21st International Symposium on Computer Architecture and High Performance Computing

Thread to Core Assignment in SMT On-Chip Multiprocessors

Carmelo Acosta †, Francisco J. Cazorla ⋆, Alex Ramirez †⋆, Mateo Valero †⋆

† Universitat Politecnica de Catalunya
HiPEAC European Network of Excellence
Barcelona, Spain
{cacosta,aramirez,mateo}@ac.upc.edu

⋆ Barcelona Supercomputing Center
Barcelona, Spain
{francisco.cazorla,alex.ramirez,mateo.valero}@bsc.es

Abstract

State-of-the-art high-performance processors like the IBM POWER5 and Intel i7 show a trend in industry towards on-chip Multiprocessors (CMP) *involving* Simultaneous Multithreading (SMT) *in each core. In these processors, the way in which applications are assigned to cores plays a key role in the performance of each application and the overall system performance.*

In this paper we show that the system throughput highly depends on the Thread to Core Assignment (TCA)*, regardless the* SMT Instruction Fetch (IFetch) Policy *implemented in the cores. Our results indicate that a good* TCA *can improve the results of any underlying* IFetch Policy*, yielding speedups of up to 28%.*

Given the relevance of TCA*, we propose an algorithm to manage it in CMP+SMT processors. The proposed throughput-oriented* TCA Algorithm *takes into account the* workload characteristics *and the underlying* SMT IFetch Policy*. Our results show that the* TCA Algorithm *obtains thread-to-core assignments 3% close to the optimal assignation for each case, yielding system throughput improvements up to 21%.*

1 Introduction

The performance achievable by traditional superscalar processor designs does not scale with the increasing transistor count, due to the limitations imposed by *Instruction Level Parallelsim (ILP)*. As a consecuence, *Thread Level Parallelism (TLP)* has become a common strategy for improving processor performance. Since it is difficult to extract more *ILP* from a single program, multithreaded processors rely on using the additional transistors to simultaneously execute several programs. This strategy has led to a wide range of multithreaded processor architectures like *SMT* [15, 16, 18], *CMP* [9], or combinations of both. Currently, many of the main processor vendors have some multithreaded processors. Some examples are the Intel Core 2 Duo [17], a dual core processor, the Intel Core i7 [1], quad core processor comprised of 2-context SMT cores, and the IBM POWER5 [12] and POWER6 [8], dual core processors comprised of 2-context SMT cores.

In an SMT processor, the *scheduling process* is comprised of two main steps. First, in the *co-schedule selection* [6] step the *Operating System (OS) Job Scheduler* selects the workload to be executed from the ready task list. Next, in the *resource sharing* [6] step the *SMT* processor's resource allocator decides how to prioritize threads; carried out by the *IFetch Policy* [3, 5, 14, 16].

In a *CMP* processor comprised of *SMT* cores (CMP+SMT) the traditional *SMT scheduling process* requires an additional intermediate step. Since the SMT hardware contexts are distributed in different SMT cores, some applications must be co-scheduled to the same core. We call this additional scheduling step the *Thread to Core Assignment (TCA)*. In this paper we analyze the TCA, focusing on the relation between the *TCA* and *resource sharing* scheduling steps. Our results indicate that a bad *TCA* may negate the performance advantage of a *robust* (i.e., that appropriately handles long-latency loads) *IFetch Policy*, like the *FLUSH* [14] and *STALL* [14] policies.

The importance of the *TCA* lies in the fact that performance degradation can be prevented by appropriately assigning the threads to co-schedule in each *SMT* core. An illustrative example is shown in Figure 1. It depicts the throughput of a 2-core processor with 2 hardware contexts per core, using the *ICOUNT* and *FLUSH* policies (See Section 2 for details). The applications in the workload (*A,B,C,D*) are assigned to the *SMT* cores (e.g., [A,B] = *A* and *B* assigned to the same core). Notice in Figure 1 that while the first *TCA* yields similar results for both policies the second *TCA* obtains an improvement of 19%.

As the amount of SMT cores increases, the number of possible TCAs exponentially grows. Assuming 2 hardware contexts per SMT core, there are 3, 105, and millions of different TCAs for 2, 4, and 8-core implementations, respectively. Therefere, the selection of a good TCA should not involve an excessive overhead, proportional to the num-

1550-6533/09 $26.00 © 2009 IEEE
DOI 10.1109/SBAC-PAD.2009.13

IEEE computer society

TCA	ICOUNT	FLUSH
1) [A,B] [C,D]	1,57	1,57
2) [A,C] [B,D]	1,63	1,87

+0%
+19%

Workload: A,B,C,D
A = vpr
B = equake
C = fma3d
D = vortex

Figure 1. TCA Example.

ber of cores. In this paper we propose a *TCA Algorithm* that selects an appropriate TCA for each case, according to the workload characteristics and the underlying IFetch Policy, without adding excessive overhead. Our results show that the proposed *TCA Algorithm* obtains assignments 3% close to the optimal for each case, yielding system throughput improvements up to 21%.

2 Methodology

We use a trace driven *SMT* simulator derived from *SMTsim* [15]. The simulator consists of our own trace driven front-end and an improved version of the *SMTsim* back-end that provides multicore support. Our simulator also permits simulating the impact of executing along wrong paths on the branch predictor and the instruction cache by having a separate basic block dictionary in which information of all static instructions is contained.

Our workloads use the *SPEC2000 benchmark suite*. From them, we have collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [10]. Whenever a benchmark is used more than once in a workload each additional instance is forwarded 1 million instruccions more than the prior one. Each program is compiled with the *–O2 –non_shared* options using DEC Alpha AXP-21264 C/C++ compiler and executed using the reference input set. Since a complete study of all benchmarks is not feasible due to excessive computational cost we have randomly chosen some of them. Table 1 shows the main simulation parameters and the chosed workloads. The workload size is denoted by the prefix *xW*, where *x* stands for the number of benchmarks involved. We simulate CMP+SMT configurations with shared multibanked L2 Cache and two hardware contexts per core. Additionally, in order to assure a minimal cache share 16-thread and 32-thread workloads (16W & 32W) are simulated using a shared 6MB 6-banked L2 Cache (instead of 4MB 4-banked).

We simulate each workload employing 4 different IFetch Policies: *Round Robin (RR)* [16], *ICOUNT* [16], *STALL* [14] and *FLUSH* [14]. In all cases, simulations are executed for a fixed interval of 140 millions of simulation cycles. In our simulations we assume this simulation interval as a single OS quantum of execution.

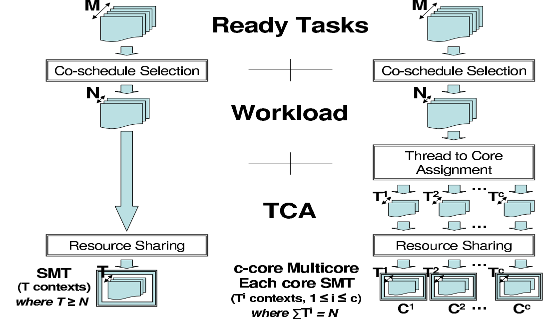


Figure 2. Scheduling Layers in SMTs and Multicored SMTs.

The SMT fairness concept can not be directly appliyied to the proposed *TCA Algorithm*, since it does not add any hardware/software mechanism to stop/interrupt the application's execution; it just assign threads to *SMT* cores. This is the main reason to not include in this work additional metrics like the *Harmonic Mean* or *Weighted Speedup*.

3 Scheduling in Multicore Simultaneous Multithreaded Processors

In an SMT processor the scheduling of a set of tasks requires decisions at two levels, as shown on the left side of Figure 2. First, when the number of available ready tasks *M* is larger than the *T* hardware contexts supported by the SMT processor, we need to determine which tasks to schedule together. The OS Job Scheduler seletcs a set of *N* tasks (where $N \leq T$) from the *M* ready tasks: the workload. This first scheduling layer is known as *co-schedule selection* [6].

Second, we need to perform the resource distribution among co-scheduled tasks in an SMT processor. The OS passes the workload to the hardware, which must decide how to partition the SMT processor resources among all workload's applications, in order to avoid resource monopolization by any running thread. This second layer is known as *resource sharing* [6], as shown on left side of Figure 2. There are several proposals in the literature [3, 5, 14, 16] to manage the resource sharing. In this work we focus on four IFetch Policies: *RR* [16], *ICOUNT* [16], *STALL* [14] and *FLUSH* [14]. Far from representing the state-of-the-art of *IFetch Policies*, we use them as an easy-to-explain example.

In a CMP+SMT processor, the *T* hardware contexts are distributed among all SMT cores, as shown on the right side of Figure 2. Each execution core works as a different SMT processor with its own resource allocation scheme. So, the *N* workload applications are distributed among the *c* SMT cores. Since the multicore processor's resources are statically distributed among all SMT cores, the way in which

Simulation Parameters				Benchmarks					
Pipeline depth	11 stages	L1 I-Cache	64KB, 4-way, 8 banks	gzip	a	vortex	j	mesa	s
Queues Entries	64 int, 64 fp, 64 ld/st	L1 D-Cache	32KB, 4-way, 8 banks	vpr	b	bzip2	k	fma3d	t
Execution Units	4 int, 3 fp, 2 ld/st	L1 lat./miss	3/22 cycs.	gcc	c	twolf	l	sixtrack	u
Physical Registers	320 regs.	I-TLB ,D-TLB	512 ent. Full-assoc.	mcf	d	art	m	facerec	v
ROB Size*	256 entries	TLB miss	300 cycs.	crafty	e	swim	n	applu	w
Branch Predictor	perceptron	L2 Cache	4MB, 12-way, 4 banks	perlbmk	f	apsi	o	galgel	x
	(4K local, 256 pers)	L2 latency	15 cycs.	parser	g	wupwise	p	ammp	y
BTB	256 entries,	M. Memory lat.	250 cycs.	eon	h	equake	q	mgrid	z
	4-way associative			gap	i	lucas	r		
RAS*	100 entries								

Type	Workload	Type	Workload	Type	
4W1	b, q, t, j	8W1	d, l, b, g, h, j, a, f	16W1	d, l, b, g, m, n, r, q, i, j, c, f, k, e, a, h
4W2	l, n, o, e	8W2	b, g, m, n, a, h, w, p	16W2	l, l+1, l+2, l+3, g, g+1, g+2, g+3, k, e, a, h, o, p, s, t
4W3	r, i, f, p	8W3	m, n, r, q, f, j, e, h	16W3	b, n, b+1, n+1, b+2, n+2, b+3, n+3, o, p, s, t, w, u, x, z
32W1	d, l, b, g, m, n, r, q, m+1, m+2, b+1, b+2, q+1, q+2, g+1, g+2, i, j, c, f, k, e, a, h, p, s, w, o, h+1, j+1, a+1, f+1				
32W2	l, l+1, b, b+1, m, m+1, n, n+1, g, g+1, g+2, b+2, q, q+1, q+2, r, j, j+1, j+2, h, h+1, a, a+1, f, u, p, p+1, p+2, c, c+1, s, s+1				
32W3	d, b, b+1, b+2, n, n+1, n+2, q, m, m+1, m+2, m+3, l, l+1, l+2, l+3, u, h, h+1, h+2, h+3, j, j+1, f, a, a+1, a+2, p, p+1, w, w+1, f+1				

Table 1. Simulation parameters and Workloads. (resources marked with * are replicated per thread)

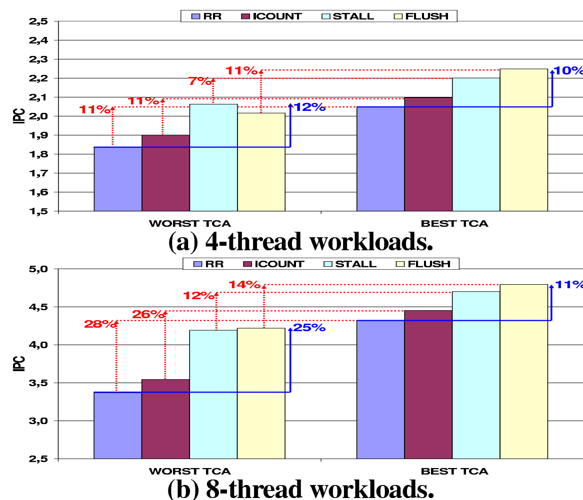
we schedule together tasks determines the performance of the underlying resource sharing policy in each core. Obviously, the more the tasks (N) in the workload the more possible schedulings, or TCAs, growing exponentially with the number of tasks. The three layers of the task scheduling in multicore processors comprised of SMT cores are shown on the right side of Figure 2.

In an state-of-the-art general-purpose OS like the *Linux 2.6* [2] the *TCA* does not have a significant role in the scheduling algorithm. *Linux 2.6* considers each hardware context as a different logical domain, hierarchically organized according to the hardware context distribution on the chip. The OS keeps a different queue of runnable applications, sorted by process priority. In order to keep balanced these queues a load balancing process may be triggered, implying thread migrations from one core to another. Besides the process priority, the decision whether a job has to be scheduled in a given core basically depends on the fact whether that job was recently executed in that core, to take profit of the data that could remain in the cache. However, the load balancing process performed by the *OS* may involve thread migrations between execution cores, losing the data in the private caches.

4 Thread to Core Assignment and the IFetch Policy

In order to analyze the relation between the TCA and the underlying IFetch Policy we simulated all 4 (*4W*) and 8-thread (*8W*) workloads in Table 1 on 2 and 4-core CMP+SMT processors, respectively. Figure 3 break downs the average results for each IFetch Policy into *WORST* and *BEST TCA* (i.e., WORST TCA corresponds to the TCA that yields the worst throughput among all possible TCAs).

Figure 3 shows some interesting values on top of the graph itself. On the one hand, the percentages on top of each bar indicate the throughput improvement achievable for the



(a) 4-thread workloads.

(b) 8-thread workloads.

Figure 3. TCA Sensitivity.

corresponding IFetch Policy using the BEST TCA, with respect to the WORST TCA. That is, the relative importance of the TCA or *TCA Sensitivity*. On the other hand, the percentages on the right side of each group of bars indicate the relative importance of the IFetch Policy when using similar TCAs. That is, the throughput obtained using the TCA that yields the best/worst results for each workload and IFetch Policy. Comparing both results it is straightforward that the TCA has similar or even more importance in CMP+SMT processors than the IFetch Policy in SMT processors. Four main conclusions can be inferred from the average results shown in Figure 3:

1. *A good IFetch Policy reduces the negative effect of an inappropriate TCA*. That is, when counterproductive threads are assigned to the same SMT core the goodness of the IFetch Policy is critical to obtain high system throughput. So, in 8-thread workloads (Figure 3(b)) the TCA's relative importance ranges from *28%* to *12%* for RR and STALL, respectively.

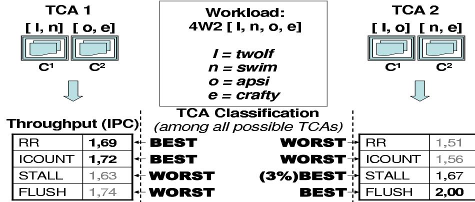


Figure 4. Example with different TCAs for a 4-thread workload.

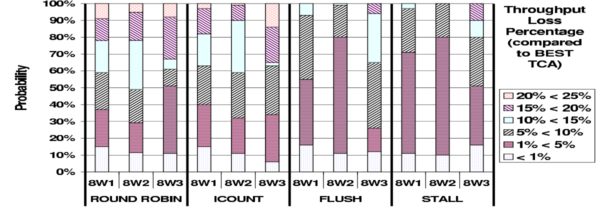


Figure 5. Probability of Throughput Loss in 8-thread workloads using Random TCAs.

2. *An appropriate TCA improves the results obtained regarless the underlying IFetch Policy*. The results in Figure 3 show that a good TCA improves the system throughput by more than 10% even using STALL or FLUSH policies.

3. *An inappropriate TCA could negate the performance advantange of a better IFetch Policy*. That is, the TCA should not be neglected even when implementing robust IFetch Policies. Notice in Figure 3 that the BEST TCA results using RR surpass those obtained with the WORST TCA using FLUSH.

4. *There is not a single TCA good for all cases*. As a matter of example Figure 4 shows the results yielded by workload *4W2* using two different TCAs. While *TCA 1* yields the BEST TCA results for RR and ICOUNT policies, and the WORST TCA results for STALL and FLUSH, *TCA 2* yields just the opposite results.

Finding the BEST TCA for each case is not a trivial task; the number of possible TCAs exponentially grows with the number of SMT cores. As a matter of example, there are 105 different TCAs for 8-thread workloads using a 4-core *CMP+SMT* processor. Since state-of-the-art OS like the *Linux Kernel 2.6* does not explicitly take into account TCA, a *RANDOM TCA* is assumed. Figure 5 shows the probability of randomly obtaining each TCA and the corresponding throughput loss. Notice that the probability of randomly obtaining the BEST TCA (loss lower than 1%) is in average close to 10%. The remainder TCAs highly depend on the IFetch Policy and the specific characteristics of each workload. Thus, while randomly selecting a TCA for workload *8W2* incurs in more than 5% of throughput loss with a probability of 71%, using *RR*, this probability drops to 20% using *FLUSH*. However, the same claim can not be stated for workload *8W3*, where this probability is close to 50% and 75% for *RR* and *FLUSH* policies, respectively. Obviously, the specific characteristics of workload *8W3* turn it into a more difficult target for *FLUSH* policy. Consequently, it is important to have a mechanism that assures some amount of reliability in terms of TCA selection.

5 Thread to Core Assignment Algorithm

In the following subsections we describe in detail the proposed *TCA Algorithm*, so as the *TCA Calibration* mechanism, aimed at handling the TCA scheduling layer in CMP+SMT processors. A complete evaluation is also included, with results revealing up to 21% of improvement over current TCA scheme.

5.1 TCA Algorithm Foundations

In order to properly manage TCA in CMP+SMTs we take into account both the workload characteristics and the SMT IFetch Policy. The proposed *TCA Algorithm* is designed for homogeneous implementations, with the same IFetch Policy implemented in all 2-hardware-context cores. Further implementations are left for future work.

From a deep analysis of the applications' behavior we have determined that the main reasons for an application to negatively interfere with another application assigned to the same core are the *memory behavior* and the *ILP*. On the one hand, applications with bad memory behavior, or memory-bounded (MEM), tend to monopolize too many resources, specially instruction queue entries and rename registers, without a proportional increment to the overall system performance. On the other hand, cpu-bounded (CPU) applications tend to greedily monopolize the functional units, due to the high parallelism they exhibit. As a single indicator of both characteristics we use the number of committed instructions per cycle (IPC) during a similar portion of execution. So, we assume that CPU and MEM applications yield high and low IPC values, respectively.

Once characterized the applications we analyze the response of each SMT IFetch Policy to each application type. According to this response we can classify SMT policies into *naive* (RR and ICOUNT) and *robust* (STALL and FLUSH) whether they poorly or properly handle long latency instructions, respectively. According to this classification, we could state that in general CPU and MEM applications may only be assigned to the same SMT core in case of robust policy.

Robust policies must detect long latency instructions. Reacting too early or too late may negatively affect the

Algorithm 5.1: TCA(IPC)

1- Arrange threads by IPC.

2- Split sorted thread list into two sublists: HIGH-list *and* LOW-list *for upper and lower halves, respectively.*

3- For $i=0$ *to* $\frac{\textit{Number of Threads}}{8}$ *do*

- *3.1- Assign the last two threads on the* LOW-list *to one empty core.*
- *3.2- Assign the threads on the top and the tail of the* HIGH-list *to one empty core.*
- *3.3- Remove the assigned threads from the lists.*

4- While (Not Empty HIGH-list *and* LOW-list*) do*

- *4.1- Assign the thread on the* HIGH-list *top and* LOW-list *tail to one empty core.*
- *4.2- Remove assigned threads from the lists.*

Figure 6. TCA Algorithm implementation for FLUSH/STALL policy.

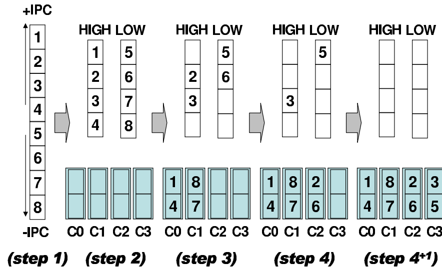


Figure 7. TCA Algorithm FLUSH/STALL Example (8 Threads).

system throughput [14]. Consequently, applications with a high rate of memory misses may impose a severe obstacle to a co-scheduled high performing application. Hence, isolating the threads with the worst memory behavior avoids such possible performance degradations. We assume that the longer the workload the more possible harmful MEM threads present in it. Therefore, we isolate MEM threads according to the workload size.

5.2 TCA Algorithm

The proposed *TCA Algorithm* implementation for robust IFetch Polices (i.e., *FLUSH* and *STALL*) is presented in Figure 6. The *TCA Algorithm's* foundations explained above, can be easily identified in the implementation shown in Figure 6. First, in *steps 1* and *2*, the workload applications are classified according to their memory behavior and ILP. In general, all applications going through a program phase with good memory behavior lay on the HIGH-list; LOW-list otherwise. Within each sublist, applications with high ILP lay on the upper possitions.

The third step in Figure 6 refers to the last paragraph in Section 5.1. Based on empirical results we assume that from each eight threads in the workload we should isolate the one with the lowest IPC as potentially harmful MEM. That thread is then isolated in an SMT core with the following thread with the lowest IPC; the less potentially affected by resource contention. To keep balanced both lists, we isolate a pair of HIGH-list threads for each pair of LOW-list threads isolated. According to the ILP reasoning in Section 5.1, we assign together threads with different degrees of ILP. That is, threads on the top list positions are assigned with those on the bottom list positions.

In the *fourth step* the remainder threads are assigned according to the SMT fetch policy. In case of robust policies (i.e., STALL/FLUSH) a thread from the HIGH-list is paired with a thread from the LOW-list. According to ILP reasoning, top-list threads are paired with bottom-list threads. Figure 7 shows a TCA example performed by the *TCA Algorithm* in a 4-core implementation using the FLUSH policy in each SMT core.

The main difference between the *TCA Algorithm* implementation for robust policies, shown in Figure 6, and the corresponding implementation for naive policies (i.e., RR/ICOUNT) lies in the fourth step. Due to their bad response to bad memory behavior, in case of naive policies threads are paired with threads from the same thread list. Additionaly, the bad response to MEM applications also motivates that the third step in Figure 6 is performed just once in case of naive policies.

Notice that the *TCA Algorithm* does not hamper the execution of any running thread. It does not stop threads but determines which threads should be assigned together to the same SMT core.

5.3 TCA Calibration

The proposed *TCA Algorithm* requires an IPC value for each application. However, as the execution flows applications go through different program phases [4, 11]. The behavior of an application may significantly change from one program phase to another. Consequently, it is required an IPC prediction mechanism that periodically supply the TCA Algorithm with updated IPC values.

To assist the TCA Algorithm, we have developed an IPC prediction mechanism: the *TCA Calibration*. On every context switch an initial *TCA Calibration Phase* is triggered. As shown in Figure 8, the TCA Calibration simply consists of executing in pure-CMP mode (i.e., one application per core in single thread (ST) mode) for a short and fixed amount of time. We assume negligible interference in the shared L2 cache. Since the processor is comprised of 2-hardware-context cores, two evaluation intervals are required, in the worst case, to fully test the whole workload.

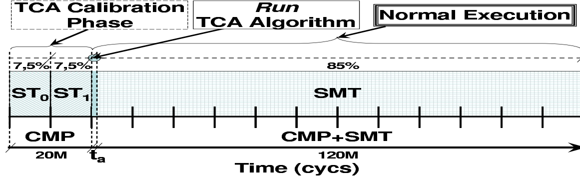


Figure 8. TCA Calibration.

The IPC values obtained are then stored in a small structure from which are feed to the TCA Algorithm. Although the IPC predictions obtained might not be fully accurate they are valid for the TCA Algorithm as long as the relative order between applications would be representative.

In an state-of-the-art general-purpose OS like *Linux 2.6* [2], the lenght of the OS quantums of execution ranges from 0 to 3200 millions of cycles, with typical values lying in the 40 to 500 millions of cycles interval. Based on temporal locality, we assume that each application would keep a similar behavior for the following 140 millions of cycles. This means that every 140 millions of cycles an additional TCA Calibration Phase must be triggered in order to keep track of the varying applications' behavior. However, in case of having IPC values with less than 140 millions of cycles old the TCA Calibration Phase is not required. Obviously, the shorter these intervals the lesser the negative effects of using the single thread mode. After some experiments, we adjusted the size of these intervals to 10 millions of cycles. Adding these single-threaded intervals (ST_0 and ST_1) to the TCA Algorithm overhead itself (denoted as t_a in Figure 8) the maximum overhead is 15+t_a%. Due to the *simplicity* of the *TCA Algorithm* we assume negligible the contribution of t_a to the final overhead.

5.4 TCA Algorithm Evaluation

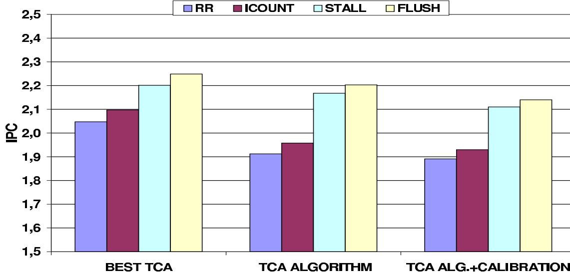
In order to evaluate the performance of the proposed TCA Algorithm and TCA Calibration mechanisms we applied them to all 4 (*4W*) and 8-thread (*8W*) workloads in Table 1, simulated in 2 and 4-core *CMP+SMTs* respectively. The average results are shown in Figure 9. The BEST TCA results shown on the left side of Figure 9((a) and (b)) are obtained simulating all different TCAs (i.e., 3 and 105 for 2 and 4-core implementations, respectively) for each workload and IFetch Policy, selecting the ones which yield the highest throughput. Since short OS quantums may not require additional TCA Calibration, Figure 9 shows two groups of results using the TCA Algorithm. The results shown in the middle of Figure 9((a) and (b)) are obtained supplying the TCA Algorithm with IPC values obtained from a prior off-line single-threaded 300M-cycle execution of each application on the same execution core; that is, without requiring from any IPC prediction mechanism. The results shown on the right side of Figure 9((a) and (b)) involve the TCA Calibration mechanism to supply the TCA Algorithm with the IPC predictions, as explained in Section 5.3.

Figure 9 shows that the TCA Algorithm yields results very close to the optimal for each case, a *3%* in average. Since the TCA is not explicitly taken into account by current OS in CMP+SMT processors, the state-of-the-art TCA policy is represented by a RANDOM TCA. From the results in Figure 5 it can be inferred that the probability for a RANDOM TCA to achieve similar results to the ones yielded by the TCA Algorithm are only of *26%*, using the RR policy. Using a robust IFetch Policy (e.g., FLUSH), this probability is increased to *33%*. As a matter of example, the TCA Algorithm yields a speedup of *21%* in *8W1* using the RR policy, with respect to the WORST TCA.

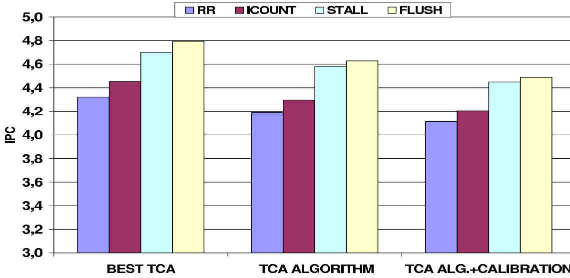
Using the TCA Calibration mechanism slightly reduces the speedup yielded by the TCA Algorithm. As shown on the right side of Figure 9, the single-threaded portion of the execution, required by the TCA Calibration mechanism, slightly reduces the final throughput. The results in this case are in average *5%* close to the optimal for each case. In this case, the probability for a RANDOM TCA to achieve similar results to that obtained using the TCA Algorithm raise to *41%*, using the RR policy. Using FLUSH this probability is increased to *58%*.

As mentioned in Section 5.3, not all context switches would require from the TCA Calibration Phase. Only those threads that have executed for more than 140M cycles since the last calibration would require from a TCA Calibration, and the corresponding use of single-thread mode. This fact would reduce the overall use of the single-thread mode, and therefore reduce the final throughput overhead. Nevertheless, considering the minimal overhead involved, the TCA Algorithm supported by the TCA Calibration mechanism offers a quite interesting complexity-effective improvement.

In order to evaluate the scalability of the proposed TCA Algorithm in forthcoming microprocessor generations, we simulated all the 16 and 32-thread workloads in Table 1, using 8 and 16-core CMP+SMT implementations, respectively. Due to exponentional computational costs, we do not directly compare the TCA Algorithm results for 16 and 32-thread workloads with the BEST TCA, as done for 4 and 8-thread workloads. Instead, we randomly selected a group of 100 TCAs for each workload and IFetch Policy. From them, we selected the TCA which yields the highest throughput and called it *BEST of 100*. As done in Figure 9, Figure 10 shows the average results obtained using the TCA Algorithm. The results on the middle group of Figure 10 shows figures very close to the optimal for each case, a 3% in average. Therefore, from Figure 10 it can be inferred that the TCA Algorithm scales to future 8 and 16-core implementations. As shown on the right side of Figure 10, the

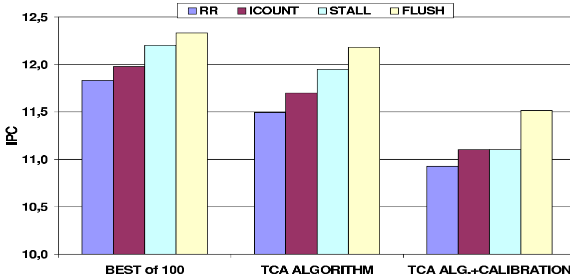


(a) 4-thread workloads.

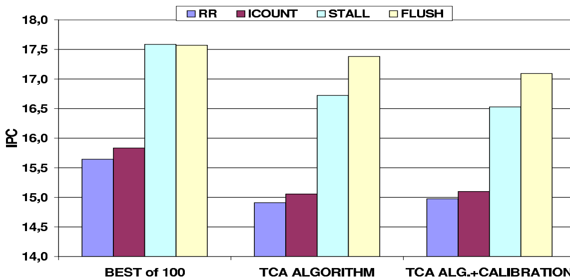


(b) 8-thread workloads.

Figure 9. TCA Algorithm results.



(a) 16-thread workloads.



(b) 32-thread workloads.

Figure 10. TCA Algorithm results.

effect of the TCA Calibration mechanism on the TCA Algorithm's results is similar to that of 2 and 4-core CMP+SMT implementations.

6 Related Work

To the best of our knowledge, this is the first work which explicitly identifies the need of an intermediate layer, that we call TCA, in the scheduling process for CMP+SMT processors. We include a complete analysis, which motivates the need of this new TCA layer, studying the scheduling process in the emerging CMP+SMT processors. We also propose a TCA Algorithm that handles this intermediate scheduling step with quite accurate and reliable results.

In [6] Jain et al. it is explored for the first time the soft realtime scheduling on an SMT processor, focusing on the co-schedule selection. They propose new co-scheduling variations that consider resource sharing and try to utilize SMT more effectively by exploiting application symbiosis. In this work we extend this exploration to a new scenario: CMP+SMT. In this scenario, we identify the need of a new step in the scheduling process: the TCA, and propose a complexity-effective solution: the TCA Algorithm. Similarly to what happens with the co-scheduling selection in SMT processors, the TCA is directly related with the next step of the scheduling process, the resource sharing.

In [13] and [19] several schedulers and heuristics are proposed to manage the co-schedule selection and increase system throughput in SMT processors. We focus on the next step of the scheduling process for CMP+SMT processors. Once the OS has selected the workload to be executed in the next OS quantum each application in the workload must be assigned to one of the SMT cores. The goodness of this assignment determines the overall system throughput. These proposals might work in conjuction with the TCA Algorithm, selecting easy-to-schedule applications for the TCA Algorithm in the underlying system. Nevertheless, more reseach is required to analyze the relation between the co-schedule selection and TCA scheduling layers (left for future work).

Kumar et al. propose in [7] some assignment policies to increase system throughput in Single-ISA Heterogeneous Multicore processors. They focus on obtaining the best match between heterogeneous cores and applications. We focus on a different scenario (i.e., homogeneous CMP+SMT) and the assignment is focused on obtaining the best match between co-scheduled applications in each SMT core.

7 Conclusions

The scheduling process in the emerging CMP+SMT processors differs from that of prior SMT and CMP processors, requiring a new and intermediate scheduling step, that we call *Thread to Core Assignment (TCA)*. In this paper we include a complete analysis that motivates the importance of this new scheduling step in the system throughput.

On the one hand, we show that a good TCA may yield up to 28% system throughput improvement. This paper also analyzes the relation between the TCA and the resource sharing, generally managed by the IFetch Policy implemented in hardware. We identify that robust (and complex) IFetch Policies may help reducing the negative effect of bad TCA selections. However, we also show that a bad TCA can negate the performance advantange of a robust IFetch Policy. As a consequence, better results can be obtained using a CMP+SMT processor implementing Round Robin policy, and the appropriate TCA, than a CMP+SMT implementing a better and complex IFetch Policy like FLUSH. Therefore, there must be a balance between the TCA selection policy and the IFetch Policy implemented.

The TCA which yields the best results depends on both the underlying IFetch Policy and the specific workload characteristics. Consequently, there is not a single TCA which yield the best results for all cases. Moreover, due to the TCA result distribution, it gets harder to obtain the optimal TCA as the workload size increases since the number of different TCAs exponentially grows with the workload size.

In order to manage the TCA scheduling step, we propose the TCA Algorithm, that generates close-to-the-optimal TCAs for each case, considering both the workload's characteristics and the underlying IFetch Policy implemented in the hardware. To do so, the TCA Algorithm just requires a representative IPC value of each application in the workload. To assist the TCA Algorithm, supplying these IPC values, we also propose an IPC prediction mechanism, that we call TCA Calibration. Our results show that the proposed TCA Algorithm obtains thread-to-core assignments 3% close to the optimal assignation for each case, yielding system throughput improvements up to 21%. Besides, its accuracy scales with the workload size and number of on-chip SMT cores.

Finally, we want to emphasize the simplicity of the proposed TCA Algorithm, a key aspect considered during its development. We do think the proposed TCA Algorithm's design is simple enough to allow a real implementation. Thus, each vendor would develop the corresponding TCA Algorithm implementation for each new processor and distribute it with its product, as currently done with the drivers. The TCA module could be then added to the OS, just requiring an additional Kernel recompilation or dynamic linkage.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN2007-60625, the Barcelona SuperComputing Center(BSC) and the HiPEAC European Network of Excellence.

References

[1] http://www.intel.com/products/processor/corei7/index.htm.

[2] D. P. Bovet and M. Cesati. Understanding the Linux Kernel - Third Edition.

[3] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of MICRO-37*, 2004.

[4] A. Dhodapkar and J. Smith. Comparing Program Phase Detection Techniques. In *Proc. of MICRO-36*, 2003.

[5] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. of HPCA-9*, 2003.

[6] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proc. of 23th Intl. Real-Time Systems Symposium*, 2002.

[7] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA-31*, 2004.

[8] H. Le, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[9] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proc. of ASPLOS-7*, 1996.

[10] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. of PACT-10*, 2001.

[11] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. of ISCA-30*, 2003.

[12] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[13] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, 2001.

[14] D. M. Tullsen and J. A. Brown. Handling Long-latency loads in a Simultaneous Multithreaded Processor. In *Proc. of MICRO-34*, 2001.

[15] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of ISCA-22*, 1995.

[16] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of ISCA-23*, 1996.

[17] O. Wechsler. Inside Intel Core Microarchitecture - Setting New Standards for Energy-efficient Performance . *White Paper*, 2006.

[18] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. of PACT*, 1995.

[19] O. Zaki, M. McCormick, and J. Ledlie. Adaptively Scheduling Processes on a Simultaneous Multithreading Processor.