Feliu-Pérez, J.; Eyerman, S.; Sahuquillo Borrás, J.; Petit Martí, SV.; Eeckhout, L. (2017). Improving IBM POWER8 Performance Through Symbiotic Job Scheduling. IEEE Transactions on Parallel and Distributed Systems. 28(10):2838-2851. doi:10.1109/TPDS.2017.2691708

Improving IBM POWER8 Performance through Symbiotic Job Scheduling

Josué Feliu, Stijn Eyerman, Julio Sahuquillo, *Member, IEEE,* Salvador Petit, *Member, IEEE,* and Lieven Eeckhout, *Senior Member, IEEE.*

Abstract—Symbiotic job scheduling, i.e., scheduling applications that co-run well together on a core, can have a considerable impact on the performance of processors with simultaneous multithreading (SMT) cores. SMT cores share most of their microarchitectural components among the co-running applications, which causes performance interference between them. Therefore, scheduling applications with complementary resource requirements on the same core can greatly improve the throughput of the system.
This paper enhances symbiotic job scheduling for the IBM POWER8 processor. We leverage the existing cycle accounting mechanism to build an interference model that predicts symbiosis between applications. The proposed models achieve higher accuracy than previous models by predicting job symbiosis from throttled CPI stacks, i.e., CPI stacks of the applications when running in the same SMT mode to consider the statically partitioned resources, but without interference from other applications. The symbiotic scheduler uses these interference models to decide, at run-time, which applications should run on the same core or on separate cores.
We prototype the symbiotic scheduler as a user-level scheduler in the Linux operating system and evaluate it on an IBM POWER8 server running multiprogram workloads. The symbiotic job scheduler significantly improves performance compared to both an agnostic random scheduler and the default Linux scheduler. Across all evaluated workloads in SMT4 mode, throughput improves by 12.4% and 5.1% on average over the random and Linux schedulers, respectively.

Index Terms—Symbiotic job scheduling, performance estimation, interference model, IBM POWER8, simultaneous multithreading (SMT)

1 Introduction

The current manycore/manythread era generates a lot of challenges for computer scientists, going from productive parallel programming, over network congestion avoidance and intelligent power management, to circuit design issues. The ultimate goal is to squeeze out as much performance as possible while limiting power and energy consumption and guaranteeing a reliable execution. A scheduler is an important component of a manycore/manythread system, as there are often a combinatorial amount of different ways to schedule multiple threads or applications, each with a different performance due to interference among applications. Picking an optimal schedule can result in substantial performance gain.

Selecting which applications to run on which cores or thread contexts has an impact on performance because cores share resources for which threads compete. As such, threads can interfere with each other, causing performance degradation or improvement for other threads. The level of sharing is not equal for all cores or thread contexts: all cores on a chip usually share the memory system, but a cache can be shared by smaller groups of cores, and threads on an SMT-enabled core share almost all of the core resources. Good schedulers should reduce negative interference as much as possible by scheduling complementary tasks close to each other.

- *J. Feliu, J. Sahuquillo, and S. Petit are with the Department of Computer Engineering (DISCA), Universitat Politécnica de Valéncia, Spain. E-mail: jofepre@gap.upv.es, {jsahuqui,septit}@disca.upv.es*
- *S. Eyerman is with Intel Belgium. This work was done while S. Eyerman was with Ghent University. E-mail: stijn.eyerman@intel.com*
- *L. Eeckhout is with the Department of Electronics and Information Systems, Ghent University, Belgium. E-mail: lieven.eeckhout@UGent.be*

The most prevalent architecture for high-end processors is a chip-multicore processor (CMP) consisting of simultaneous multithreading (SMT) cores (e.g., Intel Xeon and IBM POWER servers). Scheduling for this architecture is particularly challenging, because SMT performance is very sensitive to the characteristics of the co-running applications. When the number of available threads exceeds core count, the scheduler must decide which applications should run together on one core. Selecting the optimal schedule is an NP-hard problem [13], and predicting the performance of a schedule is a non-trivial task due to the high degree of sharing on an SMT core.

This paper presents a new scheduler for the IBM POWER8 [23] architecture, which is a multicore processor on which every SMT core can execute up to 8 threads. It provides both high single-threaded performance by means of aggressive out-of-order cores that can dispatch up to 8 instructions per cycle, as well as high parallelism, with 80 available thread contexts on our system (10 cores times 8 threads per core). This recent architecture is chosen for this study because of its high core count, which makes the scheduling problem more challenging, and because of the availability of an extensive performance counter architecture, including a built-in mechanism to measure cycles per instruction (CPI) stacks.

Previous work on symbiotic job scheduling for SMT uses sampling to explore the space of possible schedules [24], relies on novel hardware support [8], or performs an offline analysis to predict the interference between applications on an SMT core [27]. In contrast to these works, we propose an online model-based scheduler [10] that does not require sampling and can be used on an existing commercial processor. The devised scheduler leverages the existing CPI stack accounting mechanism on the IBM POWER8 to build a model that predicts the interference among threads on an SMT core. Using this model, it is possible

to quickly explore the schedule space, and select the optimal schedule for the next time slice. As the scheduler constantly monitors the CPI stacks of all applications, it can also quickly adapt to phase behavior.

In this work we extend our symbiotic job scheduler [10], making the following contributions.

1) We redefine the construction of the model such that the interference of applications running on the same SMT core is predicted more accurately. This is done by estimating job symbiosis from *throttled* single-threaded ($ST_{throttled}$) CPI stacks, instead of using isolated ST CPI stacks (see Section 3.2). This also allows the symbiotic scheduler to get rid of the correction factors used in our previous work [10].
2) The scheduler algorithm is extended to four SMT threads. Due to state explosion, an approximate optimization algorithm is required to find a (close to) optimal schedule in a limited time frame. The scheduler implementation has also been enhanced to allow the applications to run while the process selection is being performed, avoiding the overhead of finding out the best schedule.
3) We explain the performance differences among cores that occur in our experimental platform and associate them to the non-uniform memory access (NUMA) effects that affect our system due to the fact that only one memory module is installed. To deal with these effects we implement a NUMA-aware symbiotic scheduler, which allocates the application combinations that perform more memory accesses on the fastest NUMA node.
4) We evaluate the enhanced symbiotic job scheduler and its NUMA-aware version on the IBM POWER8 running workloads comprising two and four applications per core, i.e., SMT2 and SMT4 modes, respectively[1].

The proposed NUMA-aware symbiotic scheduler performs 9.9% and 12.4% better than a random scheduler in SMT2 and SMT4 modes, respectively, across all evaluated workloads, consisting of 8 to 20 applications in SMT2 mode and 16 to 40 applications in SMT4 mode. It also performs 5.2% and 5.1% better than the default Linux scheduler in SMT2 and SMT4 modes across the same workloads. Moreover, the performance benefits with respect to Linux can be as high as 11.0%, achieved in SMT2 mode and 6-core workloads. The overhead of using a performance model and exploring the possible schedules is negligible. Furthermore, our scheduler is completely software-based and with appropriate training of the model it could also be used for other architectures.

2 Related Work

Simultaneous multithreading (SMT) was proposed by Tullsen et al. [26] as a way to improve the utilization and throughput of a single core. Enabling SMT increases the area and power consumption of a core (5% to 20% [2], [14]), mainly due to replicating architectural and performance-critical structures, but it can significantly improve throughput. Recently, Eyerman and Eeckhout [9] show that a multicore processor consisting of SMT cores has an additional benefit other than increasing throughput. SMT is flexible when thread count varies: if thread count is low, per-thread performance is high because only one or a few threads execute concurrently on one core, whereas if thread count is high, it can increase throughput by executing more threads concurrently. As such, a multicore consisting of SMT cores performs as well as or even better than a heterogeneous multicore that has a fixed proportion of fast big cores and slow small cores.

The importance of intelligently selecting applications that should run together on an SMT core has been recognized quickly after the introduction of SMT. The performance benefit heavily depends on the characteristics of the co-running applications, and some combinations may even degrade total throughput, for example due to cache trashing [12]. Snavely and Tullsen [24] were the first to propose a mechanism to decide which applications should co-run on a core to obtain maximum throughput. At the beginning of every scheduler quantum, they shortly execute all (or a subset of) the possible combinations, and select the best performing combination for the next quantum. Because the number of possible combinations quickly grows with the number of applications and hardware contexts, the overhead of sampling the performance quickly becomes large and/or the fraction of combinations that can be sampled becomes small. To overcome the sampling overhead, Eyerman and Eeckhout [8] propose model-based coscheduling. A fast analytical model predicts the slowdown each application encounters when coscheduled with other applications, and the best performing combination is selected. However, the inputs for the model require hardware support, which is not available in current processors. Our proposal uses a similar model, but it avoids sampling overhead and it uses existing performance counters.

Other studies have explored the use of models and profiling to estimate the SMT benefit. Moseley et al. [18] use regression on performance counter measurements to estimate the speedup of SMT when coexecuting two applications. Porter et al. [20] estimate the speedup of a multithreaded application when enabling SMT, based on performance counter events and machine learning. Settle et al. [22] predict job symbiosis using offline profiled cache activity maps. Feliu et al. [11] propose to balance L1 cache bandwidth requirements across the cores in order to reduce interference and improve throughput. Mars et al. [15] use microbenchmarks called *bubbles* to measure first, the pressure on the memory subsystem that the applications generate, and second, how much the applications suffer from different levels of pressure in the memory subsystem introduced by the *bubbles*. Using this information, obtained during a characterization phase, the complexity of finding good co-locations of the applications is reduced. In follow-up work, Zhang et al. [27] propose a similar methodology to predict the interference among threads on an SMT core. They develop microbenchmarks called *rulers* that stress different core resources, and by co-running each application with each ruler in an offline profiling phase, the sensitivity of each application to contention in each of the core resources is measured. By combining resource usage and sensitivity to contention, interference can be predicted and used to guide the scheduling. Our proposal does not require an offline profiling phase for each new application, and it takes into account the impact of contention in all shared resources, not only cache and memory contention.

3 Predicting Job Symbiosis

Our symbiotic scheduler for a CMP of SMT cores is based on a model that estimates job symbiosis. The model predicts for any combination of applications, how much slowdown each of the applications would experience if they were co-run on an SMT core. It is fast, which enables us to explore all possible

1. We do not evaluate the symbiotic scheduler in SMT8 mode since we did not observe performance benefits in SMT8 mode over SMT4 mode with the Linux scheduler and our SPEC multiprogram workloads. See Section 5 for further details.

combinations. The model only requires inputs that are readily obtainable using performance counters.

3.1 Interference model

The model used in our scheduler is based on the model proposed by Eyerman and Eeckhout [8], which leverages CPI stacks to predict job symbiosis. A CPI stack (or breakdown) divides the execution cycles of an application on a processor into various components, quantifying how much time is spent or lost due to different events, see Figure 1 on the left. The base component reflects the ideal CPI in the absence of miss events and resource stalls. The other CPI components account for the *lost* cycles, where the processor is not able to commit instructions due to different resource stalls and miss events. The SMT symbiosis model uses the CPI stacks of an application when executed in single-threaded (ST) mode, and then predicts the slowdown by estimating the increase of the components due to interference, see Figure 1 on the right. Eyerman and Eeckhout [8] estimate interference by interpreting normalized CPI components as probabilities and calculating the probabilities of events that cause interference. For example, if an application spends half of its cycles fetching instructions, and the other application one third of its execution time, there is a 1/6 probability that they want to fetch instructions at the same time, which incurs a delay because the fetch unit is shared. However, Eyerman and Eeckhout use novel hardware support [7] to measure the ST CPI stack components during multi-threaded execution, which is not available in current processors.

Interestingly, the IBM POWER8 has a built-in cycle accounting mechanism, which generates CPI stacks both in ST and SMT mode. However, this accounting mechanism is different from the cycle accounting mechanisms proposed by Eyerman and Eeckhout for SMT cores [7], which means that their model [8] cannot be used readily. Some of the components relate to each other to some extent (e.g., the number of cycles instructions are dispatched [7] versus the number of cycles instructions are committed for the POWER8), but provide different values. Other counters are not considered a penalty component in one accounting mechanism, while they are accounted for in the other mechanism, and vice versa. For example, following Eyerman and Eeckhout approach [7], a long-latency instruction only has a penalty if it is at the head of the reorder buffer (ROB) and the ROB gets completely filled (halting dispatch), while for the IBM POWER8 accounting mechanism, the penalty starts from the moment that the long-latency instruction inhibits committing instructions, which could be long before the ROB is full. On the other hand, the entire miss latency of an instruction cache miss is accounted as a penalty by Eyerman and Eeckhout [7], while for the POWER8 accounting mechanism, the penalty is only accounted from the moment the ROB is completely drained (which means that the penalty could be zero if the miss latency is short and the ROB is almost full). Furthermore, some POWER8 CPI components are not well documented, which makes it difficult to reason about which events they actually measure.

Because of these differences, we develop a new model for estimating the slowdown caused by co-running threads on an SMT core. The model uses regression, which is more empirical than the purely analytical model by Eyerman and Eeckhout [8], but its basic assumption is similar: we normalize the CPI stack by dividing each component by the total CPI, and interpret each component as a probability. We then calculate the probabilities that interfering events occur at the same time, which cause some delay

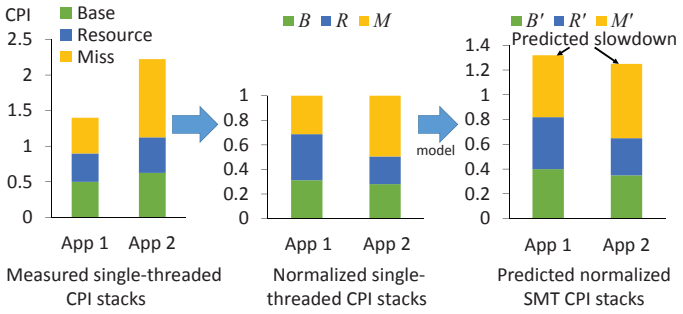


Fig. 1. Overview of the model: first, measured CPI stacks are normalized to obtain probabilities; then, the model predicts the increase of the components and the resulting slowdown (1.32 for App 1 and 1.25 for App 2).

that is added to the CPI stack as interference. The components are divided into three categories: the base component, resource stall components and miss components. The model for each category is discussed in the following paragraphs. For now, let us assume that we have the ST CPI stacks at our disposal, measured off-line using a single-threaded execution on the POWER8 machine. This assumption will no longer be necessary in Section 3.3. The stack is normalized by dividing each component by the total CPI, see Figure 1. We denote B the normalized base component, R_i the component for stalls on resource i, and M_j the component for stalls due to miss event j (e.g., instruction cache miss, data cache miss, branch misprediction). We seek to find the CPI stack when this application is co-run with other applications in SMT mode, for which the components are denoted with a prime (B', R'_i, M'_j).

3.1.1 Base component

The base component in the POWER8 cycle component stack is the number of cycles (or fraction of time after normalization) where instructions are committed. It reflects the fraction of time the core is not halted due to resource stalls or miss events. During SMT execution, the dispatch, execute and commit bandwidth are shared between threads, meaning that even without miss events and resource stalls, threads interfere with each other and cause other threads to wait.

We find that the base component in the CPI stack increases when applications are executed in SMT mode compared to ST mode. This is because multiple threads can now commit instructions in the same cycle, so each thread commits fewer instructions per cycle, meaning that the number of cycles that a thread commits instructions increases. The magnitude of this increase depends on the characteristics of the other threads. If the other threads are having a miss or resource stall, then the current thread can use the full commit bandwidth. If the other threads can also commit instructions, then there is interference in the base component. So, the increase in the base component of a thread depends on the base component fractions of the other threads: if the base components of the other threads are low, there is less chance that there is interference in this component, and vice versa.

We model the interference in the base component using Equation 1. For a given thread j, B_j represents its base component when running in ST mode (the ST base component), while B'_j identifies the SMT base component of the same thread.

$$B'_j = \alpha_B + \beta_B B_j + \gamma_B \sum_{k \neq j} B_k + \delta_B B_j \sum_{k \neq j} B_k \qquad (1)$$

The parameters α_B through δ_B are determined using regression, see Section 3.2. α_B reflects a potential constant increase in the base component in SMT mode versus ST mode, e.g., through an extra pipeline stage. Because we do not know if such a penalty exists, we let the regression model find this out. The β_B term reflects the fact that the original ST base component of a thread remains in SMT execution. It would be intuitive to set β_B to one (i.e., the original ST component does not change), but the next terms, which model the interference, could already cover part of the original component, and this parameter then covers the remaining part. It can also occur that there is a constant relative increase in the base component, independently of the other applications. In that case β_B is larger than 1. γ_B is the impact of the sum of the base components of the other threads. δ_B specifically models extra interactions that might occur when the current thread (thread j) *and* the other threads have big base components, similar to the probabilistic model of Eyerman et al. [8] (a multiplication of probabilities). Although not all parameters have a clear meaning, we keep the regression model fairly general to be able to accurately model all possible interactions.

3.1.2 Resource stall components

A resource stall causes the core to halt because a core resource (e.g., functional unit, issue queue, load/store queue) is exhausted or busy. In the POWER8 cycle accounting, a resource stall is counted if a thread cannot commit an instruction because it is still executing or waiting to execute on a core resource (i.e., not due to a miss event). By far, the largest component we see in this category is a stall on the floating-point unit, i.e., a floating-point instruction is still executing when it becomes the oldest instruction in the ROB. This can have multiple causes: the latency of the floating-point unit is relatively large, there are a limited number of floating-point units, and some of them are not pipelined. We expect a program that executes many floating-point instructions to present more stalls on the floating-point unit, which is confirmed by our experiments. Along the same line, we expect that when co-running multiple applications with a large floating-point unit stall component, the pressure on floating-point units will increase even more. Our experiments show that in this case, the floating-point stall component per application indeed increases. Therefore, we propose the following model to estimate the resource stall component in SMT mode ($R_{j,i}$ represents the ST stall component on resource i for thread j):

$$R'_{j,i} = \alpha_{Ri} + \beta_{Ri} R_{j,i} + \gamma_{Ri} \sum_{k \neq j} R_{k,i} + \delta_{Ri} R_{j,i} \sum_{k \neq j} R_{k,i} \qquad (2)$$

Similar to the base component model, α indicates a constant offset that is added due to SMT execution (e.g., extra latency). β indicates the fraction of the single-threaded component that remains in SMT mode, while the term with γ models the fact that resource stalls of the other applications can cause resource stalls in the current application, even if the current application originally had none. The last term models the interaction: if the current application already has resource stalls, and one or more of the other applications too, there will be more contention and more stalls.

3.1.3 Miss components

Miss components are caused by instruction and data cache misses at all levels, as well as by branch mispredictions. In contrast to resource stall components, a miss event of a thread does not directly cause a stall for the other threads. For example, if one thread has an instruction cache miss or a branch misprediction, the other threads can still fetch instructions. Similarly, on a data cache miss for one thread, the other threads can continue executing instructions and access the data cache. One exception is that a long-latency load miss (e.g., a last-level cache (LLC) miss) can fill up the ROB with instructions of the thread causing the miss, leaving fewer or no ROB entries for the other threads. As pointed out by Tullsen et al. [25], this is a situation that should be avoided, and we suspect that current SMT implementations (including POWER8) have mechanisms to prevent this to happen.

However, misses can interfere with each other in the branch predictor or cache itself. For example, a branch predictor entry that was updated by one thread can be overwritten by another thread's branch behavior, which can lead to higher or lower branch miss rates. Similarly, a cache element belonging to one thread can be evicted by another thread (negative interference) or a thread can put data in the cache that is later used by another thread if both share data (positive interference). Furthermore, cache misses of different threads can also contend in the lower cache levels and the memory system, causing longer miss latencies. Because we only evaluate multiprogram workloads consisting of single-threaded applications, which do not share data, we see no positive interference in the caches.

To model this interference, we propose a model similar to that of the previous two components:

$$M'_{j,i} = \alpha_{Mi} + \beta_{Mi} M_{j,i} + \gamma_{Mi} \sum_{k \neq j} M_{k,i} + \delta_{Mi} M_{j,i} \sum_{k \neq j} M_{k,i} \qquad (3)$$

Although the model looks exactly the same, the underlying reasoning is slightly different. α again relates to fixed SMT effects (e.g., cache latency increase). The β term is the original miss component of that thread, while the γ term indicates that an application can get extra misses due to interference caused by misses of the other applications. We also add a δ interaction term: an application that already has a lot of misses will be more sensitive to extra interference misses and contention in the memory subsystem if it is combined with other applications that also have a lot of misses.

3.2 Model construction and slowdown estimation

The model parameters are determined by linear regression based on experimental training data. This is a less rigorous approach than the model presented by Eyerman and Eeckhout [8], which is built almost completely analytically, but as explained before, this is due to the fact that the cycle accounting mechanism is different and partially unknown.

In our previous work [10], the SMT CPI stacks are predicted from the ST CPI stacks obtained when the applications are running alone in the system. The model mainly focuses on interference between events of different threads, but it does not consider the impact of hardware partitioning. SMT processors share most of their internal resources, which are fully available for an application running alone in ST mode. This resource sharing can be implemented either by applying dynamic sharing or partitioning techniques. For instance, resources such as the ROB or the arithmetic units, among others, are dynamically shared in the POWER8 while other internal resources, like instruction buffers, register renaming tables or load/store buffers are partitioned [23]. If the resources are shared, interference among the threads can rise. On the contrary, if the resources are partitioned there is no interference among

threads, but the performance gap between the ST and SMT modes can grow since a given thread cannot use the resources allocated to another thread. In addition, other characteristics such as instruction dispatch restrictions, prefetching, or branch prediction capabilities also differ among ST and the different SMT modes, further increasing the gap between ST and SMT performance.

Taking the previous rationale into account, and considering that the goal of the model is to estimate the interference between applications, this paper refines the model construction we proposed [10]. In particular, the enhanced models determine the SMT CPI stacks of the applications running in a schedule from their CPI stacks running in the same SMT mode. This CPI stacks consider the statically partitioned structures, but there is no interference on the shared resources caused by other co-running applications. These CPI stacks will be referred to as *throttled*-ST ($ST_{throttled}$) CPI stacks and, from a practical point of view, replace the ST CPI stacks used in Section 3.1 to discuss the interference model. Thus, with the new methodology, the SMT CPI stacks on a 2-application schedule (SMT2 CPI stacks) will be estimated from the $ST_{throttled}$ CPI stacks of the applications in SMT2 mode. Similarly, SMT4 CPI stacks would be predicted from $ST_{throttled}$ CPI stacks of applications executed in SMT4 mode. Notice the difference with our previously proposed models [10], were the SMT CPI stacks were predicted from the ST CPI stacks.

The SMT modes are automatically set by the IBM POWER8 depending on the number of threads running on a core and therefore, the $ST_{throttled}$ CPI stacks cannot be obtained when the applications are executed alone. To solve this problem, we implemented a *nop-microbenchmark*, which is constantly performing *nop* operations. The nop-microbenchmark is intended to force the processor to work in the desired SMT mode while introducing negligible interference at the shared core resources. Thus, it is designed with the opposite goal of other microbenchmarks [15], [27], which are used to introduce contention in the shared resources. Note that obtaining $ST_{throttled}$ CPI stacks is only required to determine the model parameter values, but does not affect how the model and the scheduler work during normal execution.

To train the model, we first run all benchmarks alone in each SMT mode (see Section 5 for the benchmarks we evaluate), and collect the $ST_{throttled}$ CPI stacks every scheduler quantum (100 ms). To run an application in SMT2 or SMT4 modes, it is scheduled on a core with one or three instances of the *nop-microbenchmark*, respectively. We keep track of the instruction count per quantum to determine which part of the program is being executed in each quantum (we evaluate single-threaded programs with a bounded total instruction count). We also normalize each CPI stack to its total CPI.

Next, we execute all possible 2-benchmark mixes and a large and representative set of 4-benchmark mixes on a single core[2]. Notice that the number of possible 4-benchmark mixes exponentially grows with the number of benchmarks and evaluating all of them would take too much time. During these executions, we also collect per-thread CPI stacks and instruction counts for each quantum. Next, we normalize each SMT CPI stack to the previously collected $ST_{throttled}$ CPI of the same instructions. We normalize to the $ST_{throttled}$ CPI because we want to estimate the slowdown each application gets versus its execution alone in the same SMT mode, which equals the SMT CPI divided by the $ST_{throttled}$ CPI (see the last graph in Figure 1). This is also in line with the methodology proposed by Eyerman and Eeckhout [8]. Because the performance of an application differs between $ST_{throttled}$ and SMT executions due to co-runner interference, and the quanta are fixed time periods, the instruction counts do not exactly match between both executions. To solve this problem, we interpolate the $ST_{throttled}$ CPI stacks between two quanta to ensure that $ST_{throttled}$ and SMT CPI stacks are covering approximately the same instructions.

Once the model has been constructed, we can use it to estimate the SMT CPI stacks from the $ST_{throttled}$ CPI stacks for any combination of applications. We first calculate each of the individual components using Equations 1 to 3, and then add all of the components. The resulting number will be larger than one, and indicates the slowdown the application encounters when executed in that combination (see Figure 1 on the right). This information is used to select combinations with minimal slowdown (see Section 4).

3.3 Obtaining $ST_{throttled}$ CPI stacks in SMT mode

Up to now, we assumed that we have the $ST_{throttled}$ CPI stacks available. This is not a practical assumption, since it would require to keep all of the per-quantum $ST_{throttled}$ CPI stacks in a profile. This is a large overhead for a realistic scheduler. An alternative approach is to periodically get the $ST_{throttled}$ CPI stacks (sampling), and assume that the measured CPI stack is representative for the next quanta. Because programs exhibit varying phase behavior, it requires to resample at periodic intervals to capture this phase behavior. Sampling $ST_{throttled}$ execution incurs performance overhead, because it has to temporarily stop other threads to obtain the $ST_{throttled}$ CPI stacks, and it can also be inaccurate if the program exhibits fine-grained phase behavior. Moreover, this approach is not easily applicable since the model uses the isolated performance in the SMT modes, which will require to execute the nop-microbenchmark during sampling periods.

Instead, we propose to estimate the $ST_{throttled}$ CPI stacks during SMT execution, similar to the cycle accounting technique presented by Eyerman and Eeckhout [7]. However, this technique requires hardware support that is not available in current processors. To obtain the $ST_{throttled}$ CPI stacks during SMT execution on an existing processor, we propose to measure the SMT CPI stacks and 'invert' the model: estimating $ST_{throttled}$ CPI stacks from SMT CPI stacks. Once these estimations are obtained, the scheduler applies the 'forward' model (i.e., the model described in the previous sections) on the estimated $ST_{throttled}$ CPI stacks per application to estimate the potential slowdown for thread-to-core mappings that are different from the current one. By continuously rebuilding the $ST_{throttled}$ CPI stacks from the current SMT CPI stacks, the scheduler can detect phase changes and adapt its schedule to improve performance. Note that, the proposed approach does not require any sampling phase and thus, it does not incur any sampling overhead.

Inverting the model is not as trivial as it sounds. The 'forward' model calculates the normalized SMT CPI stacks from the normalized $ST_{throttled}$ CPI stacks. As stated in Section 3.1, both stacks are normalized to the *single-threaded* CPI. However, without profiling, the $ST_{throttled}$ CPI is unknown in SMT mode, which means that the SMT components normalized to the $ST_{throttled}$ CPI (B', R'_i and M'_j in Equations 1 to 3) cannot be calculated. Nevertheless, we can calculate the SMT CPI components normalized to the *multi-threaded* CPI (see Figure 2b). By definition, the sum of these

2. The interference models used by the symbiotic scheduler are built with the data collected from all benchmarks. However, leave-p-out cross-validation is performed to evaluate their accuracy. See Section 6.1 for further details.

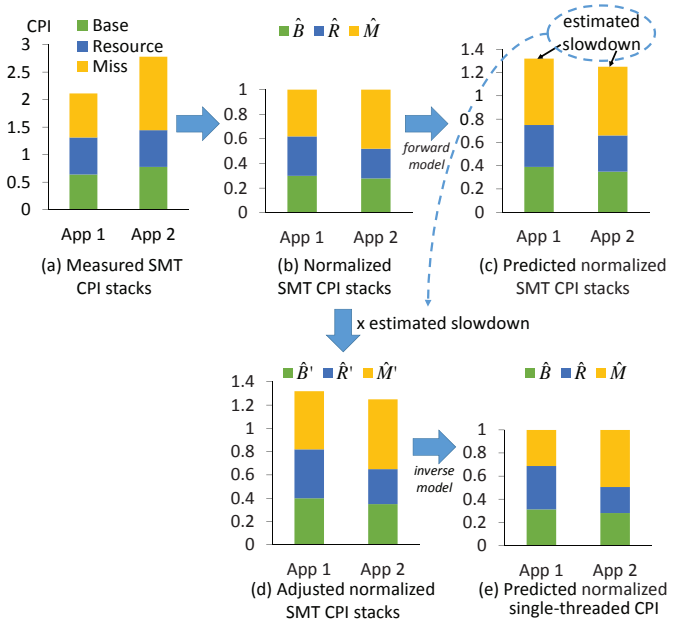


Fig. 2. Estimating the single-threaded CPI stacks from the SMT CPI stacks. First, SMT CPI stacks (a) are normalized to the SMT CPI (b); next, the forward model is applied to get an estimate of the slowdown due to interference (c); then the SMT CPI stacks are adjusted using the estimated slowdown to obtain a more accurate normalized SMT CPI stacks (d); lastly, the inverse model is applied to obtain the normalized single-threaded CPI stacks (e).

components equals one, which means that they are inaccurate estimates for the SMT components normalized to the $ST_{throttled}$ CPI, because the latter add to the actual slowdown, which is higher than one (see the last graph in Figure 1).

Because we do not know the $ST_{throttled}$ CPI, the model cannot be inverted in a mathematically rigorous way, which means we have to use an approximate approach. We observe that the SMT components normalized to SMT CPI are a rough estimate for the $ST_{throttled}$ components normalized to the $ST_{throttled}$ CPI (B, R_i and M_j), for two reasons. First, both normalized CPI stacks add to one. Second, if all the components experience the same relative increase between the $ST_{throttled}$ and SMT executions (e.g., all components are multiplied by 1.3), then the SMT CPI stack normalized to the SMT CPI would be exactly the same as the $ST_{throttled}$ stack normalized to the $ST_{throttled}$ CPI. Obviously, this is usually not the case, but intuitively, if a $ST_{throttled}$ stack has a relatively large component, it is expected that this component will also be large in the SMT stack, so the relative fraction should be similar.

Therefore, a first-order estimation of the $ST_{throttled}$ CPI stack is to take the SMT CPI stack normalized to the SMT CPI (see Figure 2b). The resulting $ST_{throttled}$ CPI stack component estimations are however not accurate enough to be used in the scheduler. Nonetheless, by applying the 'forward' model to these first-order single-threaded CPI stack estimations (see Figure 2c), a good initial estimation of the slowdown each application has experienced in SMT mode can be provided. This slowdown estimation can be used to renormalize the *measured* SMT CPI stacks by multiplying them with the estimated slowdown (see Figure 2d). This gives new, more accurate estimates for the SMT CPI stacks normalized to the $ST_{throttled}$ CPI (B', R_i' and M_j').

Next, we mathematically invert the model to obtain new estimates for the $ST_{throttled}$ CPI stacks (see Figure 2e). The mathematical inversion involves solving a set of equations. For two threads, we have two equations per component (one for each of the two threads), which both contain the two unknown single-threaded components, so a set of two equations with two unknowns must be solved (similar to four threads: four equations with four unknowns). Due to the multiplication of the single-threaded components in the δ term, the solution for two threads is in the form of the solution of a quadratic equation. For four threads, the inversion cannot be done analytically. We therefore decide to set δ to zero and train the model omitting this component of the model equation, which simplifies the formulas. This does not lead to a significant decrease in accuracy. The sum of the resulting estimates for the single-threaded normalized components (B, R_i and M_j) usually does not exactly equal one. Thus, the estimation can be further improved by renormalizing them to their sum.

4 SMT Interference-Aware Scheduler

In this section, we describe the implementation of the symbiotic scheduler that uses the interference model to improve the throughput of the processor. The goal of our proposed scheduler is to divide n applications over c (homogeneous) cores, with $n > c$, in order to optimize the overall throughput. Each core supports at least $\lceil \frac{n}{c} \rceil$ thread contexts using SMT. Note that we do not consider the problem of selecting n applications out of a larger set of runnable applications, we assume that this selection has already been made or that the number of runnable applications is smaller than or equal to the number of available thread contexts. As described in Section 5, we implement our scheduler as a user-level scheduler in Linux, and evaluate its performance on an IBM POWER8 machine. The scheduler implementation involves several steps which we discuss in the next sections.

4.1 Reduction of the cycle stack components

The most detailed cycle stack that the PMU (Performance Monitoring Unit) of the IBM POWER8 can provide involves the measurement of 45 events. However, the PMU only implements six thread-level counters. Four of these counters are programmable, and the remaining two measure the number of completed instructions and non-idle cycles. Furthermore, most of the events have structural conflicts with other events and cannot be measured together. As a result, 19 time slices or quanta are required to obtain the full cycle stack. Requiring 19 time slices to update the full cycle stack means that, at the time the last components are updated, other components contain old data (from up to 18 quanta ago). Since the scheduler uses 100ms quanta, this issue would make it less reactive to phase changes in the best scenario, and completely meaningless in the worst case.

An interesting characteristic of the CPI breakdown model is that is built up hierarchically, starting from a top level consisting of 5 components, and multiple lower levels where each component is split up into several more detailed components [1]. For example, the completion stall event of the first level, which measures the completion stalls caused by different resources, is split in several sub-events in the second level, which measure, among others, the completion stalls due to the fixed-point unit, the vector-scalar unit and the load-store unit. To improve the responsiveness of the scheduler and to reduce the complexity of calculating the model, we measure only the events that form the top level of the cycle breakdown model. This reduces the number of time slices to measure the model inputs to only two. The measured events

Counter	Explanation
PM_GRP_CMPL	Cycles where this thread committed instructions. *This is the base component in our model.*
PM_CMPLU_STALL	Cycles where a thread could not commit instructions because they were not finished. *This counter includes functional unit stalls, as well as data cache misses.*
PM_GCT_NOSLOT_CYC	Cycles where there are no instructions in the ROB for this thread, due to instruction cache misses or branch mispredictions.
PM_CMPLU_STALL_THRD	Following a completion stall (PM_CMPLU_STALL), the thread could not commit instructions because the commit port was being used by another thread. *This is a commit port resource stall.*
PM_NTCG_ALL_FIN	Cycles in which all instructions in the group have finished but completion is still pending. *The events behind this counter are not clear in [1], but it is non-negligible for some applications.*

TABLE 1
Overview of the measured IBM POWER8 performance counters to collect cycle stacks.

are indicated in Table 1. Note that the PM_CMPLU_STALL covers both resource stalls and some of the miss events. Because the underlying model for both is essentially the same, this is not a problem. Although the accuracy of the model could be improved by splitting up this component, our scheduler showed worse performance because of having to predict job symbiosis with old data for many of the components.

4.2 Selection of the optimal schedule

The scheduler uses the measured CPI stacks and the model to schedule the applications among cores. To simplify the scheduling decision, we make the following assumptions:

- The interference in the resources shared by all cores (shared last-level cache, memory controllers, memory banks, etc.) is mainly determined by the characteristics of all applications running on the processor, and not so much by the way these applications are scheduled onto the cores. This observation is also made by Radojković et al. [21]. As a result, with a fixed set of runnable applications, scheduling has no impact on the inter-core interference and the scheduler should not take inter-core interference into account.
- The IBM POWER8 cores implement an issue queue divided in two symmetric halves. Some of the execution pipelines, such as the fixed-point, floating-point, vector, load and load-store pipelines are similarly split into two sets. In the SMT modes, the threads can only issue instructions to a single half of the issue queue [23]. Thus, two 4-application schedules such as ABCD and ACBD may reach different performance, since in the first case application A is sharing some of the execution pipelines with application B, and in the second case it shares these pipelines with application C. We have experimentally checked that the performance difference of these schedules is on average 0.9% across 50 application combinations. Therefore, we assume that they perform equally and they do not need to be evaluated individually.

Even with these simplifications, the number of possible schedules is usually too large to perform an exhaustive search. The number of schedules considering n applications and c cores equals $\frac{n!}{c!\left(\frac{n}{c}!\right)^c}$ (assuming n is a multiple of c). For scheduling 16 applications on 8 cores in SMT2 mode, there are already more than 2 million possible schedules. To efficiently cope with the large number of possible schedules, we use a technique proposed by Jiang et al. [13]. The technique models the scheduling problem for two applications per core as a minimum-weight perfect matching problem, which can be solved in polynomial time using the blossom algorithm [5].

When scheduling for higher SMT modes (e.g., SMT4), the number of possible combinations becomes prohibitive for even a relatively low number of cores. For example, to schedule 20 applications on 5 cores in SMT4, there are more than 2 billion possible combinations. In addition, the scheduling problem for more than two applications per core cannot be modeled as a minimum-weight perfect matching problem. In fact, Jiang et al. also prove that this problem becomes NP-complete as soon as $\frac{n}{c} > 2$.

To address this issue, we use the hierarchical technique also proposed by Jiang et al. [13]. Using this approach, the applications are first divided into pairs, and these pairs are then combined to quadruples, using the blossom algorithm at both levels. Next, a local optimization step rearranges applications in each pair of quadruples to obtain better performance. While this technique is not guaranteed to give the optimal solution, Jiang et al. [13] show it to perform well in a setup where applications need to be scheduled in a clustered architecture, where each cluster shares a cache.

In summary, the scheduler does the following steps at the beginning of each time slice to schedule the application in SMT4 mode. To schedule applications in SMT2 mode, steps 4 and 5 are not done.

1) Collect the SMT CPI stacks for all applications over the previous time slice.
2) Use the inverted model to get an estimate of the $ST_{throttled}$ CPI stacks for each application.
3) Use the SMT2 forward model to predict the performance of each 2-application combination, and use the blossom algorithm to find the optimal schedule.
4) Use the SMT4 forward model to predict the performance of each 4-application combination, combining the pairs of applications selected in the previous step, and use the blossom algorithm to find a close to optimal schedule.
5) Apply the local optimization to each pair of 4-application combinations selected in the previous step to further improve the selected schedule.
6) Run the best schedule for the next time slice.

4.3 Scheduler implementation

Normally, workload execution and scheduler work are performed in a serial way. In other words, the applications do not run while the process selection is being performed. However, depending on the number of possible schedules that need to be evaluated, this serialization could cause a considerable overhead. For instance, scheduling applications in SMT2 mode incurs a negligible overhead, which is clearly compensated by the speedup reached by selecting the optimal schedules. In contrast, when scheduling four or more applications per core, the explosion in the number of possible schedules can easily cause that the benefits achieved by running better schedules end up being canceled out by the overhead of evaluating these schedules.

To avoid this overhead, we let applications run in parallel while the scheduler evaluates the possible schedules and selects the one that will be executed in the next quantum. In order to avoid the

workload to slow down this scheduling step, we choose to devote one of the cores exclusively to it. This design decision implies that while the scheduler evaluates and selects the schedule for the next quantum, the number of runnable applications (n) is higher than the number of available cores ($c-1$). During this period, we let Linux perform the task scheduling. As soon as the schedule for the next quantum is determined, the applications are allocated on the cores accordingly and executed using the c cores. The (lower) throughput achieved during the fraction of the workload execution where the applications run on $c-1$ cores is included in the performance results presented for the symbiotic scheduler.

5 Experimental Setup

We perform all experiments on an IBM Power System S812L machine, which is a POWER8 machine consisting of 10 cores. Each core can execute up to 8 hardware threads simultaneously. A core can be in single-threaded mode, SMT2 mode, SMT4 mode or SMT8 mode. Mode transitions are done automatically, depending on the number of active threads. We focus our experimental evaluation in SMT2 and SMT4 modes with multiprogram SPEC CPU 2006 workloads. We do not evaluate the SMT8 mode since we did not notice performance benefits with the Linux scheduler in SMT8 mode over SMT4 mode. On average across 10 32-application workloads to be run on 4 cores, the Linux scheduler performs slightly better (0.9%) in SMT8 mode compared to SMT4 mode. As the number of cores grows, the performance benefits in SMT8 mode are reduced and turn into performance losses. Thereby, on average across 10 80-application workloads ran with 10 cores, the Linux scheduler performs 7.8% worse in SMT8 mode than in SMT4 mode. This behavior should be related with the fact that SPEC benchmarks aim to stress the processor and the memory subsystem. Thus, the SMT8 mode should provide performance benefits, when running multithreaded scale-out applications that share a considerable amount of code and present a small memory footprint. Our setup uses an Ubuntu 14.04 Linux distribution with kernel 3.16.0.

We use all of the SPEC CPU 2006 benchmarks that we were able to compile for the POWER8 to evaluate our scheduler (21 out of 29). We run all benchmarks with the reference input set. For each benchmark, we measure the number of instructions required to run during 120 seconds in isolated execution and save it as the target number of instructions for the benchmark. This reduces the amount of variation in the benchmark execution times during the experiments. For the multiprogram experiments, we run until the last application completes its target number of instructions. When the applications reach their target number of instructions, their IPC and turnaround time are saved and the application is relaunched. This method ensures that we compare the same part of the execution of each application, and that the workload is uniform during the full experiment.

The time taken to collect the required inputs to train the model (see Section 3.2) is approximately 12 hours for the SMT2 mode and almost 50 hours for the SMT4 mode. However, training the model is a one-time offline step, and then the model can be used to schedule any workload whenever its characteristics resemble those of the training applications. The training phase is negligible compared to the model lifetime, and therefore it has not been included as a performance overhead in the next sections.

Our target metric is total system throughput (STP), which we measure by means of the weighted speedup metric [6]. More precisely, we measure the time each application requires to execute its target number of instructions in the multiprogram experiment and then divide the isolated time (120 seconds) by the multiprogram time, adding this number over all applications. To provide a more solid performance evaluation, we also evaluate the average normalized turnaround time (ANTT) [6] of the workloads, which is calculated as the arithmetic average across the normalized turnaround time of the applications, and corresponds with the reciprocal of the harmonic mean. Unlike STP, which is a system-oriented metric, ANTT provides insight into the per-application performance reached by each scheduler.

We evaluate 160 workloads overall. 80 workloads are devised to evaluate the SMT2 mode and their number of applications double the number of cores considered. Thus, they range from 8 (4-core workloads) to 20 (10-core workloads) applications. The remaining 80 workloads aim to evaluate the SMT4 mode, and include four application per core, thus ranging from 16 (4-core workloads) to 40 (10-core workloads) applications.

5.1 NUMA effects on the IBM POWER8

Our IBM POWER8 system has 32 GB of RAM memory on a single memory module. This apparently small fact presents strong implications. The IBM POWER8 processor is implemented as a dual-chip module (DCM) processor but it works as a single chip processor [3]. More precisely it is built by mounting two chips (chiplets) containing half the number of cores each. Both chiplets are interconnected by fast local SMP links and each one implements a memory controller that governs four DRAM slots. Figure 3 shows a block diagram of the processor core and memory subsystem. This design implies that our system includes 2 non-uniform memory access (NUMA) nodes. The first node is comprised of processor cores 0 to 4, while the second node contains the remaining five cores. Since the system only includes a single memory module connected to one of the NUMA nodes, it is expected that cores in this node have a non-negligible memory performance difference compared to cores in the other node. To confirm this behavior, we use the LMbench [17] and STREAM [16] benchmarks and measure the DRAM latency and bandwidth, respectively. These applications aim to stress the memory subsystem by accessing the elements of data arrays whose size increases up to 1792MB.

Figure 4 presents the memory latency that the cores of each NUMA node experience for each tested array size. Memory requests access the array in 128-byte strides, which matches the POWER8 cache line size. We did not appreciate any latency differences between cores in the same NUMA node. The latency is

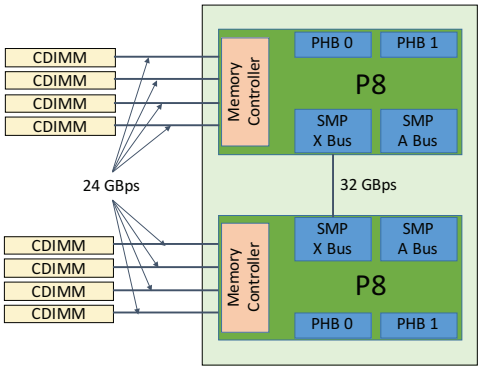


Fig. 3. Logical diagram of the POWER8 and memory subsystem.

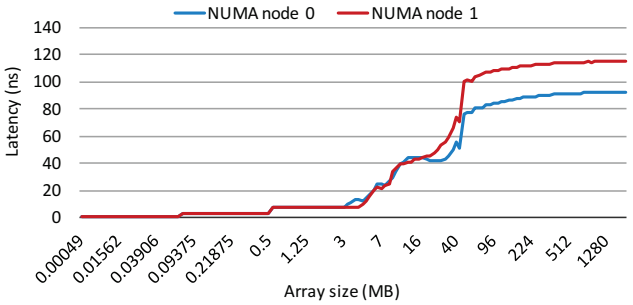


Fig. 4. Memory latency varying the array size.

Function	Kernel	NUMA node 0	NUMA node 1
Copy	$a(i) = b(i)$	17.7 GB/s	16.8 GB/s
Scale	$a(i) = q \times b(i)$	17.3 GB/s	16.5 GB/s
Add	$a(i) = b(i) + c(i)$	24.3 GB/s	22.5 GB/s
Triad	$a(i) = b(i) + q \times c(i)$	24.3 GB/s	22.4 GB/s

TABLE 2
Bandwidth reported by the STREAM benchmark for the two NUMA nodes.

identical for both NUMA nodes when the array fits in the L1, L2 or L3 caches. However, when the array exceeds the cache size and the main memory is accessed, the cores on the first NUMA node, where the memory slot is plugged in, achieve a latency around 20% lower over the cores on the second NUMA node.

Regarding memory bandwidth differences between NUMA nodes, Table 2 presents the average bandwidth achieved by cores in both nodes when running the STREAM benchmark. The results show that the cores on the first NUMA node achieve between 5.1% and 8.5% more memory bandwidth, depending on the executed kernel. It is also worth noting that the cores on the first NUMA node almost reach the theoretical maximum memory bandwidth, which is 24GB/s.

The Linux OS is aware of the system being a NUMA system. For instance, the *lscpu* command identifies two NUMA nodes, the first one including logical CPUs from 0 to 39, and the second one including logical CPUs from 40 to 79[3]. Since the kernel version 3.8, the Linux scheduler is able to perform NUMA-aware scheduling [4] and allocates each application to the NUMA node closest to the main memory where most of the application data resides. In our system, this NUMA node is always the first node (node 0), since it is the only one with a memory module installed. This scheduling behavior turns into performance improvements that must be taken into account in the experimental evaluation.

6 Experimental Evaluation

We now evaluate how well the scheduler performs compared to the default scheduler and prior work. Before showing the scheduler results, we first evaluate the accuracy of the interference prediction models devised for the SMT2 and SMT4 modes. Then, the system throughput, the per-application performance, and the stability of the selected coschedules are analyzed for the SMT2 and SMT4 modes.

6.1 Model accuracy

To study the accuracy of the models, we analyze the error deviation of the predicted CPI stacks with respect to the measured CPI stacks. The evaluation needs to be done in two steps since it is not possible to measure both the $ST_{throttled}$ and SMT CPI stacks together in the same quantum. In a preliminary step, we measure the per-quantum $ST_{throttled}$ CPI stacks of the applications off-line, keeping them in a profile with their instruction counts. These $ST_{throttled}$ CPI stacks will be used to check the model accuracy. Next, we run the combinations of applications. The SMT CPI stacks of the applications when running the different combinations are predicted before each quantum starts from their profiled $ST_{throttled}$ CPI stacks. When the quantum expires, the predicted SMT CPI stacks for the schedule are compared against the measured SMT CPI stacks. As done in the model construction, the $ST_{throttled}$ CPI stacks of consecutive quanta are interpolated, if needed, to ensure that the profiled $ST_{throttled}$ CPI stacks closely match the same instructions as the SMT CPI stacks. We explore all possible combinations of applications in SMT2 mode and a very large set of combinations in SMT4 mode, considering multiple time slices per combination to capture the phase behavior.

We use the leave-p-out cross-validation methodology to evaluate the accuracy of the proposed interference models. More precisely, leave-two-out cross validation and leave-four-out cross validation are used to measure the error of the SMT2 and SMT4 models, respectively. For each possible pair of applications, leave-two-out cross validation builds a model using the data from the remaining 19 applications, and then evaluates the model error when predicting the SMT CPI stacks for the pair of applications left out to build the model. The average absolute error and error histograms are obtained combining the errors measured for each pair of applications with the model built leaving them out. The same steps are performed to evaluate the SMT4 model, but leaving out 4-application combinations. Notice that in this case, the training data set is significantly reduced with respect to the model built using all applications.

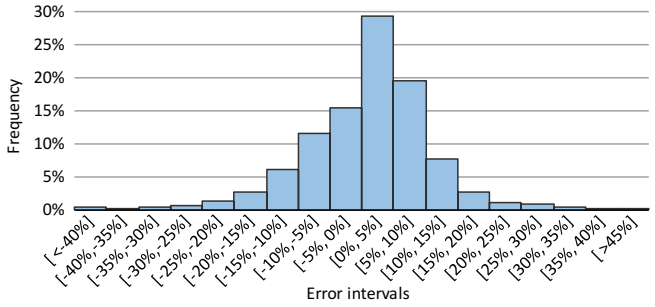
Regression models accuracy. Figure 5 shows the histograms of the errors of the interference prediction models (the 'forward' model) for the SMT2 (Figure 5(a)) and SMT4 (Figure 5(b)) modes, respectively. It shows the deviation committed when predicting the per-application slowdown from the $ST_{throttled}$ CPI stacks of the applications to be co-run.

Since there are fewer applications interfering with each other on SMT2 schedules than on SMT4 schedules, it is to be expected that the SMT2 interference model is more accurate than the SMT4 model. On average, the deviation is by 7.6% and 11.5% for the SMT2 and SMT4 models, respectively. Note that the for the SMT2 model, 45% of the deviations is within [−5%,5%] (29% for the SMT4 model). Remark that the models proposed in this paper are more accurate than our previously proposed models [10], because the presented approach considers the exact amount of private resources each hardware context has available in each SMT mode.

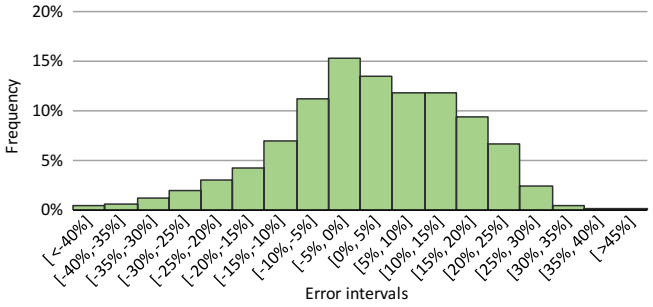
Inverse models accuracy. The inverse models estimate the $ST_{throttled}$ CPI stacks from the SMT CPI stacks of the applications when running concurrently on a schedule. By definition, the $ST_{throttled}$ CPI stacks add to one. Since the last step of our model inversion approach is a normalization, the predicted stacks will also add to one. Thus, the accuracy of the inverse models cannot be measured by comparing the CPI stacks as a sum of their components.

Figure 6 shows the distribution of the error for the inverse models obtained when predicting the *completion stalls* component for the SMT2 (Figure 6(a)) and SMT4 (Figure 6(b)) modes.

3. Each core of the POWER8 accounts for 8 logical CPUs in Linux. Logical CPUs 0 to 7 identify the 8 threads that can be run in core 0 with SMT8 mode, logical CPUs 8-15 identify those threads of core 1, and so on.

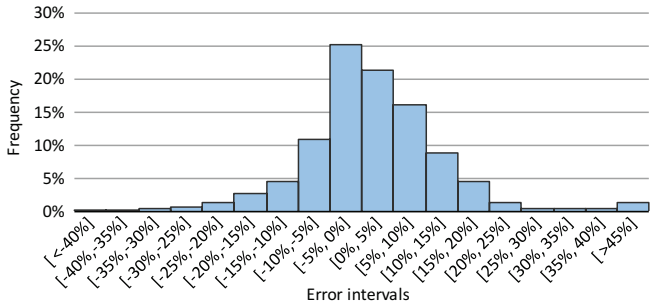


(a) SMT2 mode

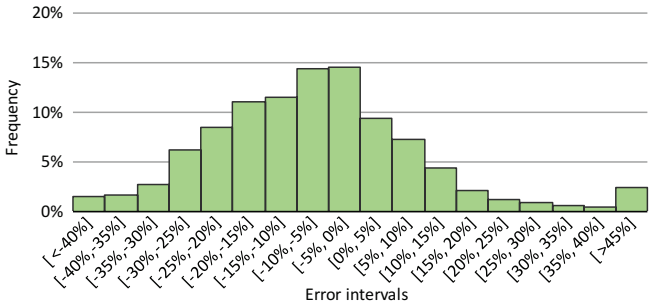


(b) SMT4 mode

Fig. 5. Forward model error distribution.



(a) SMT2 mode



(b) SMT4 mode

Fig. 6. Inverse model error distribution.

Completion stalls is the largest component and clearly dominates the CPI stack. It presents the highest average absolute error, which makes it a good estimate to evaluate the accuracy of the inverse models. The average absolute errors for the completion stalls component are 9.3% and 15.1%, in SMT2 and SMT4 modes, respectively. Notice that these average absolute error values do not highly differ from that obtained with the forward model. Finally, the frequency where the errors fall in the range $[-5\%,5\%]$ also reaches similar frequencies to that of the forward model, being 47% and 24%, respectively, in SMT2 and SMT4 modes.

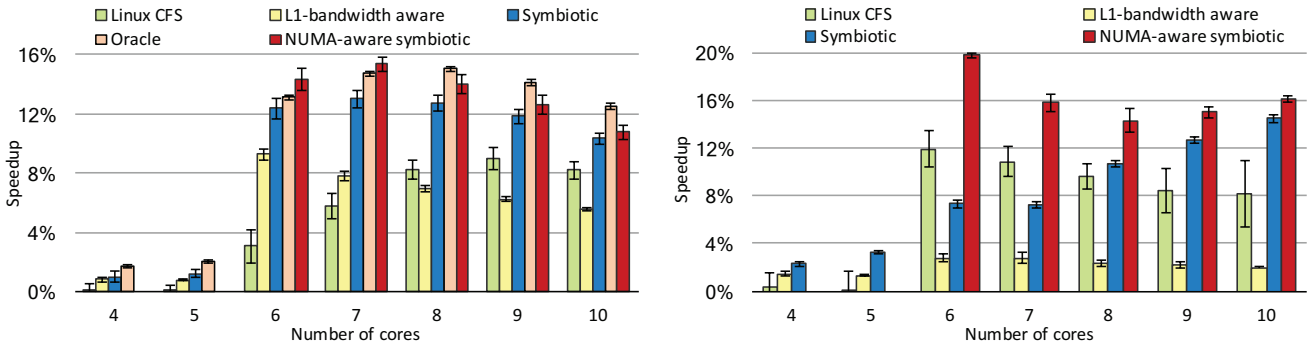
6.2 Scheduler performance

Now that we have shown that the interference prediction models are accurate, we evaluate the performance of our proposed scheduler that uses the models to obtain better schedules. We also analyze the impact of symbiotic scheduling on per-application performance and study the stability of the selected coschedules.

To evaluate the results achieved with the symbiotic job scheduler, we compare five different schedulers:

1) Random scheduler: applications are randomly distributed across the cores and contexts. Each time slice, a new schedule is randomly determined.
2) Linux CFS scheduler: the default Completely Fair Scheduler (CFS) in Linux. As discussed in Section 5.1, the CFS scheduler incorporates different patches to perform NUMA-aware scheduling. Thus, Linux is aware of the NUMA effects on the IBM POWER8 system and schedules the applications accordingly.
3) L1-bandwidth aware scheduler [11]: this scheduler is the most recent and closest prior work to our scheduler. It balances the L1 bandwidth requirements of the applications across the cores. It also executes on unmodified hardware and we implement it as a user-level scheduler in Linux. It was originally designed to run in SMT2 mode, but we extend it to support the SMT4 mode.
4) Symbiotic job scheduler: our baseline proposal. This approach considers the processor as a UMA (uniform memory architecture) system and after selecting the schedules they are allocated on the cores in increasing core order.
5) NUMA-aware symbiotic job scheduler. It works as the symbiotic job scheduler, but the selected schedules are allocated on the cores considering the main memory bandwidth utilization. This is done by measuring the main memory requests of the applications at runtime using performance counters, and allocating the schedules with higher memory bandwidth utilization on the cores of the NUMA node 0, the one connected to the memory controller that has installed DRAM modules in our machine (see Section 5.1).
6) Oracle scheduler: this scheduler uses an off-line measured profile with the slowdowns of the applications for the isolated run of each possible couple. However, it is not the most optimal scheduler: it is not NUMA-aware and, in addition, some applications can progress slower during the workload execution than during the profile runs, shifting their execution from the profile. Building a profile covering all possible situations is too costly. For the same reason, this scheduler is only evaluated in SMT2 mode.

The aim of the NUMA-aware optimization is not to perform sophisticated NUMA-aware scheduling, but to provide a fair comparison with the (NUMA-aware) Linux CFS scheduler. For this purpose, the basis of both NUMA-aware schedulers is the same: measure the memory accesses of the applications and allocate the most memory-intensive applications on the NUMA node where the physical memory is installed. Without the NUMA optimization, the benefits of selecting better schedules (the purpose of the symbiotic scheduler) can be hidden in case memory-



(a) SMT2 mode

(b) SMT4 mode

Fig. 7. Average system throughput increase of the symbiotic scheduler, NUMA-aware symbiotic scheduler, Linux CFS scheduler, L1-bandwidth aware scheduler, and oracle scheduler (only in SMT2 mode), relative to the random scheduler.

intensive applications were allocated on cores of the NUMA node more distant to the DRAM module. This issue can become an important limitation of the symbiotic scheduler over the Linux CFS scheduler, as experimental results will show.

6.3 System throughput

Figure 7 presents the system throughput increase achieved by the proposed symbiotic and NUMA-aware symbiotic schedulers, the Linux CFS scheduler, the L1-bandwidth aware scheduler, and the oracle scheduler (only in SMT2 mode) over the random scheduler, when running in SMT2 (Figure 7(a)) and SMT4 (Figure 7(b)) modes. The speedups are averaged per core count, ranging from 4 to 10 cores. For each core count and scheduler, the bars represent the average speedup for a set of ten different workloads that are run 15 times, plotting 95% confidence intervals. As mentioned in Section 5, the number of applications of the SMT2 and SMT4 workloads doubles and quadruples, respectively, the number of cores considered in the experiment.

The results include the negligible overhead incurred by the symbiotic schedulers, mainly the time needed to gather the event counts from the performance counters and update the scheduling variables. As explained in Section 4.3, the applications are kept running while the schedules for the next quantum are being obtained, which allows the scheduler to avoid the process selection overhead. In addition, the new models are more accurate and the scheduler does no longer require correction factors [10], further preventing the overhead of estimating single-thread performance used to calculate them.

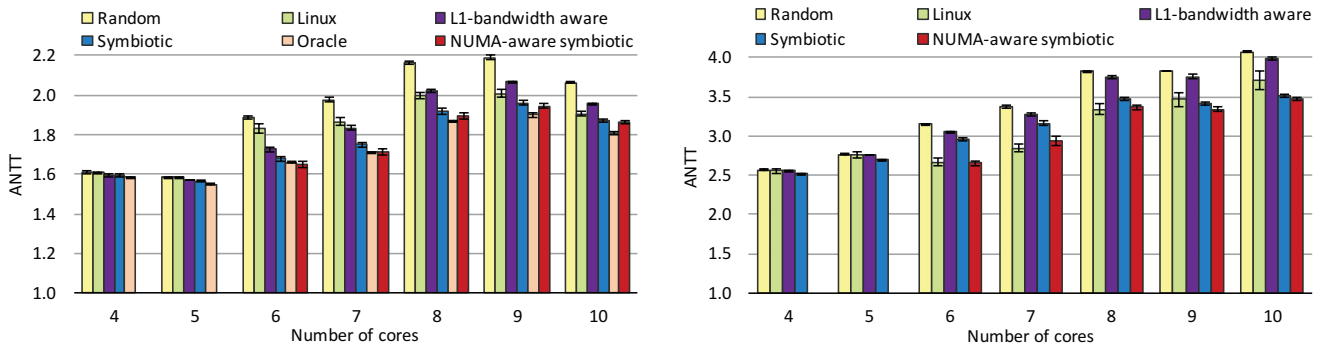
SMT2 mode. The symbiotic job scheduler and its NUMA-aware version distinctly outperform all other schedulers, with system throughout increases that are particularly high when the workloads run on six or more cores. On average across all core counts and workloads, the symbiotic scheduler performs 8.9% better than the random scheduler, 4.0% better than the default Linux CFS scheduler, and 3.4% better than the L1-bandwidth aware scheduler. These average performance improvements, however, are limited by the slight performance differences for the small workloads. For instance, on the 7-core workloads, the system throughput increase of the symbiotic scheduler over the random and Linux CFS schedulers are as high as 13.1% and 7.4%, respectively.

By taking into account the main memory accesses performed by each application to deal with the NUMA effects on our experimental platform, the NUMA-aware symbiotic scheduler improves the performance achieved by the symbiotic scheduler. On average, across workloads devised for 6 to 10 cores (the ones where the NUMA-effects appear), it performs 13.5% better than the random scheduler, 6.7% better than the Linux CFS scheduler, 5.9% better than the L1-bandwidth aware scheduler, and 1.3% better than the symbiotic scheduler. With respect to the Linux CFS scheduler, it achieves a maximum average performance benefit of 11.0% on 6-core workloads. The comparison of the NUMA-aware symbiotic scheduler against the Linux CFS scheduler is the best one to highlight the performance benefits provided by the symbiotic scheduling since both schedulers implement similar NUMA-aware optimizations.

Regarding the Linux CFS scheduler, its performance benefit presents an ascendant trend with the number of cores, but get somehow stabilized above 8 cores. The L1-bandwidth aware scheduler follows the opposite trend, and its performance benefits tend to decrease with the number of cores. These behaviors are clearly related with how they perform the scheduling. On one side, the Linux CFS scheduler monitors memory behavior and tries to reduce memory contention, which is more beneficial when there are more applications and therefore more possible contention. In addition, it also performs NUMA-aware scheduling and tries to allocate the applications with higher memory requirements on the cores of the NUMA node 0. In some cases, the Linux CFS scheduler even decides to pause threads, especially on the cores belonging to the NUMA node 1 (the farthest from the main memory modules) and when there are a lot of memory-intensive applications. On the other side, the L1-bandwidth aware scheduler deals with L1-bandwidth contention which plays a more important role when the number of applications is lower and main memory contention is not the main performance bottleneck. Anyway, both schedulers clearly perform worse than our proposed symbiotic schedulers.

Finally, we also compare the symbiotic scheduler with an oracle scheduler. This comparison helps estimating the performance losses of the symbiotic scheduler due to errors of the SMT interference model. Experimental results show small performance differences between both schedulers. On average across all the evaluated workloads, the oracle scheduler only performs 1.2% better than the symbiotic scheduler. The highest differences, around 2.5% on average, are observed for 9-core workloads.

SMT4 mode. The NUMA-aware symbiotic scheduler clearly outperforms all other schedulers across all thread counts. Considering the workloads devised to run on at least six cores, the first scenario where the NUMA effects emerge, the NUMA-aware symbiotic scheduler performs, on average, 16.2% better than the



(a) SMT2 mode

(b) SMT4 mode

Fig. 8. Average ANTT achieved by the symbiotic scheduler, NUMA-aware symbiotic scheduler, oracle scheduler (only in SMT2 mode), L1-bandwidth aware scheduler, Linux CFS scheduler, and random scheduler. ANTT is a lower is better metric.

random scheduler, 5.9% better than the Linux CFS scheduler, and 5.3% better than the symbiotic scheduler. In general the achieved speedup grows with the number of cores, as so does the speedup of the symbiotic scheduler, since there is higher interference and more difference between the best and worst schedules. However, being NUMA-aware extraordinarily enhances the throughput for 6- and 7-core workloads, which somehow breaks the trend. The reasons that explains the big improvements for these workloads, is that with only one or two cores belonging to the NUMA node 1, the node with lower memory performance, a NUMA-aware scheduler is able to allocate all memory intensive applications on the cores of the NUMA node 0, the node with higher memory performance. However, with workloads for higher number of cores, more cores from the NUMA node 1 are considered, and not all the memory intensive applications can be allocated on the NUMA node 0. Notice that same behavior is observed for the Linux CFS scheduler. An interesting observation is that being NUMA aware has a stronger impact on the performance of the SMT4 mode. This effect can be related with several issues. For instance, sharing the ROB with four threads can increase the penalty of a long-latency memory access. In addition, SMT4 workloads include more applications and thus, demand more memory bandwidth than SMT2 workloads.

Regarding the symbiotic scheduler it is really interesting to observe that its system throughput increase uniformly grows for all core counts from 4 to 10 cores. It performs better than the Linux CFS scheduler in all core counts, except for 6- and 7-core workloads, where as we have explained before its NUMA unawareness affects its performance. Meanwhile, the system throughput increase for the Linux CFS scheduler follows a decreasing trend where the number of cores considered grows from 6 to 10. This is another indicative of the fact that being NUMA-aware is critical for 6- and 7-core workloads, but reduces its importance as more cores are considered in the experiments. Finally, the L1-bandwidth aware scheduler achieves the lowest performance benefits, and only improves the Linux CFS scheduler on the smallest workloads, although its performance for higher core counts may well be constrained by not being NUMA-aware. Its low speedup also shows that L1 bandwidth is not one of the main contention points in SMT4 since other critical microarchitectural resources that strongly affect per-thread performance like the L1 cache space, the physical registers, and the number of ROB entries are reduced (per thread) as the number of supported threads increases.

6.4 Per-application performance

Although the main goal of the proposed symbiotic scheduling is to maximize the system throughput, we also evaluate its impact on the average normalized turnaround time (ANTT) metric. ANTT is essentially a measure of the average per-application performance, but since the harmonic mean tends to be lower when there is much variance, it also incorporates a notion of fairness [6].

Figure 8 depicts the ANTT achieved by the random scheduler, the Linux CFS scheduler, the L1-bandwidth aware scheduler, the oracle scheduler (only in SMT2 mode), the symbiotic scheduler, and the NUMA-aware symbiotic scheduler, when running the evaluated workloads for the SMT2 (Figure 8(a)) and SMT4 (Figure 8(b)) modes. The bars represent the average ANTT of the different schedulers across the ten workloads evaluated for each number of cores, plotting 95% confidence intervals.

SMT2 results. Figure 8(a) shows that the symbiotic scheduler and its NUMA-aware version clearly reach the lowest ANTT values across all evaluated workloads, followed by the L1-bandwidth aware and Linux CFS schedulers. The symbiotic schedulers reach the highest differences over the random and Linux CFS schedulers when the number of considered cores ranges from 6 to 8. For instance, the NUMA-aware symbiotic scheduler achieves an ANTT 8.6% lower than the Linux CFS scheduler across these workloads. Between both symbiotic schedulers, the NUMA-aware version achieves the lowest ANTT across all core counts. The results show that, as a side effect, by reducing interference as much as possible to maximize system throughput, the symbiotic schedulers also reduce the average normalized turnaround time of the applications.

Regarding the Linux CFS, L1-bandwidth aware, and oracle schedulers, the same trends observed on the system throughput appear with the ANTT metric. The L1-bandwidth aware scheduler reaches lower ANTT than the Linux CFS scheduler on workloads up to 7 cores, and it improves the ANTT over the L1-bandwidth aware scheduler for larger workloads. As explained before, this is due to the fact that the Linux CFS scheduler addresses memory bandwidth contention, which grows with the number of cores, and the L1-bandwidth aware scheduler deals with L1-bandwidth contention, which is more important for lower core counts. Finally, the oracle scheduler mainly reduces the ANTT over the symbiotic scheduler in the workloads with higher core counts.

SMT4 results. Figure 8(b) shows that the NUMA-aware symbiotic scheduler is the scheduler that achieves the best per-application performance, according to the ANTT metric. The performance benefit is high over the random scheduler, specially

as the workloads run on a higher number of cores. For instance, on the 10-core workloads, the NUMA-aware symbiotic scheduler achieves an ANTT 14.7% lower than the random scheduler. The difference with the Linux CFS scheduler is negligible except on the 9-core and 10-core workloads, where the NUMA-aware symbiotic scheduler is 3.7% and 6.6% better, respectively. Regarding the symbiotic scheduler, it reaches high ANTT on workloads from 6 to 8 cores (only surpassed by the random scheduler), which as discussed before is due to not being aware of the NUMA effects on the POWER8, which particularly affects the workloads for these numbers of cores.

6.5 Symbiosis patterns

The symbiotic scheduler constantly re-evaluates the optimal schedule, which means that it adapts to phase behavior, updating the combinations of applications that are run together. If there is no phase change behavior, a static schedule would suffice, avoiding the overhead of recalculating the schedules. Figure 9 and Figure 10 present the frequency matrices of the schedules selected by the symbiotic job scheduler for two 5-core workloads in SMT2 mode and a 5-core workload in SMT4 mode, respectively. The frequency matrices are symmetric matrices that represent the percentage of quanta where each combination of applications is scheduled on one core. The darker the color of the cell, the more frequently the associated pair of applications runs together on the same core.

SMT2 mode. The two matrices of Figure 9 represent two distinct behaviors that we have observed in the symbiotic scheduling runs. The frequency matrix of workload 5_1 shows a workload where two couples are scheduled very frequently (*h264ref* is coscheduled with *libquantum* and *milc* with *bwaves*, in 66% and 70% of the time slices, respectively). This high frequency suggests that the applications present high symbiosis (e.g., a memory-bound application with a cpu-bound application) and a constant phase behavior. A different behavior is observed in the matrix of workload 5_2, where there is not a predominant pair of applications that is usually coscheduled, but all the applications are coscheduled with multiple corunners. This pattern occurs when the applications present phase behavior that changes the symbiosis of the applications, which makes it important to adapt the coschedule to the current phase.

SMT4 mode. The frequency matrix of Figure 10 also shows the application behaviors described for the two SMT2 frequency matrices. For instance, from the group of applications *cactusADM* (two times), *h264ref*, *sjeng*, and *gobmk*, four of them are usually scheduled together on the same SMT core. Other group of jobs formed by applications *leslie3d*, *libquantum* (two times), and *gcc* also tends to be scheduled on the same SMT core. This behavior

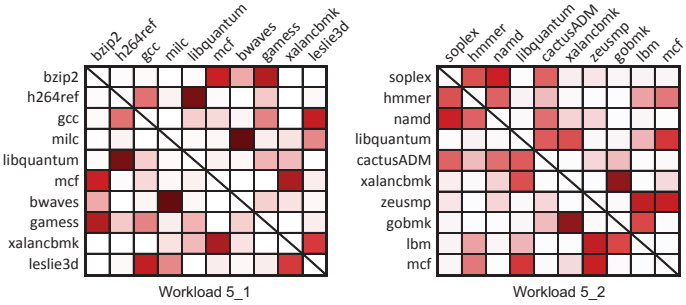


Fig. 9. Frequency matrices for two 5-core workloads running in SMT2 mode.

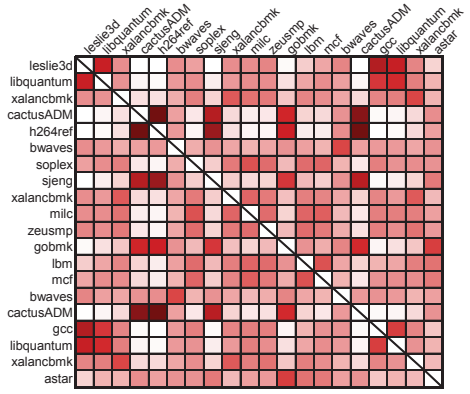


Fig. 10. Frequency matrix for a 5-core workload running in SMT4 mode.

is expected for applications that exhibit low phase behavior and high symbiosis among them. The other applications, either do not present high symbiosis with any application of the workload or they show a high phase behavior that makes them run on schedules with several applications through their execution.

7 Conclusions and Future Work

Scheduling has a considerable impact on highly threaded processors because of the interference between threads in shared resources. We propose a novel symbiotic job scheduler for a multicore processor consisting of multi-threaded (SMT) cores. The scheduler uses a model based on cycle component stacks, that is used to estimate the symbiosis between applications at runtime without sampling. Experiments on an IBM POWER8 server show that a NUMA-aware version of the symbiotic scheduler improves throughput, on average across 6- to 10-core workloads, by 6.7% and 5.9% over Linux in SMT2 and SMT4 modes, respectively. Due to the use of an analytical model, the overhead of our scheduler is negligible.

Although our current implementation is designed for the IBM POWER8, our scheduler could be adapted to other CMP architectures with SMT cores that provide a similar cycle accounting mechanism, e.g., an Intel Xeon server [19]. This only requires a one-time training step. The scheduler can also support heterogeneous architectures, by creating different models for the various core types.

Finally, the symbiotic scheduler proposed in this work is focused on single-threaded applications. Scheduling for parallel applications requires from distinct strategies tailored to their specific characteristics. The design of such a scheduler is left as future work.

Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under grants TIN2015-66972-C5-1-R and TIN2014-62246-EXP, as well as by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement No. 259295.

References

[1] IBM Knowledge Center, Analyzing Application Performance on Power Systems servers, 2015.

[2] J. Burns and J.-L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(2):142–155, 2002.

[3] A. Caldeira, V. Haug, M. Kahle, C. Maciel, M. Sanchez, and S. Sung. IBM Power Systems S812L and S822L Technical Overview and Introduction. *IBM Redbooks*, 2014.

[4] J. Corbet. NUMA Scheduling Progress. *Link: https://lwn.net/Articles/568870/*, 2013.

[5] J. Edmonds. Maximum Matching and a Polyhedron with 0, L-Vertices. *J. Res. Nat. Bur. Standards B*, 69(1965):125–130, 1965.

[6] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[7] S. Eyerman and L. Eeckhout. Per-Thread Cycle Accounting in SMT Processors. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–144, Mar. 2009.

[8] S. Eyerman and L. Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–102, Mar. 2010.

[9] S. Eyerman and L. Eeckhout. The Benefit of SMT in the Multi-Core Era: Flexibility Towards Degrees of Thread-Level Parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 591–606, 2014.

[10] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. Symbiotic Job Scheduling on the IBM POWER8. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 669–680, 2016.

[11] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 123–132, 2013.

[12] S. Hily and A. Seznec. Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. 1997.

[13] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–229, 2008.

[14] Y. Li, K. Skadron, D. Brooks, and Z. Hu. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–82, 2005.

[15] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[16] J. McCalpin. STREAM Benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what*, 1995.

[17] L. W. McVoy, C. Staelin, et al. lmbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294, 1996.

[18] T. Moseley, J. Kihm, D. Connors, and D. Grunwald. Methods for Modeling Resource Contention on Simultaneous Multithreading Processors. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 373–380, Oct 2005.

[19] A. Nowak, D. Levinthal, and W. Zwaenepoel. Hierarchical Cycle Accounting: a New Method for Application Performance Tuning. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 112–123, March 2015.

[20] L. Porter, M. A. Laurenzano, A. Tiwari, A. Jundt, W. A. Ward, Jr., R. Campbell, and L. Carrington. Making the Most of SMT in HPC: System- and Application-Level Perspectives. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):59:1–59:26, Jan. 2015.

[21] P. Radojkovic, V. Cakarevic, J. Verdu, A. Pajuelo, F. Cazorla, M. Nemirovsky, and M. Valero. Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2513–2525, Dec 2013.

[22] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, 2004.

[23] B. Sinharoy, J. Van Norstrand, R. Eickemeyer, H. Le, J. Leenstra, D. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. Fernsler. IBM POWER8 Processor Core Microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, Jan 2015.

[24] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Nov. 2000.

[25] D. M. Tullsen and J. A. Brown. Handling Long-Latency Loads in a Simultaneous Multithreading Processor. In *International Symposium on Microarchitecture (MICRO)*, pages 318–327, 2001.

[26] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[27] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *International Symposium on Microarchitecture (MICRO)*, pages 406–418, 2014.

Josué Feliu received his MSc and PhD degrees in computer engineering from the Universitat Politècnica de València, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher at the Department of Computer Engineering of the same university. His research interests include scheduling strategies and performance modeling for multicore and multithreaded processors.

Stijn Eyerman received the MSc and PhD degrees from Ghent University, in 2004 and 2008, respectively. He is currently working as a research scientist with Intel. He has published more than 40 papers at conferences and journals, two of which have been awarded with an IEEE Micro Top Picks selection. His interests include processor performance modeling and scheduling on (heterogeneous) multicore processors.

Julio Sahuquillo received his MSc, and PhD degrees in computer engineering from the Universidad Politécnica de Valencia (Spain). Since 2002 he is an Associate Professor at the Department of Computer Engineering. He has published more than 100 refereed conference and journal papers. His current research topics include multi- and manycore processors, memory hierarchy design, cache coherence, and power dissipation. He has co-chaired several workshops related with these topics, collocated in conjunction with IEEE supported conferences.

Salvador Petit received his PhD degree in computer engineering from the Universitat Politècnica de València, Spain. Currently, he is an associate professor in the Department of Computer Engineering at UPV where he teaches several courses on computer organization. His research topics include multithreaded and multicore processors, memory hierarchy design, as well as real-time systems.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University, in 2002. He is a professor with Ghent University, Belgium. His research interests are in the area of computer architecture, with a specific interest in performance analysis, evaluation, and modeling. He is the current editor-in-chief of the IEEE Micro. His research is funded by the European Research Council under the European Communitys 7th Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, as well as by the European Commission under the 7th Framework Programme, Grant Agreement no. 610490.