Characterizing Thread Placement in the IBM POWER7 Processor

Stelios Manousopoulos*†, Miquel Moreto*‡, Roberto Gioiosa*, Nectarios Koziris†, and Francisco J. Cazorla*§

*Barcelona Supercomputing Center (BSC), Barcelona, Spain
†National Technical University (NTUA), Athens, Greece
‡International Computer Science Institute (ICSI), Berkeley, CA 94704, USA
§Spanish National Research Council (IIIA-CSIC), Barcelona, Spain

***Abstract*—There is a clear trend in current processor design towards the combination of several thread level parallelism paradigms on the same chip, exemplified by processors such as the IBM POWER7. In those processors, the way threads are assigned to different hardware contexts, denoted *thread placement*, plays a key role in improving overall performance.**

In this paper we analyze the thread placement problem in the IBM POWER7 processor. Under each thread placement setup we analyze in detail how hardware resources are shared among running threads. We show to which extent a software designer can characterize an application on the POWER7 and based on that characterization, select the best thread placement configuration to improve a target metric. Our results show that a 54% reduction in execution time can be obtained (11.2% on average) when running pairs of desktop parallel applications under the appropriate thread placement.

***Keywords*-Thread placement; IBM POWER7; Resource sharing; SMT;**

I. Introduction

Thread level parallelism (TLP) has become the most common strategy to improve processor performance. Current processors implement different TLP paradigms: threads executing in Simultaneous Multithreading (SMT) processors [11] [17] share most of the processor resources, while threads running on Chip-Multiprocessors (CMP) only share some levels of the cache and memory hierarchy [10]. The combination of several of those TLP paradigms offers several benefits: While SMT processors reduce fragmentation in on-chip resources, CMPs replicate multiple cores leading to improved energy efficiency. This has made CMP+SMT the predominant approach in current designs such as the IBM POWER7 [12] and the Intel i7 [8].

Even if all cores in a CMP+SMT processor are homogeneous, the combination of TLP paradigms leads to an heterogeneous sharing of hardware resources. In a CMP+SMT processor the interaction between threads varies depending on which level of resources they have in common: threads in the same core share many more resources, and hence interact much more than threads in different cores. This makes hardware resource sharing complex to analyze and control in order to optimize metrics like throughput or fairness.

In this paper, we present a performance characterization of the thread placement problem in the POWER7 processor [12], as a representative of current multi-TLP processors. The POWER7 is a CMP+SMT processor comprising 8 cores in which each core is 4-way SMT. Moreover, every core is divided into two clustered pipelines, with each one supporting two threads[1]. Threads in the same cluster share more resources than threads in different clusters, although threads in different clusters still share some on-core resources. Hence, resources are shared between threads in different cores, between all threads running in a core, between threads running in a cluster and between threads running in different clusters.

In such an architecture with 4 levels of resource sharing, thread placement plays a key role in performance: By properly placing threads, so that threads with heterogeneous resource profiles are placed together, per-thread and system performance can be improved. In this paper, with the help of specialized micro-benchmarks that stress particular parts of the architecture, we extract detailed knowledge about the internal behavior of resource sharing on the POWER7, which we use to derive the best thread placement for different workloads.

- When only one thread runs in a core, we corroborate that the highest performance is obtained if it runs in [AX][XX] placement (on average 41.1% better than the equally worst cases of [XX][AX] and [XX][XA]).
- When running two threads we conclude that [AB][XX] placement is the optimal as long as none of the tasks constantly executes low-latency or data-dependent operations. Otherwise [AX][XB] or [AX][BX] equally provide the best results. We further observe that POWER7 limits the instruction fetch rate of threads missing in the Last Level Cache (LLC). As a result, LLC-missing threads do not clog shared resources and do not harm the performance of other co-running threads. This is not the case for cache-bound and core-bound applications with low IPC, that significantly decrease the performance of the co-executing threads in the same core.
- When running four threads in a core, we have to bind threads with similar resource usage profiles to different execution clusters in order to minimize resource conflicts. In case we have 2 pairs of threads, assigning threads with similar resource sharing profiles to contexts that do not share resources ([AB][BA]) leads to 12% better performance than placing them in the same cluster ([AA][BB]). If all 4 threads are different, the same behavior is observed: By changing the thread placement

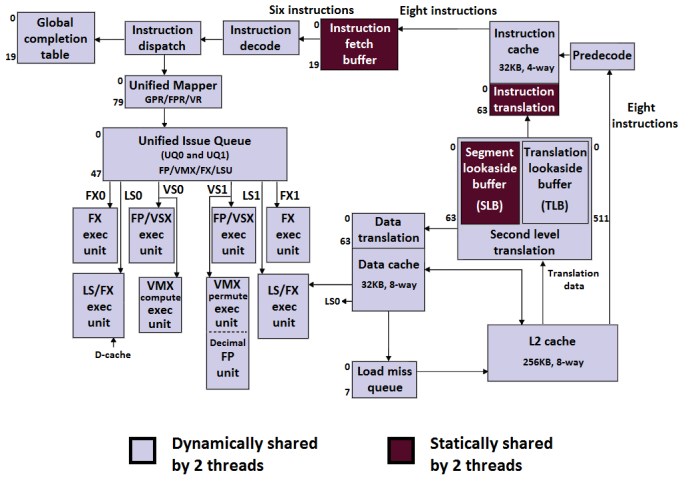
[1]In each core we denote hardware threads as Contexts 0, 1, 2 and 3. Contexts 0-1 and 2-3 are located into the same cluster. We use [AB][CX] notation where thread A executes on Context 0, B on Context 1, C on Context 2, while Context 3 is disabled.

978-1-4673-4532-3/12/$31.00 ©2012 IEEE

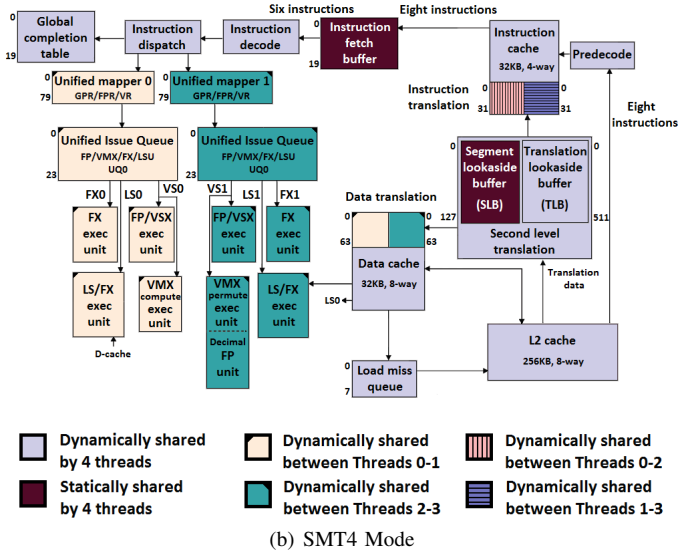
we get significant improvements of up to 66%.

We make use of the previous characterization to study the thread placement of parallel applications from the PARSEC benchmark suite [1]. We observe that the highest performance for a single parallel application is obtained under the maximum thread-count configuration (32 threads - 4 per core). However, with proper thread placement, execution with 16 threads reaches a similar performance (only 7.2% worse). The need to complete multiple tasks in the least possible time, motivates us to run pairs of PARSEC applications simultaneously with a suboptimal number of threads. We successfully take advantage of the different levels of shared resources to run pairs of PARSEC applications faster while sharing the cores. On the one hand, running parallel applications in isolation, one after another, is the best choice in only 13.3% of the studied pairs of benchmarks. On the other hand, when running them simultaneously, we can place threads from two applications either in the same or in separate cores. Sharing all cores with configuration [AB][BA] proves to be the best option with an 11.2% average reduction in execution time. This last result contradicts previous studies that advocate for running parallel applications in different cores to avoid cache thrashing and other hardware resource conflicts. In this scenario, it is very important to disable the hardware threads after an application finishes, otherwise fewer hardware resources are available for the other application and its performance drops considerably. In particular, an automatic thread-disabling mechanism used in the Linux kernel provides 2.4x speedups for some applications with respect to the case where hardware threads are not disabled.

The rest of this paper is structured as follows. Section II describes how resource sharing is done in POWER7, while Section III presents our experimental setup. Section IV presents the performance characterization results with micro-benchmarks, and Section V characterizes PARSEC applications. Section VI presents the related work and Section VII concludes this paper.



(a) SMT2 Mode



(b) SMT4 Mode

Figure 1. POWER7 shared resources in SMT2 and SMT4 modes. In ST mode, all resources belong to the same thread. These two figures are modifications of some figures in [12].

II. The POWER7 Processor

The IBM POWER7 is an 8-core CMP processor in which each core is 4-way SMT, counting 32 threads in total. Each core has its private data and instruction L1 caches, a private L2 cache and a local L3 region that can be shared between all cores.

A. Pipeline Description

At core level, POWER7 implements out-of-order logic with advanced branch prediction optimized for enhanced single-threaded performance. The main stages of an instruction are: instruction fetching and dispatching, register renaming, instruction issuing, execution and completion. As shown in Figure 1, up to 8 instructions per cycle are read from the I-cache and sent to the instruction buffers (IBUFs). Thread priority, pending cache misses, IBUF occupancy and thread balancing are used to determine which thread is selected for fetching in a given cycle [12].

Register renaming is done using mappers before instructions are placed in the issue queues. The general purpose register (GPR) and vector/scalar register (VSR) are merged into a unified register rename file, counting 80 entries in total. In the instruction issuing stage, POWER7 uses a 48-entry Unified (issue) Queue (UQ), implemented as two halves in order to achieve high frequency.

Instructions are executed in the functional units, which include a fixed-point unit (FXU), a load-store unit (LSU) and a vector-scalar unit (VSU). All three functional units are implemented as two separate pipelines. Both the floating-point unit (FPU), and the vector media extension (VMX) unit are incorporated in the VSU in order to save area and power. The LSU has the added capability of executing simple fixed-point operations. A global completion table (GCT) is used to track all in-flight instructions after dispatch. Instructions in the core are tracked as groups of instructions and, thus, will dispatch and complete as a group.

Table I
RESOURCE DISTRIBUTION UNDER DIFFERENT CORE MODES

Thread binding	Running threads	Core mode	Main available Resources				
			GPR ren.regs.	UQ	Execution Units	IBUF	I-ERAT/D-ERAT
[AX][XX]	1	ST	80	48	FX0/LS0/VS0 FX1/LS1/VS1	10	32 / 64
[XA][XX]	1	SMT2	80	48	FX0/LS0/VS0 FX1/LS1/VS1	10	32 / 64
[XX][AX]	1	SMT4	80	24	FX1/LS1/VS0,1	5	32 / 64
[XX][XA]	1	SMT4	80	24	FX1/LS1/VS0,1	5	32 / 64
[AB][XX]	2	SMT2	80 (shAB)	48 (shAB)	FX0/LS0/VS0 (shAB) FX1/LS1/VS1 (shAB)	10 each	32 each / 64(shAB)
[AX][BX]	2	SMT4	80 each	24 each	FX0/LS0/VS0,1 (A) FX1/LS1/VS0,1 (B)	5 each	32(shAB) / 64 each
[AX][XB]	2	SMT4	80 each	24 each	FX0/LS0/VS0,1 (A) FX1/LS1/VS0,1 (B)	5 each	32 each / 64 each
[AB][CX]	3	SMT4	80 (shAB) 80 (C)	24 (shAB) 24 (C)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (C)	5 each	32(shAC) - 32(B)/ 64(shAB) - 64(C)
[AB][XC]	3	SMT4	80 (shAB) 80 (C)	24 (shAB) 24 (C)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (C)	5 each	32(A) - 32(shBC)/ 64(shAB) - 64(C)
[AB][CD]	4	SMT4	80 (shAB) 80 (shCD)	24 (shAB) 24 (shCD)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (shCD)	5 each	32(shAC) - 32(shBD)/ 64(shAB) - 64(shCD)

B. Cache Hierarchy and Translation Logic

The POWER7 core has a dedicated 32-KB 4-way instruction cache and a 32-KB data cache. The first level of address translation is formed by an instruction effective-to-real-address translation (IERAT) and a D-ERAT. The I-ERAT consists of two 32-entry cache-like structures. The D-ERAT consists of two 64-entry cache-like structures. The second level of address translation is comprised of a Segment Lookaside Buffer (SLB) with 32 entries per thread and a Translation Lookaside Buffer (TLB) with 512 entries. Each core has a 256KB L2 cache and a local 4MB L3 region that can be shared between all cores forming a 32MB global L3 cache.

C. Core SMT Modes

A major characteristic of the POWER7 processor is that it is an architecture with two clustered execution pipelines. Some resources are duplicated in each cluster while other are shared between clusters. Depending on how running threads are bound to the four available contexts the processor operates in different modes: When a single thread runs and it is bound to Context 0, the processor executes in *Single-Thread (ST) mode*. When Contexts 2 and 3 are not active, and Context 1 is, the core runs in *SMT2 mode*. When Contexts 2 or 3 are active, the processor runs in *SMT4 mode*.

ST mode can only be used for single-thread execution as its name suggests. Also, mode SMT2 is designed for allocating all resources to 2 running threads. In contrast, SMT4 mode is designed for 4 thread execution, but has multiple uses. In some cases it can be used for running only 2 threads, while isolating them in different clusters. In other cases it is possible to execute only 1 thread, limiting it to just one cluster.

Even though most resources are shared between threads in the same cluster, some resources are shared between threads in different clusters and some others by all threads in the core. Table I and Figure 1 illustrate how the main hardware resources are shared under different core modes and for varying thread placements. In this table *sh* stands for shared. There are several microarchitectural resources shared between threads, but in this paper we focus on those mentioned in [12]:

1) Front-end: In ST and SMT2 modes, each thread uses a 16-entry link stack. In the SMT4 mode, each thread uses an 8-entry link stack. The IBUF holds up to 20 entries, each 4-instructions wide. In SMT4 mode, each thread can have 5 entries, whereas in ST and SMT2 modes, a thread has up to ten entries.

2) Execution pipelines: In ST and SMT2 modes, the two GPR files maintain identical contents, so there are a total of 80 rename registers; instructions can be issued to any of the two UQ halves and subsequently be executed in any of the functional units. In SMT4 mode the two GPR files have different contents, with threads in Contexts 0-1 executing in the upper pipeline (UQ0, FX0/LS0) and threads in Contexts 2-3 in the lower pipeline (UQ1, FX1/LS1). However, the majority of vector and scalar operations can be issued to any of the two UQ partitions and therefore be executed in any of the two VSU pipelines.

3) Address translation: In the instruction-fetch stage the IERAT table supports Contexts 0 and 2 in the first 32 entries, and Contexts 1 and 3 in the last 32 entries. The D-ERAT consists of two 64-entry caches. In ST and SMT2 modes the two halves have identical contents, but in SMT4 mode they have different contents with one half dynamically shared between Contexts 0-1 and the other between Contexts 2-3. The TLB is dynamically shared by all four threads

4) GCT: the 20 entries of the GCT that track groups of 8 instructions are dynamically shared between all four threads. The processor can complete one group per thread pair per cycle, with Contexts 0 and 2 forming one pair, and Contexts 1 and 3 forming the other one.

Overall, POWER7 features four levels of resource sharing. First, the *Intra-cluster* level, includes all the resources shared between Contexts 0-1 and Contexts 2-3, like the GPR files,

the UQ halves, and the functional units (FXU/LSU). Also, the *Inter-cluster* level, is formed by all resources that are shared between two threads from different clusters, e.g. the IERAT and the completion bandwidth. Additionally, the *Intra-core* level, features the selection of resources shared by all the threads within the core (both clusters), like the L1 data and instruction caches, the L2 cache, the local L3 cache, the GCT and the TLB. Finally, there is the *Inter-core* resource sharing level, which consists of the resources shared by all threads even in different cores, like the global L3 cache.

III. Experimental Setup

We use an IBM PS701 BladeCenter to run our experiments, which contains a single IBM POWER7 processor. The system runs SUSE Linux Enterprise 10 SP2 with kernel 3.0.9.

A. METbench Micro-benchmarks

We use a set of micro-benchmarks to perform an in-depth study of resource sharing on POWER7 under different thread placement setups. Micro-benchmarks are simple and repetitive tasks that stress specific processor parts. Each of the micro-benchmarks features a loop body with specific instructions depending on the behavior we want to achieve. For example some tasks continuously perform integer additions, while others continuously hit in a specific cache level. The loop body is repeated enough times so that the micro-benchmark runs for at least one second. The micro-benchmarks are organized into three categories based on the type of instructions they execute: integer, floating-point and memory.

- Integer micro-benchmarks include `cpu_int_add`, `cpu_int`, `cpu_int_mul` and `lng_chain_cpuint`. The loop body of `cpu_int` and `lng_chain_cpuint` can be seen in Table II. While `cpu_int` contains mixed integer instructions -specifically one multiplication every two additions- `cpu_int_add` only contains additions and `cpu_int_mul` only multiplications. `Lng_chain_cpuint`, to which we will refer as `lng_chain` for short, includes mixed integer instructions (also one multiplication every two additions) but is specifically designed to limit ILP with a long dependency chain of instructions.
- In the floating-point micro-benchmarks category we include `cpu_fp_asm` benchmark, which we will refer to as `cpu_fp` for short. It is written in POWER assembly to get more precision over its behavior and it is made of floating-point subtractions, multiplications and additions.
- Memory micro-benchmarks include all cache or memory related micro-benchmarks: `ldint_l1`, `ldint_l2`, `ldint_l3` and `ldint_mem`. The loop body structure of `ldint_X` micro-benchmarks can be seen in Table II, where the size of the array M varies to match the desired behavior. A pointer chasing technique is used to achieve the desired amount of loads on the relative memory hierarchy level. Specifically, an array is initialized with pointers, such that each element has the address of the next one, while the last element contains the address of the first entry of the array in order to execute the loop multiple times.

Table II
Source code of some micro-benchmarks

(a) `cpu_int`

```
for (it=0;
 it<M;
 it++) {
 LOOP_UNROL_20(
   a = a+a+it;
   b = b+b+it;
   c = c*c*it;
   d = d+d+it;
   e = e+e+it;
   f = f*f*it;
   g = g+g+it;
   h = h+h+it;
   i = i*i*it;
 )
}
```

(b) `lng_chain`

```
for (it=0;
 it<M;
 it++) {
 LOOP_UNROL_20(
   a = a+i;
   b = b+a;
   c = c*b;
   d = d+c;
   e = e+d;
   f = f*e;
   g = g+f;
   h = h+g;
   i = i*h;
 )
}
```

(c) `ldint_lX`

```
for (it=M;
 it>0;
 it=i-1) {
 LOOP_UNROL_20(
   p = *p;
 )
}
```

The METbench micro-benchmark suite integrates the FAME evaluation methodology [19]. This methodology ensures that every application in a multiprogrammed workload is fairly represented in the final results. FAME re-executes each application in the workload until its average accumulated IPC is similar to the IPC of that application when the workload reaches a steady state.

Validation of Micro-benchmarks behavior. We have used performance counters in single-thread mode to validate the behavior of METbench micro-benchmarks. `Cpu_fp` executes 99,7% of its instructions in the VSU. `Cpu_int`, `cpu_int_add`, `cpu_int_mul` and `lng_chain` use the FXU or the LSU in a percentage of at least 99,7%. The Load-Store pipelines in POWER7 can complete simple fixed-point operations, like adds and logical instructions [12], which explains why 27,8% of `cpu_int_add` and 16,8% of `cpu_int` instructions are executed in the LSU. Micro-benchmarks in the memory category mostly execute in the LSU with a percentage of at least 94%. The remaining percentage of instructions are branches and fixed-point operations, which are necessary for their functionality. We check that the memory micro-benchmarks fetch data from the correct level of memory hierarchy. Indeed `ldint_l1`, `ldint_l2` and `ldint_l3` fetch at least 99% of their data from L1, L2 and L3 local respectively. `Ldint_mem` fetches 87% of its data from memory and the rest from L1.

B. PARSEC Benchmark Suite

With the prevalence of CMP processors and the continuous trend to parallel application programming, comes the need for benchmark programs that are representative of the current and future real-world applications. Such a benchmark suite is PARSEC [1], which we use to corroborate the insights obtained with micro-benchmarks. It features state-of-the-art, computationally intensive algorithms and very diverse workloads from different areas of computing. Its programs use various parallelization approaches, including data- and task-parallelization. PARSEC is comprised of 13 benchmark programs, from which we use ten[2]: `blackscholes`, `bodytrack`, `dedup`,

[2]We encountered compilation or execution errors with `canneal`, `facesim`, `raytrace`

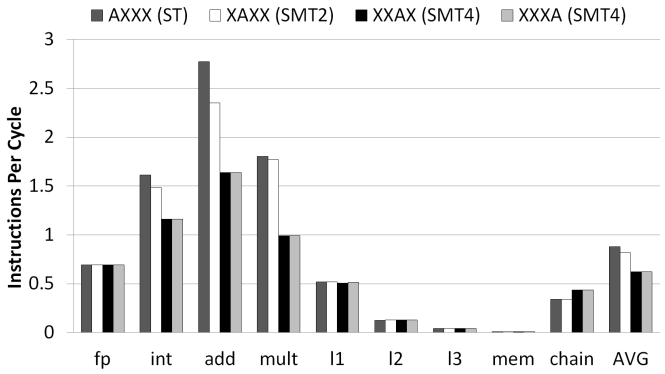


Figure 2. IPC of single-thread micro-benchmarks when they are bound to a given context, while the other contexts are disabled.

`ferret`, `fluidanimate`, `freqmine`, `streamcluster`, `swaptions`, `vips`, and `x264`. We make use of the Pthreads parallelization model for all these benchmarks, with the exception of `freqmine`, which is only available in OpenMP. Native input sets are used for all the experiments, as they are the most representative for real-world applications. The evaluation methodology for PARSEC consisted of repeating the experiments until the results variability was low. Most experiments converged with 3 repetitions, while a few required more runs to reach a steady state.

IV. Thread Placement Performance Characterization

In this section we analyze thread placement by means of micro-benchmarks and show several case studies in order to draw useful conclusions.

A. Thread Placement

Thread placement of a single-thread micro-benchmark. In previous multicore and multithreaded processors, when a task is executed in isolation, its performance is independent of the particular context to which it is bound. For example, for the IBM POWER5 and POWER6 that are 2-core, 2-thread processors, the performance of a task is the same regardless of which of the four contexts it runs in [2] [9]. This is not the case for the IBM POWER7. The binding of a single thread in one of the four core contexts can switch the core mode between ST, SMT2 and SMT4 and lead to different resource allocation. We expect that the more resources a thread has at its disposal the better it performs.

We observe in Figure 2 that when executing a single thread, binding it to different core contexts can have an important role in its performance. For some benchmarks the variation is small while for others it is significant (70% for the case of the `cpu_int_add` benchmark). On average the maximum performance is obtained when the task is bound to Context 0, with the second highest being when the task is bound to Context 1 and the lowest performance equally seen when the task is bound to Context 2 or 3.

When the benchmark is bound to Context 0 and the other hardware threads are disabled, the core runs in ST mode. Under this mode, the contents of the register files in each cluster are duplicated. This allows the benchmark to access both 24-entry partitions of the Unified Queue (UQ) and subsequently issue instructions to any of the functional units, as can be seen in Table I.

When a task is bound to Context 1, with Context 2 and 3 inactive, the core runs in SMT2 mode. Under this mode the resources that the task can use are in general equal to the resources it would have access in ST mode. However, we observe that some benchmarks obtain less performance in this mode than in ST mode. We further observe that the longer the latency of the executed instructions in a given task the lower the performance degradation that task suffers in SMT2 mode: 15%, 8% and 2% for `cpu_int_add`, `cpu_int` and `cpu_int_mul` benchmarks, respectively. From the description of the POWER7 [12] it was not clear to us why the pipeline of the POWER7 core causes this behavior. We assume that in ST mode the task in Context 0 benefits from some type of prioritization that the task does not have when it runs in Context 1 under mode SMT2.

When the task is bound to Context 2 or 3, the core runs in SMT4 mode. In this mode the GPR files have different contents and a thread can only access one UQ half (24 entries). Subsequently it can only issue instructions to one fixed-point pipeline (FX1), one load-store pipeline (LS1), but both vector-scalar pipelines (VS0 and VS1). Also, the thread in this mode can only use 5 Instruction Buffer entries, instead of 10 that it could have in the other two modes.

The `cpu_fp` benchmark does not show any degradation in performance between the different modes. This is because in all SMT modes most vector and scalar operations including floating-point can be dispatched to any of the two UQ halves and then executed on any of the two vector-scalar pipelines, as cited in [12]. Therefore it is not affected by the placement of the thread and consequently the core mode.

The `lng_chain` benchmark has a counter-intuitive behavior as its performance improves in SMT4 mode in comparison to ST and SMT2 modes. In SMT4 mode, each thread is confined to a UQ half and in that case data dependent operations can be issued back-to-back [12]. However, in ST and SMT2 modes, each thread is given access to both UQ partitions and so data-dependent operations can be issued to different partitions, suffering one cycle delay to bypass data from on cluster to the other. Table II shows that `lng_chain` is comprised of many inter-dependent instructions that greatly limit its ILP, thus dropping its performance to 3.7x slower than `cpu_int`.

All single-thread versions of cache and memory bound benchmarks including `ldint_ l1`, `ldint_l2`, `ldint_l3` and `ldint_mem` are not affected by the thread placement. Consecutive instructions of the `ldint_X` benchmark cannot be run in parallel since the pointer chasing technique is used, as explained in Section III-A. The performance of this category of micro-benchmarks is dominated by the latency of the relative level of memory hierarchy that they access.

To sum up, when a single thread runs in a POWER7 core, binding it to Context 0 provides the best results, unless it

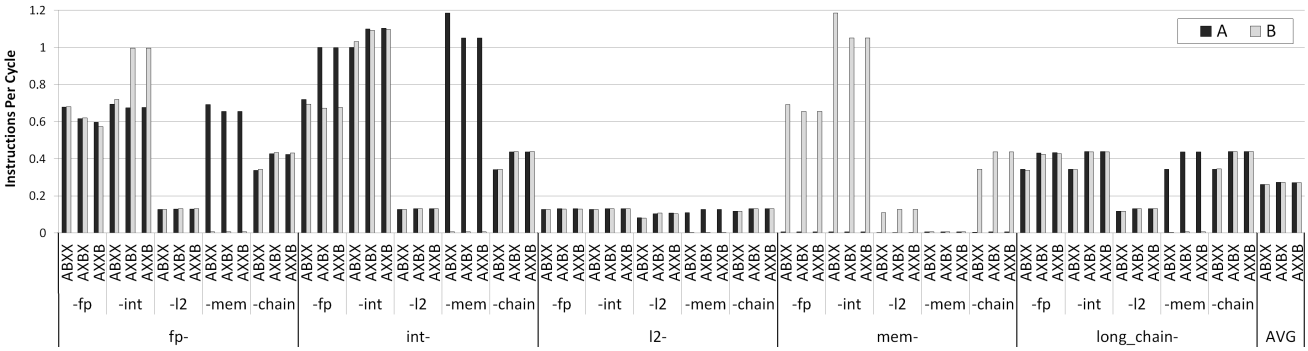


Figure 3. IPC of different pairs of single-thread micro-benchmarks when bound to two given contexts.

includes significant data dependencies, in which case Context 2 or 3 provides the best performance.

Thread placement of 2 single-thread micro-benchmarks. In this case we run two different threads under several thread assignments. Figure 3 shows the results of five representative micro-benchmarks, as well as the average values for the whole METbench micro-benchmarks suite. In each of the five sets of bars of the graph we show one of the five micro-benchmarks against all the other. For example if we take the bars for fp- in the graph and then focus on the -int subset of bars we get the results for the `cpu_fp-cpu_int` pair, where `cpu_fp` is A and `cpu_int` is B in [AB][XX] (SMT2), [AX][BX] (SMT4) and [AX][XB] (SMT4) thread placements.

We can clearly see the symmetry in performance between the various thread pairs, for example the pair `cpu_int-cpu_fp` results in the same performance for both micro-benchmarks, as that of `cpu_fp-cpu_int` in all three configurations. We conclude that *in a specific core mode* with 2 running threads the POWER7 core has symmetric behavior. In any case, configurations that change the core mode cannot possibly achieve similar performance, for instance [AB][XX] (SMT2 where all resources are shared by two threads) and [XX][AB] (SMT4 where two threads share resources in just one cluster).

We notice that the two SMT4 configurations have similar performance in all cases, which leads us to the conclusion that the micro-benchmarks do not cause significant contention in the IERAT or the GCT completion bandwidth, that are shared between hardware threads 0-2 and 1-3.

From Table I we observe that in SMT2 mode both threads *dynamically* share all core resources, while in SMT4 mode each of the 2 running threads is statically assigned half of most resources, like UQ entries and functional units. However, in SMT4 mode each pair has access to more rename registers. Whereas some pairs can benefit from dynamic resource allocation to more resources, others gain from the extra rename registers and would rather be confined in a cluster with half the core resources to themselves. Micro-benchmarks with low-latency instructions and a constant need for rename registers, such as `cpu_int`, suffer from the lack of rename registers in SMT2 and so perform better in SMT4 mode.

Moreover, there are some additional parameters to consider. We can clearly see how some low-IPC benchmarks, like `ldint_l2` and `lng_chain` limit the IPC of their co-runner. Counter-intuitively, the behavior of `ldint_mem` when co-executing with another task is entirely different to `ldint_l2`, even though its IPC is significantly lower. This is due to the existence of a hardware mechanism that automatically detects LLC (Last Level Cache) misses and reduces the rate at which instructions from the LLC-missing thread are fetched. As a result, the co-runners of `ldint_mem` manage to reach an IPC close to their peak while the same is barely affected since its performance depends primarily on the latency of memory access. This mechanism is enabled both in SMT4 and SMT2 mode, but in the latter the co-runners of `ldint_mem` have access to twice the resources and in that case they have more room for improvement when this prioritization is applied.

The only benchmark that performs better against `ldint_mem` in SMT4 rather than SMT2 is `lng_chain`. In fact `lng_chain` performs better in SMT4 mode against all other micro-benchmarks since as we have already seen in 1-thread placement it favors being limited to a UQ half.

To conclude, based on our characterization we can choose the optimal configuration between SMT2 and SMT4 modes. Two different threads perform better in SMT2 mode ([AB][XX]) than SMT4 mode ([AX][BX] or equally [AX][XB]), assuming none of them executes a high percentage of data-dependent or low-latency operations (e.g. integer adds).

Thread placement of 2 two-thread micro-benchmarks. In this experiment we co-execute 2 different pairs of threads of the same micro-benchmark in a POWER7 core. We expect the best performance in the placement that causes the least amount of resource conflicts between threads of the same type.

We have tried all possible pairs and in Figure 4 we show a representative selection of five of those pairs, in addition to the average values for all pairs. With a total of four active threads, the only possible core mode is SMT4. If we consider the core symmetry in a specific core mode, the possible placements are the following three: [AA][BB], [AB][AB] and [AB][BA]. We confirm that the symmetric placements [BB][AA], [AB][AB] and [BA][AB] do not show any difference in performance to the three previous configurations. In each of the five sections of the graph we show the IPC of each micro-benchmark against some of the other four as well as against itself, marked in the horizontal bar as 'AAAA'.

For the observed micro-benchmarks, we confirm that the placement [AB][BA] provides the best overall results (12% performance improvement w.r.t. the worst case), followed

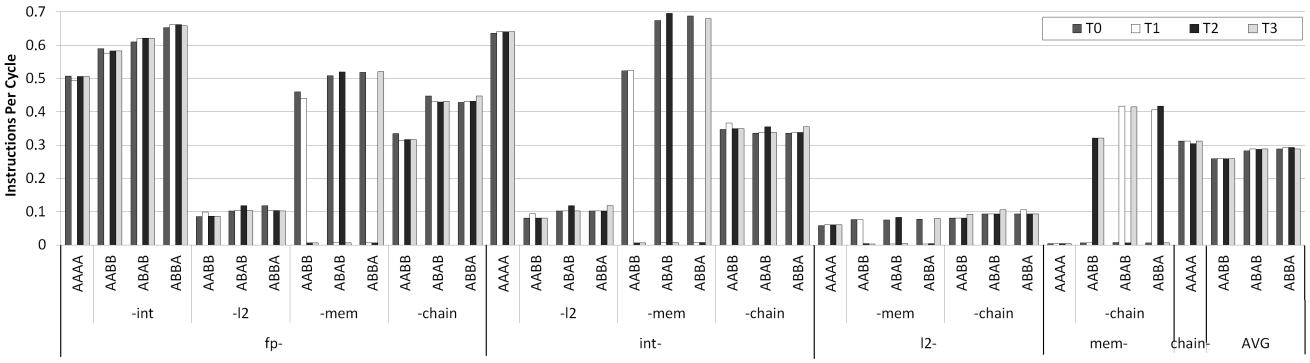


Figure 4. IPC of different pairs of two-thread micro-benchmarks when bound to four given contexts.

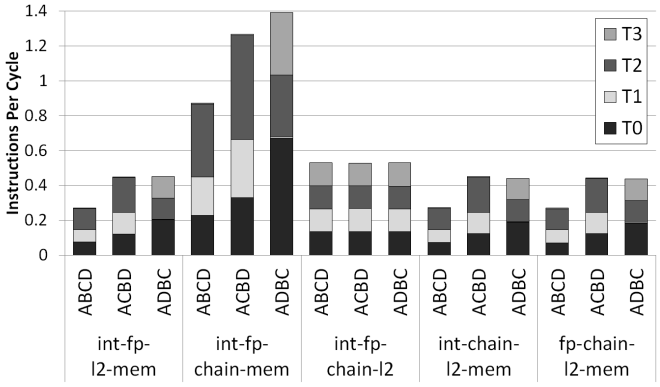


Figure 5. IPC of several combinations of 4 single-thread micro-benchmarks.

closely by [AB][AB]. The small differences between the two configurations could be explained by conflicts in the I-ERAT and the GCT completion bandwidth, which are shared between Contexts 0-2 and Contexts 1-3 and since in [AB][AB] configuration these contexts run threads of the same kind. Also, in the [AA][BB] placement -for which we notice the worst results for all three cases, by putting two threads of the same kind in each of the two clusters we are forcing them to contend for the same intra-cluster resources, including rename registers, UQ entries and functional units.

The instruction fetch unit in the POWER7 tries to balance instruction fetch rates between threads. According to our results, this happens even between those in different clusters. Consequently, applications with very low-IPC, such as `ldint_l2`, significantly deteriorate the performance of all their co-runners in all three configurations. However, the placement of two cache-bound threads in the same cluster results in further slowdown caused by resource conflicts. Similarly to the results for 2 threads, we notice that `ldint_mem` is a good co-runner due to the hardware mechanism that prevents LLC-missing threads from clogging shared resources.

Overall, with two pair of threads in a core we conclude that the best thread placement is [AB][BA] as in this configuration most of the resources are shared between threads with different resource profiling.

Thread placement of 4 single-thread micro-benchmarks. We perform a case study with 4 workloads consisting of different threads. We try different placements to investigate the cases which provide the optimal improvement with regard to core throughput. Like in previous experiments we expect maximum performance when we minimize resource conflicts in the core.

In Figure 5 we see various combinations of the five representative micro-benchmarks we have used in previous experiments. The IPC bars of each micro-benchmark are stacked in order to add up to the total throughput. We have seen that symmetric pairs (or in this case quartets) lead to similar behavior *in a specific core mode*. Also, according to our previous results with 2 two-thread micro-benchmarks, inter-cluster resource sharing causes small performance differences, so configurations such as [AB][CD] and [AB][DC] lead to similar performance. We confirm this in our experiments and reduce the total number of possible placements to 3: [AB][CD], [AC][BD] and [AD][BC].

First of all, we notice that the placement of the four executing threads plays a great role in the final performance. We can see that different thread placements lead to impressive speedups of up to 1.66x in throughput for a quartet of threads.

In order to get the maximum performance we need to follow a similar approach as to that when running 2 pairs of threads. This can be done by taking advantage of the automatic resource control mechanism that is activated with `ldint_mem`, while avoiding resource conflicts as much as possible. Also, in accordance with our previous findings, it is clear that the presence of a cache-bound application like `ldint_l2` in the quartet considerably lowers the performance of all other co-runners. By combining tasks of different profiles in a cluster we get the maximum possible performance.

Therefore, we should place the threads with the highest IPC in the same cluster as a memory-bound thread -if available, and make sure cache and memory-bound applications are put in different clusters.

B. Thread Count: Total Throughput

In this section we show the throughput of one core of the IBM POWER7 when several copies of each micro-benchmark are executing. In Figure 6 we compare the cases of one thread in isolation, two copies of a thread in SMT2 ([AA][XX]) and SMT4 ([AX][XA]) and four copies of a thread. The results vary from benchmark to benchmark but we can draw some general conclusions. As a rule of thumb, the lowest throughput is obtained when executing a single copy of a micro-benchmark and the highest when executing four copies of the

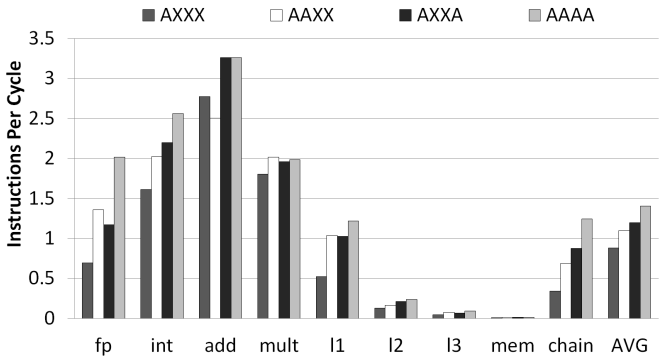


Figure 6. Core throughput when different numbers of copies of the micro-benchmarks are run.

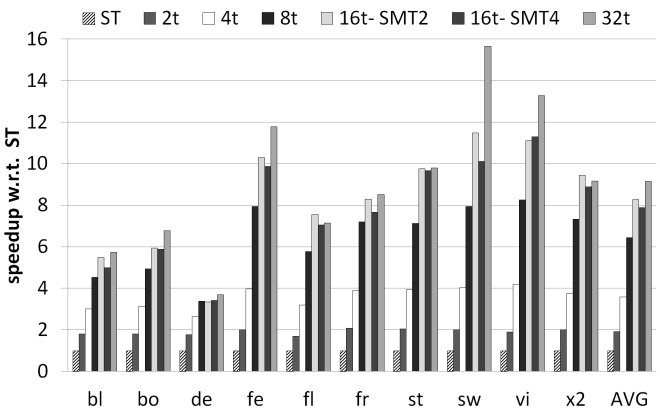


Figure 7. PARSEC speedup w.r.t Single-thread execution of different thread-count and placement setups.

micro-benchmark in the same core. Some benchmarks almost reach their peak with one or two running copies, whereas some others noticeably improve their throughput from a single thread to four threads together. Indicatively for `cpu_fp` core throughput improves by 190% and for `lng_chain` by 263% when using all four contexts w.r.t. running one thread.

The throughput of `cpu_fp` scales almost linearly up to 4 threads. This is expected if we consider that besides the fact that it is comprised of long latency floating-point operations, the Vector Scalar Unit (VSU) in POWER7 is highly pipelined and capable for dual instruction issue, with each pipe supporting different types of instructions [12].

`Cpu_int_add` saturates with 2 threads, as the peak of execution to issuing ratio is reached in the FXU and LSU (that also performs simple fixed-point instructions). On the other hand, `cpu_int_mul` almost saturates with 1 thread. Compared to `cpu_int_add`, it has a higher instruction latency (integer multiplications), that bottleneck the two FX pipelines and do not leave much space for throughput improvement with more threads. `Cpu_int` and `lng_chain` are comprised of a mix of instructions of the two previous micro-benchmarks and their behavior is more variable from one to four executing threads in the core.

The various cache and memory-bound benchmarks improve their throughput by as high as 133% when increasing the threads from one to four, but with the main improvement noted when moving to two threads. As we mentioned before, consecutive instructions from these micro-benchmarks cannot be run in parallel due to the pointer-chasing technique. Nevertheless, load instructions *from different threads* can be executed in parallel by the two LS pipelines. Of course, the ability to fetch data in parallel from a certain level of memory hierarchy depends on the amount of cache banks that level has and the ability to access them in parallel. It is known that the L1 cache is highly banked to allow multiple reads at the same time as long as they are in different banks.

In any case the maximum throughput is gained when running all four threads, despite the fact that the same might be reached with fewer threads. The abundance of resources in each POWER7 core, combined with the instruction fetch balancing between different threads contribute to maximizing throughput when running 2 or 4 threads in a core.

V. Thread Placement with PARSEC Applications

In this section we evaluate the effect of different thread placements when running PARSEC parallel applications on the POWER7 processor.

A. Thread Placement of a Single Parallel Application

Firstly, we measure the scalability of each PARSEC benchmark with the number of threads. We start by running a single-thread version of each benchmark and, in subsequent experiments, we run parallelized versions of each benchmark with 2, 4, 8, 16 and 32 threads. Figure 7 shows the speedups obtained over the single-thread version of each benchmark.

When using 2 and 4 threads, we bind each thread to the first context of a different core, thus switching them to ST mode. In the case of 8 threads, we evaluate all four possible placements with one thread per core: [AX][XX], [XA][XX], [XX][AX] and [XX][XA]), with each core operating in ST, SMT2, SMT4 and SMT4 mode, respectively. We observe that the fastest execution time for all benchmarks is achieved in ST mode, followed closely by the SMT2 mode with a degradation of 3.1% in execution time. The worst performance is obtained in SMT4 mode, with an execution time degradation of 20.2%. These results are consistent with our findings with micro-benchmarks in the previous section.

Under the 16-thread count setup, we bind two threads per core. We try three configurations: running both threads in the first cluster in SMT2 mode ([AA][XX]), or in different clusters in SMT4 mode ([AX][AX] and [AX][XA]). We notice that the two SMT4 configurations are very similar in performance, with [AX][XA] being slightly better on average. Consequently, we omit the [AX][AX] case from Figure 7. Between the two core modes, SMT2 performs better than SMT4 for all benchmarks (4.9% better on average), with the exception of `vips` and `dedup` that perform slightly better in SMT4 mode. We see that in contrast to several micro-benchmarks, few real-world PARSEC applications benefit from the extra rename registers in SMT4 mode, and therefore, generally perform better in SMT2 mode.

Under the 32-thread setup, we bind 4 threads per core. Given that all threads running in the same core execute similar

code, thread placement does not significantly change the final performance of the application. We observe that the majority of applications scale up to 32 threads. Only two benchmarks perform better with 16 threads (`fluidanimate` and `x264`), but only by a small margin. We saw in Section IV-B that with micro-benchmarks the maximum core throughput was reached with either 2 or 4 threads, although throughput with 4 threads was never lower than with 2 threads. Likewise, for the majority of PARSEC applications 32-thread setups execute in equal or shorter time than 16-thread setups.

Overall, to optimize the performance of a given parallel application, we have to take into account the thread number and their placement. We confirm the conclusions we drew with micro-benchmarks: When only one thread runs per core, the optimal configuration is ST mode. When 2 threads of a single application are executing in a core, the best configuration is SMT2 mode. Finally, we notice that the optimal throughput is achieved when running 2 or 4 threads per core, for a total of 16 and 32 threads respectively, with the latter leading to significantly higher speedup on average.

B. Thread Placement of 2 Parallel Applications

The increase in execution time when moving from 32 to 16 threads for PARSEC benchmarks is only 7.2% on average. All benchmarks except `swaptions` show very small differences between these two setups. This reduced performance degradation motivates us to execute two 16-thread PARSEC applications simultaneously. Several authors advocate for not allocating parallel applications to the same core [7] [14], since threads from different applications fight for shared intra-core resources and do not benefit from having a shared L1 data cache. However, proper thread placement can help minimizing resource conflicts in the core. Moreover the considerably large size of the inner levels of the cache hierarchy in the POWER7, contribute to reduced cache contention between different applications. Given applications A and B, we evaluate the following configurations:

1) *Sequential*: We execute applications A and B sequentially. We start by executing A in isolation and once it finishes, we execute B in isolation. Each application is executed under its optimal thread number and placement, i.e., `x264` and `fluidanimate` run in 16 threads in SMT2 mode, while the rest run in 32 threads in SMT4 mode.
2) *Split cores*: We execute applications A and B together, with the first 4 cores running 16 threads of A, and the last 4 cores running 16 threads of B. Here each core is shared only by threads of the same application.
3) *Shared cores*: We execute applications A and B together, with all 8 cores executing 2 threads of each application simultaneously, for a total of 16 threads per application. After trying all three possible placements, we confirm our micro-benchmarks conclusion that [AB][BA] is the best configuration on average when running 2 pairs of threads in a core.

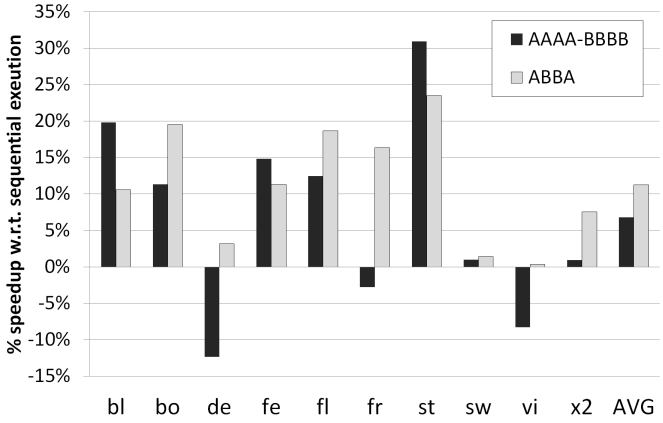


Figure 8. Average execution time improvement for each PARSEC application against all other in cases 2 and 3.

For cases 2 and 3, we compare the time required for both applications to finish executing, to the sum of execution times of A and B in case 1. Our goal is to find out whether we can benefit from running two applications simultaneously or not.

We run each PARSEC application against all other 9 applications. For a given benchmark we compute the execution time improvement of cases 2 and 3 over case 1 for all 9 benchmarks. Figure 8 reports the average improvement per benchmark, where configuration 'AAAA-BBBB' stands for case 2 and 'ABBA' for case 3.

Since threads of the same application have more similar hardware resource requirements than threads of different applications, assigning threads of different applications to the same cluster leads to reduced resource conflicts. According to Zhang et al [20], PARSEC benchmarks do not share high amounts of data on the L1 and L2 caches and so co-running two of these applications in a core should not lead to significant cache contention.

Only `dedup`, `freqmine` and `vips` experience slowdowns due to a high contention for common core resources when running 4 threads per core (case 2). These benchmarks execute close to 50% of their instructions in the FXU, while the first two do not include any floating-point instructions. As a result, running 4 threads of these applications in the same core leads to many conflicts in the FXU and thus a decrease in performance. However, when the same benchmarks co-execute with other applications in a core (case 3), we can see clear improvements that stem from reduced contention for shared core resources.

Other benchmarks such as `blackscholes`, `bodytrack`, `ferret`, `fluidanimate` and `streamcluster` perform well when their threads share a core both with threads of the same application (case 2) and with threads of a different one (case 3). Especially `bodytrack` and `ferret` benefit significantly from co-execution despite the fact that they run noticeably slower with 16 than with 32 threads. Nevertheless, it is `streamcluster` -a memory-bound clustering application, that is the best co-runner in both cases. It is by far the application with the highest amount of memory loads amongst the ones we tried, though it does not fetch high amounts of data

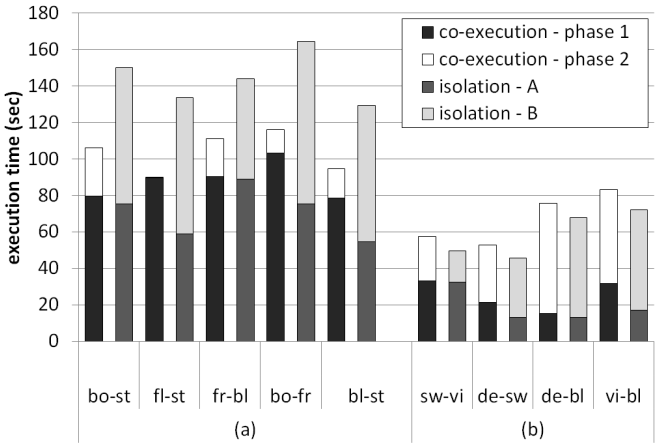


Figure 9. PARSEC applications pairs that (a) benefit and (b) suffer from sharing the core. The first benchmark in each pair corresponds to task A, and the second one to B.

from memory that could cause bandwidth issues. We have seen that memory-bound micro-benchmarks are good co-runners in a core. Additionally, `streamcluster` has a balanced instruction mix: 30% load and store, 30% fixed-point, and 30% floating-point instructions. This balanced utilization of resources helps when sharing the core amongst its threads, or with threads of another application. The largest improvements are obtained for `streamcluster` with `fluidanimate` in case 3 (45.5%) and with `blackscholes` in case 2 (54%).

In general, few pairs demonstrate worse performance when sharing the core than when running sequentially (only 12.2% of the total pairs increase their execution time by 5% or more). These pairs are mainly formed by applications that suffer from running with a suboptimal thread number, like `swaptions` or `vips`. However, the number of pairs that suffer a slowdown in case 2 with respect to case 1 is significant (37.8% of the total pairs increase their execution time by 5% or more). This difference could be caused by increased intra-core resource conflicts between threads of the same application. In any case, the majority of applications benefit from co-executing simultaneously, primarily when sharing the core with another application (case 3), with an average execution time improvement of 11.2% for all pairs.

Figure 9 shows a selection of pairs with (a) the best and (b) the worst cases of core-sharing in [AB][BA] (case 3). When two applications run in parallel, one finishes before the other. The time the two applications run together is denoted *co-execution phase 1* in Figure 9. The extra time that the slowest application needs to run alone until it finishes is denoted *co-execution phase 2*. In contrast, the second bar per pair of benchmarks stacks the execution times of the two applications while running sequentially in isolation. The vertical axis shows execution time, so the shorter the bar the faster the execution.

Figure 9 complements Figure 8, as it shows some cases of co-executing pairs of parallel applications. It also provides a visualization of our run methodology and the different phases of co-execution, while depicting how both applications' execution times are affected when running together. We observe that, while for pairs in Figure 9 (a) the duration of co-execution phase 2 is small, for pairs in Figure 9 (b) it is relatively large. For the latter pairs, this suggests that a significant portion of the slowest application's workload could not be carried out during phase 1 of co-execution due to increased resource conflicts. Note that the pairs in (a) include four of the best co-runners we saw in Figure 8, while the pairs in (b) include the three worst co-runners.

Overall, we conclude that co-executing two parallel applications leads to clear improvements in performance, either by sharing the core with another applications (preferably in [AB][BA] configuration), or by occupying half of the cores each. Also, we have to make sure that execution with less threads does not imply a significant slowdown with respect to the optimal case.

C. Thread Disabling

When there are no runnable threads in the task run-queue of a specific hardware thread, the OS assigns the idle thread to it. This is the lowest priority thread that performs an infinite loop, is always runnable and has very low IPC. Since the placement of a thread sets the core mode in the POWER7 processor, the existence of one or more idle threads can reduce the amount of available hardware resources to running applications, thus potentially impacting their performance. Modern versions of the Linux kernel automatically disable hardware threads that are idle for more than a certain time (the default value is 20 ms): first the idle thread reduces its hardware thread priority to 1 (low-power mode) and then, after 20 ms, it disables the hardware thread. External events, such as interrupts raised by the decrementer, an I/O device or another hardware thread, re-activate the disabled hardware thread, which handles the interrupt and invokes the scheduler to check whether any runnable thread has been assigned to its run-queue. If no other thread is assigned to the context, the idle loop restarts. In order to further improve performance by preventing these alternating phases from dynamically switching the core mode, we need to manually turn off unused hardware threads.

In any case, the existence of this automatic mechanism guarantees an improvement in performance in comparison to the case without this feature. This is especially important in situations where manually disabling the threads is not an option, such as parallel applications with load imbalance. In such applications, some threads finish executing faster than others and idle threads take their place. Consequently the core mode remains set in SMT4 mode, even if only one thread from the application is running. As we saw, this results in an important performance decrease. When running PARSEC applications with 32 threads, we experienced a difference in performance between kernels with or without the automatic thread-disabling feature of 22% on average, and up to 143% for `blackscholes`.

VI. Related Work

Previous works show that SMT performance heavily depends on the nature of the concurrently running applications [5], [6], [13]. Tuck et al. analyze the performance of

a real SMT processor [15], concluding that SMT architectures provide an average speedup over single-thread architectures of about 20% and that, even if the processor is designed to isolate threads, performance is still affected by resource conflicts. Vega et al. discuss the issue of thread placement on the IBM POWER7 with regard to energy efficiency [18]. They conclude that significant power saving can be drawn from proper thread placement with just a small performance tradeoff.

Other works propose the use of hardware thread priorities to control thread execution in SMT processors. Many of these proposals implement fetch policies to maximize throughput and fairness by reducing the priority, stalling, or flushing threads that experience long latency [4], [16].

Finally, other authors characterize resource sharing on Intel processors [7], [14]. These works do not study the effect of thread placement since the amount of shared resources on these processors does not vary as much as in the POWER7 processor. Cakarevic et al. [3] characterize the Oracle UltraSPARC T2 that has three resource sharing levels, but does not feature different core modes (ST, SMT2 and SMT4).

VII. Discussion and Conclusions

In this paper we have focused on the POWER7 processor as representative of current multi-TLP processors. Our characterization provides a useful study on the resource sharing complexities of that particular processor and how to alleviate them with an appropriate placement of threads. We have shown which of these configurations are the most promising in terms of thread performance and throughput when increasing the number of threads from one to four inside the core. We drew useful conclusions on POWER7's resource sharing levels using METbench micro-benchmarks, and applied this knowledge to PARSEC parallel applications. By co-executing parallel applications in the same core, we achieved 11.2% average execution time improvement with the proper thread placement.

Our study has shown the potential of thread placement to improve system performance. Similar results could be obtained with other processors that also present different levels of shared resources and can benefit from thread placement. As processors are expected to have an increasing number of accelerators and heterogeneous cores with a varying amount of hardware resources, resource sharing complexities will be even higher, increasing the importance of thread placement.

On the hardware level, processor manufacturers should provide hardware mechanisms that allow the OS to control the level of resource sharing between threads. Multithreaded processors should also provide better performance counters to identify the sensitivity of each hardware thread to different resources, simplifying the work of the OS to better adapt to applications' varying resource requirements.

On the software level, the OS can initially schedule threads on the CMP level, while using the extra SMT contexts for higher thread numbers. Preferably hardware contexts in the same core should be shared by threads of different applications, as threads of the same application have more similar resource sharing profiles. Additionally, thread re-assigning after the beginning of execution is important when co-executing two or more applications, or when running applications with load imbalance. After some threads finish running, the scheduler should bind active threads to their optimal placement, based on the conclusions of our study, so that they do not execute in a suboptimal configuration. When more than 2 SMT modes are available, the OS should automatically detect the adequate mode through inspection or by taking advantage of user-derived guidelines. Furthermore, software developers should provide specific application resource requirements, facilitating decisions made by the OS. A software stack that is aware of hardware resource sharing complexities would allow it to take advantage of the potential of thread placement techniques, resulting in more efficient systems.

Acknowledgment

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625, and by the HiPEAC Network of Excellence. M. Moreto is supported by a MEC/Fulbright Fellowship.

References

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. In *ISCA*, pages 415–426, 2008.

[3] V. Cakarevic, P. Radojkovic, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the ultrasparc T2 processor. In *MICRO*, pages 481–492, 2009.

[4] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández. Dynamically controlled resource allocation in smt processors. In *MICRO*, pages 171–182, 2004.

[5] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. Qos for high-performance smt processors in embedded systems. *IEEE Micro*, 24(4):24–31, July 2004.

[6] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *IPDPS*, pages 140–140, 2006.

[7] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *PPoPP*, pages 236–246, 2005.

[8] O. Lempel. 2nd generation intel core processor family: Intel core i7, i5 and i3. *Hot Chips*, August 2011.

[9] A. Morari, C. Boneti, F. J. Cazorla, R. Gioiosa, C.-Y. Cher, P. Bose, and M. Valero. SMT Malleability in IBM POWER5 and POWER6 Processors. *IEEE Trans. Computers*, 99(Preprint), 2012.

[10] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31, 1996.

[11] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, UCSB, 1993.

[12] B. Sinharoy et al. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, 55:191–219, May 2011.

[13] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, Nov. 2000.

[14] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011.

[15] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *PACT*, 2003.

[16] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO*, pages 318–327, 2001.

[17] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA*, 1995.

[18] A. Vega, P. Bose, and A. Buyuktosunoglu. Power-aware thread placement in SMT/CMP architectures. In *WEED*, 2012.

[19] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernández, and M. Valero. Fame: Fairly measuring multithreaded architectures. In *PACT*, pages 305–316, 2007.

[20] E. Z. Zhang, Y. Jiang, and X. Shen. The significance of CMP cache sharing on contemporary multithreaded applications. *IEEE Trans. Parallel Distrib. Syst*, 23:367–374, 2012.