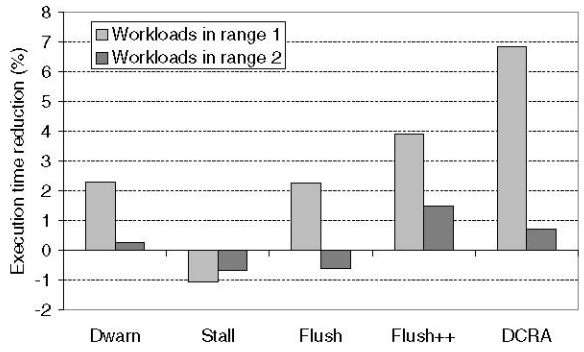
Dcra dynamically partitions resources based on memory performance. Threads with frequent L1 cache misses are given large partitions, allowing them to exploit parallelism beyond stalled memory operations. Threads that cache-miss infrequently are guaranteed some resource share since stalled threads are not allowed beyond their partitions. Hence, dcra prevents resource clog by containing stalled threads. Moreover, dcra computes partitions based on the threads anticipated resource needs, increasing distribution to the threads that can use resources most efficiently.

Figures 2(a) shows the reduction in WET that the policies considered in this section achieve over icount. These reductions are averaged over the 48 different workloads discussed above. We can see that on average all policies improve icount, except stall, although the differences are small. We observe from Figure 2(a) that the difference between the policies for low ETratios is higher than for high ETratios. It follows from Equation (3) that the maximum improvement a policy can achieve over another policy is lower for workloads with a high ETratio.

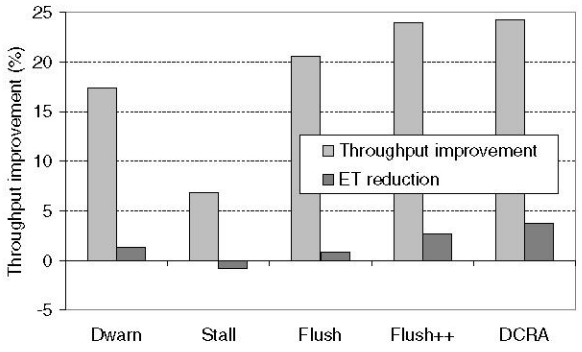
B. Correlation with throughput

In this section, we discuss the relation between throughput, that is, *the sum of the IPCs of the threads in the multithreaded period*, and WET. Figure 2(b) shows the throughput improvement of each policy over icount. Analogously, Figure 2(b) shows the WET reduction of each policy over icount. We see that, roughly speaking, the higher the throughput improvement of a policy is, the higher its WET reduction is. However, we can see that this is not always the case. For example, if we compare icount and stall, we observe that stall achieves higher throughput than icount, but icount achieves a better Workload Execution Time than stall. This indicates that improving throughput does not necessarily lead to the shortest WET. In other words, given two policies P_A and P_B, if P_A achieves higher throughput than P_B during the MT period, this does not necessarily imply that the WET of P_A is better than the WET of P_B.

Now we show a real-life example. We have two benchmarks mcf and gcc. mcf runs for 134 million instructions at an IPC of 0.147. Analogously, gcc runs for 300 million instructions at an IPC of 2.569. In this case, we execute a workload composed of *gcc* and *mcf* using the fetch policies stall and flush. The execution proceeds as shown in Figure 3. We observe that during the MT period, which takes 148 million cycles, the flush policy obtains an IPC of $0.106 + 2.298 = 2.404$. For the stall policy, the MT period takes 128 million cycles and the throughput is $0.118 + 1.984 = 2.102$. Hence, stall suffers a slowdown of 8.4% in throughput compared to flush. However, we observe that stall reduces WET by 0.5% compared to flush. This



(a) Workloads with ETratio in range 1 and range 2



(b) Throughput improvement and WET reduction for all workloads

Fig. 2. Comparing throughput and WET for several policies

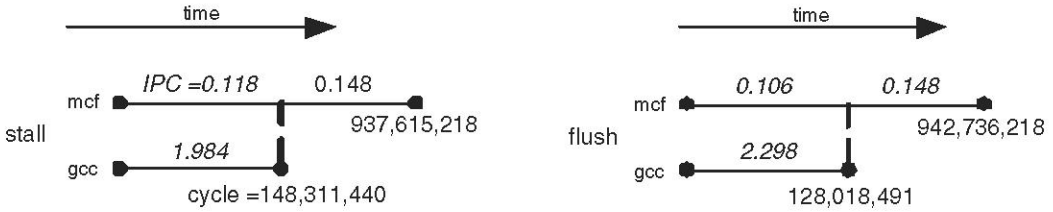


Fig. 3. Execution of the mcf and gcc benchmarks using the flush and stall policies

indicates that reducing WET is not always just a matter of increasing the throughput of the workload during the MT period. Hence, new mechanisms, other than those improving throughput, have to be proposed in order to provide reduced WET.

C. The Need for WET Optimization

The next point is to determine by how much the WET can *potentially* be reduced. This is an important point since it could be the case that the WET obtained from standard throughput oriented policies is already (almost) the shortest WET that can be obtained, in which case, it would not make sense to devise any WET oriented policies. If we assume that executing several threads in MT mode is more efficient than executing them in sequential mode, then the upper bound for the WET is $t_{T_0}^{isolated} + t_{T_1}^{isolated}$. Hence, values of the WET are in the interval:

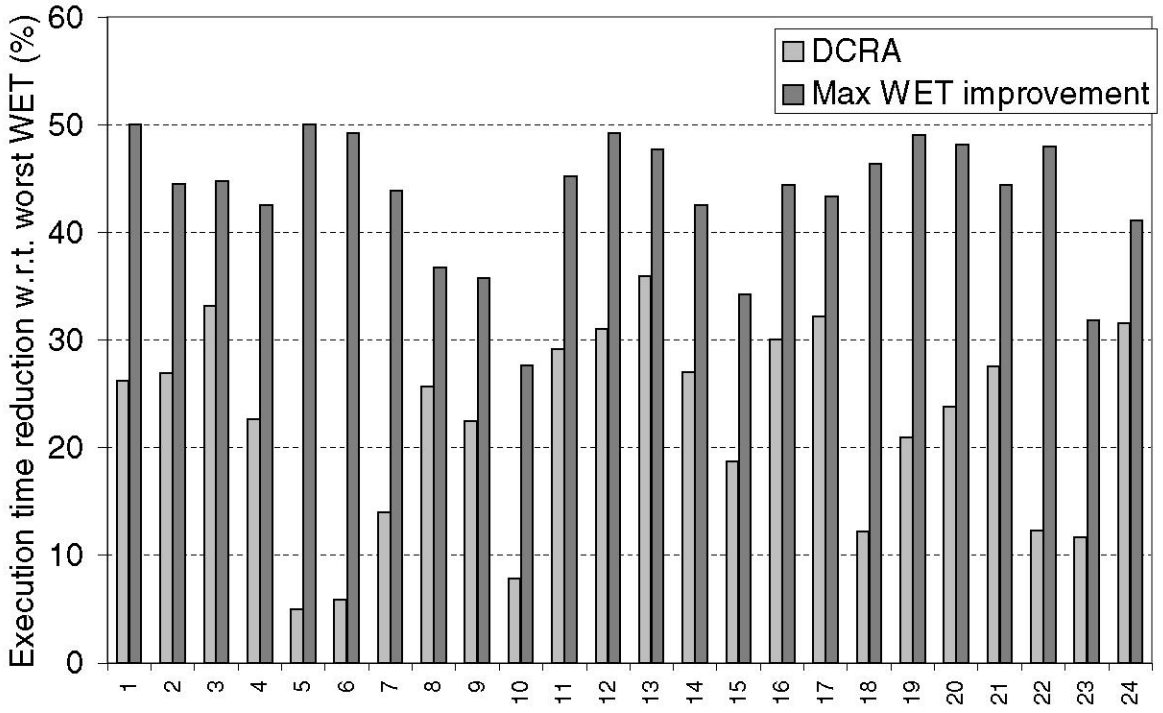
$$\left[\max\left(t_{T_0}^{isol.}, t_{T_1}^{isol.}\right) \leq WET \leq \left(t_{T_0}^{isol.} + t_{T_1}^{isol.}\right)\right]$$

The maximum WET improvement is given by:

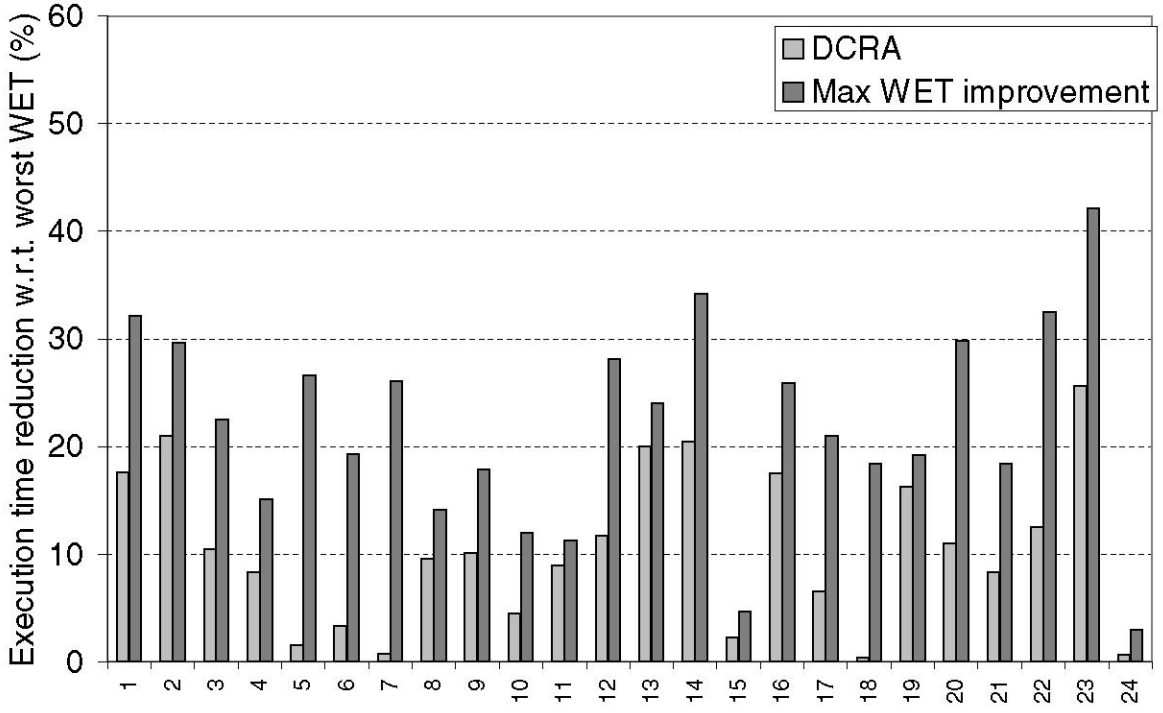
$$MaxWET_{improv} = \left(1 - \frac{\max\left(t_{T_0}^{isol.}, t_{T_1}^{isol.}\right)}{t_{T_0}^{isol.} + t_{T_1}^{isol.}}\right) \quad (4)$$

Figure 4 shows the WET achieved by the best evaluated policy, dcra, and the $MaxWET_{improvement}$ given by Equation (4). We see that in some workloads, dcra is near the maximum improvement of 2%, although in others the difference is significant. For example, for workload 5 in Figure 4(a) the difference is $50 - 5 = 45$ percentage point. On average, the difference between dcra and the best possible improvement is 24 percentage point for the workloads in range 1, and 12 percentage point for workloads in range 2, with an average of 18 percentage point. We conclude that there is a lot of space for improvement if we want to reduce Workload Execution Times, given the state-of-the-art in instruction fetch and resource allocation policies. New policies could improve WET by as much as 20 percentage point on average.

Please, note that the solution to WET minimization is not as simple as giving high priority to the thread that spends more time to be executed. This is because as noted in [12] when prioritizing a thread we can lead to a situation where performance is heavily affected. That is, let us say that we have two threads T0 and T1 where T0 requires longer to be executed and that we execute both thread at the same time on an SMT processor and T0 runs at IPC_{T0}^{SMT} and T1 at IPC_{T1}^{SMT}. Further assume that we design a policy that bias the execution towards T0 so that it finishes earlier. In that case T0 runs at $IPC_{T0}^{SMTbiasT0}$ and T1 at $IPC_{T1}^{SMTbiasT0}$. The point is that it is likely that $IPC_{T0}^{SMTbiasT0}/IPC_{T0}^{SMT} < IPC_{T1}^{SMT}/IPC_{T1}^{SMTbiasT0}$ what means that the IPC degradation of T1 incurred by prioritizing T0 is much higher that the performance improvement of T0. In this case the WET could be worse.



(a) Workloads with ETratio in range 1



(b) Workloads with ETratio in range 2

Fig. 4. Maximum improvement and dcra improvement over the worst WET

Therefore, we believe it is important to try to find new policies that are specifically geared toward WET minimization in order to improve the applicability of SMT for OpenMP and MPI-like applications in which the threads in a workload are derived from forking within one application instead of having completely independent threads, as is currently the case.

V. Conclusion

In this paper, we have discussed a problem that has not been previously addressed in the SMT literature. The problem consists of reducing the total execution time of all threads in a workload. We call this problem WET (Workload Execution Time) minimization. Since SMTs are being used more and more in parallel systems, this problem will be important in order to deal with situations where a master thread can spawn several child threads and may proceed only when all these child threads have finished. In our analysis, we have demonstrated that the best fetch policies that improve throughput achieve less than optimum performance for this problem. We also presented new metrics to evaluate the WET reduction. For the workloads evaluated in this paper, we have shown that the space for improvement can be as high as 45 percentage point, with 18 percentage point on average. We have demonstrated that improving throughput does not always cause a reduction in WET.

VI. Future Work

This paper is just a first step toward analyzing the problem and finding solutions for reducing the WET. It remains to be discussed whether the gap between the theoretical optimum execution time and the results obtained by using the different policies investigated in the paper can be closed, i.e. whether policies leveraging the remaining potential speedup are feasible.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2004-07739-C02-01, the HiPEAC European Network of Excellence, and an IBM fellowship. The authors would like to thank Germán Rodríguez and Roberto Gioiosa for their technical comments. Authors would like to thank Oliverio J. Santana, Ayose Falcón, and Fernando Latorre for their work in the simulation tool.

References

[1] http://opensparc-t1.sunsource.net/.
[2] http://www-unix.mcs.anl.gov/mpi/.
[3] http://www.cepba.upc.es/paraver/.
[4] http://www.openmp.org.
[5] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. *In Proc. of the International Journal of Supercomputer Applications*, 1995.
[6] Max Baron. Two threads for tricore 2. *Microprocessor Report*, Sep 2003.
[7] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. *Proc. of the 5th International Symposium on High Performance Computing*, October 2003.
[8] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. DCache Warn: an I-Fetch policy to increase SMT efficiency. *Proc. of the International Parallel and Distributed Processing Symposium*, April 2004.
[9] F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in smt processors. *Proc. of the 37th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 171–182, 2004.
[10] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Predictable performance in SMT processors. *ACM International Conference on Computing Frontiers*, 2004.
[11] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Architectural support for real-time task scheduling in smt processors. *In proceedings of the International CASES-2005*, pages 166–176, September 2005.
[12] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *In IEEE Transaction on Computers*, 2006.
[13] R. Jain, C.J. Hughes, and S.V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proc. of the 23th International Symposium on Real-Time Systems Symposium*, pages 134–145, Dec 2002.
[14] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, Aug 2003.
[15] P.M.W. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero. Branch classification for SMT fetch gating. *Proc. of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, pages 49–56, 2002.
[16] Markus Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, Nov 2003.
[17] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *Proc. of the ISPASS*, November 2001.
[18] D. T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.
[19] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996.
[20] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.
[21] A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *ACM SIGMETRICS*, June 2002.
[22] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. *Proc. of the 34th MICRO*, December 2001.
[23] 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. of the 23th Annual Intl. Symposium on Computer Architecture*, pages 191–202, April 1996.
[24] D.M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proc. of the 22th Annual ISCA*, 1995.
[25] D. W. Wall. Limits of instruction-level parallelism. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991.
[26] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proc. of the 1st Intl. Conf. on High Performance Computer Architecture*, pages 49–58, June 1995.