# Symbiotic Job Scheduling on the IBM POWER8

Josué Feliu[1], Stijn Eyerman[2], Julio Sahuquillo[1], and Salvador Petit[1]

[1] Dept. of Computer Engineering (DISCA), Universitat Politècnica de València, València, Spain
[2] Dept. of Electronics and Information Systems (ELIS), Ghent University, Ghent, Belgium
Email: jofepre@fiv.upv.es, stijn.eyerman@elis.ugent.be, {jsahuqui,spetit}@disca.upv.es

## ABSTRACT

Simultaneous multithreading (SMT) processors share most of the microarchitectural core components among the co-running applications. The competition for shared resources causes performance interference between applications. Therefore, the performance benefits of SMT processors heavily depend on the complementarity of the co-running applications. Symbiotic job scheduling, i.e., scheduling applications that co-run well together on a core, can have a considerable impact on the performance of a processor with SMT cores. Prior work uses sampling or novel hardware support to perform symbiotic job scheduling, which has either a non-negligible overhead or is impossible to use on existing hardware.

This paper proposes a symbiotic job scheduler for the IBM POWER8 processor. We leverage the existing cycle accounting mechanism to predict symbiosis between applications, and use that information at run-time to decide which applications should run on the same core or on separate cores. We implement the scheduler in the Linux operating system and evaluate it on an IBM POWER8 server running multiprogrammed workloads. The symbiotic job scheduler significantly improves performance compared to both an agnostic random scheduler and the default Linux scheduler. With respect to Linux, it achieves an average speedup by 8.8% for workloads comprising 12 applications, and by 4.7% on average across all evaluated workloads.

## 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 a 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.

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 the core count, the scheduler must decide which applications should run together on one core. Selecting the optimal schedule is an NP-hard problem [10], and predicting the performance of a schedule is a non-trivial task due to the high amount of sharing in an SMT core.

This paper presents a new scheduler for the IBM POWER8 [18] 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 CPI stacks.

Previous work on symbiotic job scheduling for SMT uses sampling to explore the space of possible schedules [19], relies on novel hardware support [6], or performs an offline analysis to predict the interference between applications on an SMT core [23]. In contrast, we perform online model-based scheduling, without sampling, on an existing commercial processor. To this end, we leverage the existing CPI stack accounting mechanism on the IBM POWER8 to build

978-1-4673-9211-2/16/31.00$ ©2016 IEEE

a model that predicts the interference among threads on an SMT core. Using this model, we can 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.

We make the following important contributions.

- We propose an online scheduler for a CMP consisting of SMT cores, without the need for sampling schedules, and without requiring additional hardware.
- We develop a comprehensive SMT interference model based on CPI stacks that takes into account contention in all shared resources: processor pipeline width, functional units, and cache and memory.
- We implement our scheduler in an existing Linux distribution and we evaluate it on an IBM POWER8 processor executing multiprogram workloads.

Our scheduler performs 10.3% better than a random scheduler, and 4.7% better than the default Linux scheduler across all evaluated workloads, consisting of 8 to 20 applications and evaluated on two-way SMT mode. The highest speedups with respect to Linux are achieved with workloads of 10 to 14 applications, where our scheduler reaches an average speedup of 7.0% over Linux. The overhead of using a performance model and exploring the possible schedules is negligible. As our scheduler is completely software-based, it can be shipped without changes in future Linux distributions targeted at the IBM POWER8. Furthermore, with appropriate training of the model, the scheduler can also be used for other architectures.

## 2. RELATED WORK

Simultaneous multithreading (SMT) was proposed by Tullsen et al. [22] 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, 11]), mainly due to replicating architectural and performance-critical structures, but it can significantly improve throughput. Recently, Eyerman and Eeckhout [7] show that a multicore processor consisting of SMT cores has an additional benefit other than increasing throughput. SMT is flexible when the 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, while 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 [9]. Snavely and Tullsen [19] 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 [6] 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 are generated using complex new hardware, 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. [13] use regression on performance counter measurements to estimate the speedup of SMT when coexecuting two applications. Porter et al. [15] estimate the speedup of a multithreaded application when enabling SMT, based on performance counter events and machine learning. Settle et al. [17] predict job symbiosis using offline profiled cache activity maps. Feliu et al. [8] propose to balance L1 cache bandwidth requirements across the cores in order to reduce interference and improve throughput. Zhang et al. [23] propose a methodology to predict the interference among threads on an SMT core. They developed "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 resources is measured. By combining resource usage and sensitivity to contention, the 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 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 [6], 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 at 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 at the right. Eyerman and Eeckhout [6] 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 delay because the fetch unit is shared. However, Eyerman and Eeckhout use novel hardware support [5] 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 et al. for SMT cores [5], which means that the model in [6] cannot be used as it is. Some of the components relate to each other to some extent (e.g., the number of cycles instructions are dispatched in [5] 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 it is accounted for in the other mechanism, and vice versa. For example, following [5], 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 in [5], while for the POWER 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 [6], 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 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

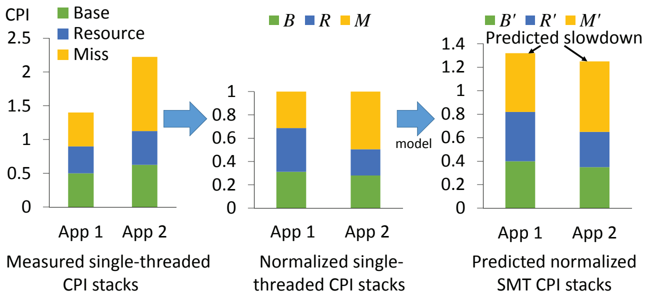

**Figure 1: Overview of the model: first, measured CPI stacks are normalized to obtain probabilities; then, the model predicts the increase of the components and the resulting slowdown (1.32 for App 1 and 1.25 for App 2).**

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 ($B'$, $R'_i$, $M'_j$).

*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 (two for the POWER8) 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.

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

The parameters $\alpha_B$ through $\delta_B$ are determined using regression, see Section 3.2. $\alpha_B$ 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 $\beta_B$ term reflects the fact that the original ST base component of a thread remains in SMT execution. It would be intuitive to set $\beta_B$ 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 $\beta_B$ is larger than 1. $\gamma_B$ is the impact of the sum of the base components of the other threads. $\delta_B$ 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. [6] (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.

*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. In 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 $j$):

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

Similar to the base component model, $\alpha$ indicates a constant offset that is added due to SMT execution (e.g., extra latency). $\beta$ indicates the fraction of the single-threaded component that remains in SMT mode, while the term with $\gamma$ 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.

*Miss components.*

Miss components are caused by instruction and data cache misses in 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 accessing 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. [21], 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:

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

Although the model looks exactly the same, the underlying reasoning is slightly different. $\alpha$ again relates to fixed SMT effects (e.g., cache latency increase). The $\beta$ term is the original miss component of that thread, while the $\gamma$ term indicates that an application can get extra misses due to interference caused by misses of the other applications. We also add a $\delta$ 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.

*Parameter values.*

Table 1 shows the parameter values for the some of the most common components after performing regression (see Section 3.2): the base component, fixed point (integer) resource stalls, floating point resource stalls and data cache misses. The parameters are for 2 SMT threads (SMT2). For most components, the highest weights are for the $\beta$ and $\delta$ terms, indicating that in general, a given SMT component equals some fraction of the corresponding ST component plus the interaction with ST components of the co-running threads ($\delta$).

For the base component, the interference terms ($\gamma$ and $\delta$) are zero, meaning that there is no interference with the co-running threads. Instead, the base component increases with a constant 1.7 factor. To explain this, we need to look at how the POWER8 commits instructions [18]. Instructions are not committed individually, but in groups of consecutive instructions that are created at dispatch time ('dispatch groups'). A dispatch group can only be committed if all instructions in it have finished. In ST mode, the dispatch groups consist of up to 8 instructions, and only one group can be committed in a cycle. In SMT mode, groups consist of up to 4 instructions, and two groups can commit per cycle. As a result, there is

| Component | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ |
|---|---|---|---|---|
| Base | -0.019 | 1.698 | 0 | 0 |
| Fixed point | 0.009 | 1.089 | 0 | -0.859 |
| Floating point | 0.008 | 1.219 | 0 | 0.308 |
| Data cache miss | 0.031 | 1.173 | 0.072 | 0.615 |

**Table 1: Model parameters for some of the most common components.**

no interference in SMT2 mode, because each thread is able to commit instructions every cycle. The increase in the base component is mainly caused by the reduction of the group size from 8 to 4, almost doubling the number of dispatch groups. The base component is not exactly doubled, because in ST mode, not all groups contain 8 instructions, because of instruction cache misses, taken branches, or structural limitations (e.g., a group can contain only up to two branches). Note that although we did not explicitly incorporate this behavior into the model, the regression was able to accurately capture this.

The fixed point resource stall component has a negative $\delta$. We find that this component regularly *decreases* from ST to SMT mode. Fixed point resource stalls mostly occur when there are no other stalls and the core is continuously executing instructions. If there are other stall events, the core is regularly halted and the short fixed point resource stalls are 'hidden' under these larger stalls. During SMT execution, interference causes longer stall times for other resources and miss events, hiding more fixed point stalls than in ST mode. This causes the fixed point component to reduce. $\delta$ is negative, which means that this effect mainly occurs when all threads have a large fixed point stall component ($\delta$ reflects the interaction between the threads). In this case, all threads have a large fraction of time without large stalls, causing a high IPC, and a high utilization of the pipeline. When these threads are combined in SMT, there is a larger chance for pipeline conflicts (e.g, in the dispatch/execute/commit bandwidth), causing an increase of the other components, and a resulting decrease in the fixed point stall component.

## 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 in [6], 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. To train the model, we first run all benchmarks in isolation and collect CPI stacks per scheduler quantum (100 ms). We also keep track of the instruction count per quantum, to know what part of the program is executed in each quantum (we evaluate single-threaded programs with a fixed instruction count). We normalize each stack to its total CPI.

Next, we execute all 2-program combinations of the evaluated benchmarks on a single core in SMT2 mode (see Section 5 for the benchmarks we evaluate). We also collect per-thread CPI stacks and instruction counts for each quantum. Next, we normalize each SMT CPI stack to the CPI of executing the same instructions in *single-threaded mode*. We normalize to the ST CPI because we want to estimate the slowdown each application gets versus single-threaded execution, which equals the SMT CPI divided by the ST CPI (see the last graph in Figure 1). This is also in line with the methodology in [6]. The ST CPI is calculated using the ST cycle and instruction counts measured in a ST profiling step. Because the performance of an application differs between ST and SMT modes, and the quanta are fixed time periods, the instruction counts do not match between ST and SMT profiles. To solve this problem, we interpolate the ST CPI stacks between two quanta to ensure that ST and SMT CPI stacks are covering approximately the same instructions.

Using all these data (normalized CPI stacks per quantum for all applications in ST mode, and for all combinations of applications in SMT mode), we determine the model parameters. There is one set of ($\alpha$, $\beta$, $\gamma$, $\delta$) parameters per component. These parameters are tied to the component only, and not to specific applications. So, provided the training set is diverse enough, we do not need to retrain this model for new applications. Due to the large set of training examples and the limited set of parameters to fit, there is no danger of overfitting the model.

Once the model has been constructed, we can use it to estimate the SMT CPI stacks from the ST 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 at the right). This information is used to select combinations with minimal slowdown (see Section 4).

## 3.3 Obtaining ST CPI stacks in SMT mode

Up to now, we assumed that we have the ST CPI stacks available. This is not a practical assumption, it would require that we execute each application in single-threaded mode first, and keep all of the per-quantum CPI stacks in a profile. This is a large overhead for a realistic scheduler. Alternatively, we could periodically run each application in ST mode (sampling), and assume that the measured CPI stack is representative for the next quanta. Because programs exhibit varying phase behavior, we need to resample at periodic intervals to capture this phase behavior. Sampling ST execution involves a performance overhead, because we have to temporarily stop other threads to allow a thread to run in ST mode, and it can also be inaccurate if the program exhibits fine-grained phase behavior.

Instead, we propose to estimate the ST CPI stacks during SMT execution, similar to the cycle accounting technique described in [5]. However, the technique in [5] requires hardware support that is not available in current processors. To obtain the ST CPI stacks during SMT execution on an existing processor, we propose to measure the SMT CPI stacks and 'invert' the model: estimating ST 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 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 CPI stacks from the current SMT CPI stacks, the scheduler can detect phase changes and adapt its schedule to improve performance.

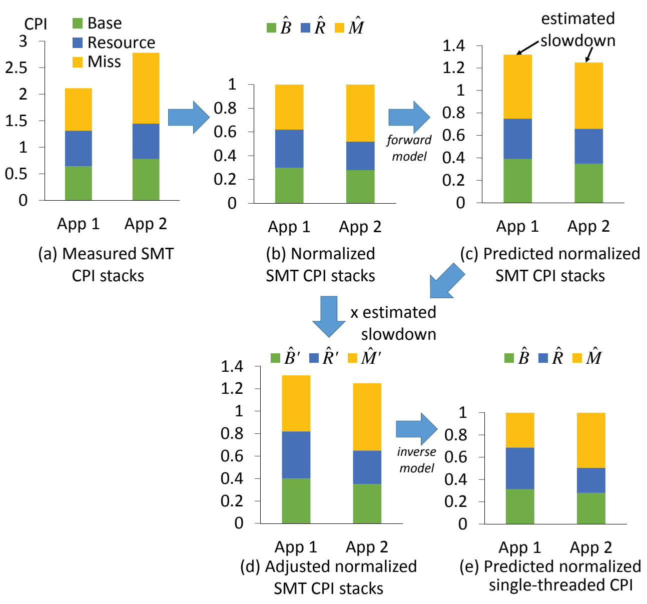

**Figure 2: Estimating the single-threaded CPI stacks from the SMT CPI stacks. First, SMT CPI stacks (a) are normalized to the SMT CPI (b); next, the forward model is applied to get an estimate of the slowdown due to interference (c); then the SMT CPI stacks are adjusted using the estimated slowdown 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).**

Inverting the model is not as trivial as it sounds. The 'forward' model calculates the normalized SMT CPI stacks from the normalized ST CPI stacks. As stated in Section 3.1, both stacks are normalized to the *single-threaded* CPI. However, without profiling, the ST CPI is unknown in SMT mode, which means that the SMT components normalized to the ST 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 components equals to one, which means that they are not good estimates for the SMT components normalized to the ST CPI, because the latter add to the actual slowdown number, which is higher than one (see the last graph in Figure 1).

Because we do not know the ST 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 components normalized to the ST CPI ($B$, $R_i$ and $M_j$), because of two reasons. First, both normalized CPI stacks add to one. Second, if all the components experience the same relative increase between the ST and SMT mode (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 stack normalized to the ST CPI. Obviously, this is usually not the case, but intuitively, if a ST stack has a relatively large component, it is expected that this component will also be a large in the SMT CPI stack, so the relative fraction should be similar.

Therefore, a first-order estimation of the ST CPI stack is to take the SMT CPI stack normalized to the SMT CPI (see Figure 2b). The resulting ST 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 CPI ($B'$, $R'_i$ and $M'_j$).

Next, we mathematically invert the model to obtain new estimates for the ST 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 $\delta$ term, the solution is in the form of the solution of a quadratic equation. The sum of the resulting estimates for the single-threaded normalized components ($B$, $R_i$ and $M_j$) usually does not exactly equals one. Thus, the estimation can be further improved by renormalizing them to their sum. We applied this technique to a large set of random numbers and found that it yields single-threaded component estimates that are within 1% to 2% of the correct values.

## 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 total 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 in Linux, and evaluate its performance on an IBM POWER8 machine. The scheduler implementation involves several steps which are discussed 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

| 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 2: Overview of the measured IBM POWER8 performance counters to collect cycle stacks.**

that, at the time the last components are updated, other components contain old data (from up to 18 quanta ago). This issue would make the scheduler less reactive to phase changes in the best scenario, and completely unmeaningful 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 are indicated in Table 2. 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 Correction factor

Because the model is developed based on insights, and because it is fit using a large range of benchmarks, the model is relatively accurate across a large set of applications, as we will show in Section 6. However, we find that the model is somewhat more inaccurate for particular applications or combinations of applications. This can have an impact on scheduling decisions, because applications or program combinations for which performance is overestimated (relative to the estimations of the other applications and combinations) could be selected more often than better combinations, whereas underestimations lead to a too small selection chance.

Another issue is that the model is constructed using single-threaded and SMT CPI stacks with only one active core. However, during scheduling, the other cores also execute applications. This causes interference in the inter-core shared resources (e.g., shared LLC, memory bandwidth), and thus an increase in some of the components. The model we developed partially captures this interference. For example, high memory bandwidth contention during a time slice could result in a higher number of stalls due to a data cache miss in the SMT execution. These extra stalls would be carried to the ST cycle stack of the application (by inverting the model), and then they would be taken into account to predict the symbiosis of the applications for the next quantum. Nonetheless, we notice that the model can be inaccurate when there is a substantial impact of off-core interference, degrading the quality of the selected schedules.

To solve the model's inaccuracy for some combinations and to factor in the impact of off-core interference, we use a dynamic correction factor. This mechanism records the performance of a schedule after it has been selected and executed during a time slice, and checks the accuracy of the prediction made during the selection. It then calculates a correction factor defined as the actual performance divided by the performance estimated by the model. For each application we keep one correction factor per possible co-runner, and we set the correction factor for unseen combinations to one (no error). When doing a prediction for the next quantum, we check the table of correction factors and we multiply the predicted performance with this correction factor. This way, we learn from previous observations and dynamically make the model more accurate.

We update the correction factor using an exponential moving average. We do not just keep the most recent value, because a small correction factor ($< 1$) at some point in time can prevent a combination from ever being selected again. Thus, if due to phase changes, the actual correction factor becomes closer to one, this is not detected and the combination is never considered during the whole execution. A moving average smooths out sudden changes, and makes the correction factor only big or small after a few subsequent under- or over-estimations. Occasionally, the applications also experience very low performance during a time slice, resulting in a very low factor. Even when using a moving average for the correction factor, such a low factor could prevent the combination from being selected again. To avoid this scenario, the correction factor is not updated when the factor for a combination is beneath 0.5. This situation occurs very occasionally and sparsely, i.e., the next time slice usually has a more 'normal' correction factor.

Calculating the correction factor requires knowledge of the isolated performance, because the model predicts the interference, i.e., the delay an application encounters by being coscheduled with other applications on one core. We can measure the performance of each application in the current schedule, but we do not know the isolated performance. The isolated performance could be determined by an offline run, but this introduces overhead and ignores phase behavior. Instead, we very sparsely execute each application in single-threaded mode on a core, and record its isolated performance. In addition, a profiling time much shorter than a normal time slice (20 ms instead of 100 ms) is used. Thus, the sampling time is very limited: it equals the number of SMT contexts times the profiling time, and it is done only every 200 time slices (accounting for 0.2% of the time for 2-way SMT). The performance overhead caused by this sampling is low, since we keep all the cores running one application in ST mode, achieving relatively good performance. Note that this sampling phase is different from SMT schedulers that use sampling [19]: we sample single-threaded execution, while they sample the set of possible schedules. We also use a much sparser sampling than used in per-thread cycle accounting proposals for SMT [12]. We find that the model has a certain bias for some applications, and once this bias is detected (i.e., the correction factor), we do not need fine-grained single-threaded execution times to obtain good accuracy.

### 4.3 Selection of the optimal schedule

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

- All cores are homogeneous, and thus have the same performance model. The scheduler could be extended for heterogeneous multicores, by building a model per core type.
- 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. [16]. 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.

Even with these simplifications, the number of possible schedules is usually too large to perform an exhaustive search, even with our fast models. The number of different schedules for dividing $n$ applications onto $c$ cores equals $\frac{n!}{c!\left(\frac{n}{c}!\right)^c}$ (assuming $n$ is a multiple of $c$). For dividing 16 applications on 8 cores, there are already more than 2 million possible schedules. Evaluating each of them would take too much time. Jiang et al. [10] prove this problem to be NP-complete as soon as $\frac{n}{c} > 2$. To efficiently cope with the large number of possible schedules, we use a technique proposed by Jiang et al. [10]. They model the scheduling problem as a minimum-weight perfect matching problem, which can be solved in polynomial time using the blossom algorithm [3].

In summary, the scheduler does the following at the beginning of each time slice:

1. Collect the SMT CPI stacks for all applications over the previous time slice.
2. Update the correction factor for the combinations that were executed the previous time slice.
3. Use the inverted model to get an estimate of the CPI stacks in isolated execution for each application.
4. Use the forward model and the correction factor to predict the performance of each combination, and use the blossom algorithm to find the best schedule.
5. Run the best schedule for the next time slice.

## 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. Because we evaluate our scheduler on multiprogram SPEC workloads, and the L1 cache of one core is limited to 64KB, we only evaluate our scheduler for SMT2. Having 4 or 8 SPEC benchmarks running on one core puts too high pressure on the L1 cache. The higher SMT modes are designed for multithreaded scale-out applications that share a considerable amount of code and have a low memory footprint. Our setup uses an Ubuntu 14.04 Linux distribution.

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 is 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. We measure total system throughput (STP) by means of the weighted speedup metric [4]. 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. We evaluate 105 workloads, ranging from 8-program combinations on 4 cores to 20-program combinations on 10 cores.

## 6. 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 model.

### 6.1 Model accuracy

*Regression model accuracy.*

Figure 3 shows a histogram of the errors of the interference prediction model (the 'forward' model). It shows the distribution of the error of predicting the per-application slowdown given the ST CPI stacks of the applications to be co-run. The results cover all possible combinations, and multiple time slices per combination to capture phase behavior. Few errors are larger than 30% (less than 3% of all

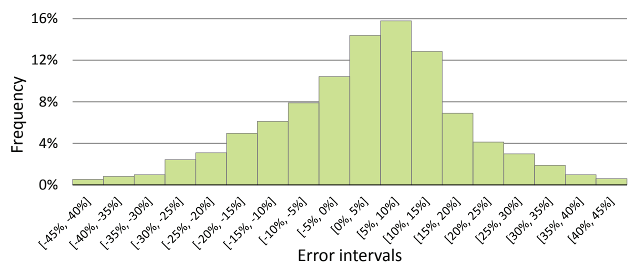

**Figure 3: Forward model error distribution histogram.**

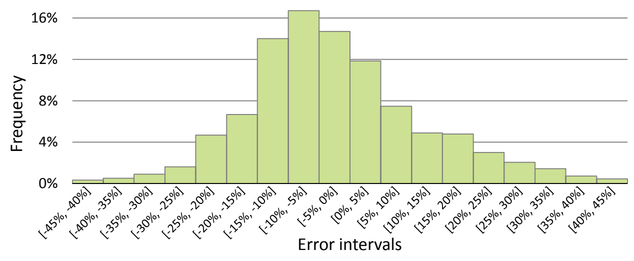

**Figure 4: Inverse model error histogram.**

points), but the majority of the errors are within 15%. The average absolute error is 12.3%.

*Inverse model accuracy.*
The inverse model estimates the ST CPI stacks from the SMT CPI stacks. Figure 4 shows the distribution of the error for the inverse model. The average absolute error is 13.4%, which is similar to the error of the forward model. The power of the inverse model is that it can estimate the ST CPI of an application during SMT execution, for which prior proposals require extra hardware [5] or extensive sampling [12]. Note that the extreme errors for both the forward and the inverse model are reduced by the dynamic correction factor (not included in these results).

## 6.2 Scheduler performance

Now that we have shown that the interference prediction model is relatively accurate, we evaluate the performance of our proposed scheduler that uses the model to obtain better schedules. We also analyze the impact of symbiotic scheduling on fairness, we quantify the overhead of the scheduler, and analyze the stability of the selected coschedules. Finally, we show the impact of exploiting slight differences between the cores.

### *6.2.1 Overall performance*

To analyze the performance benefits provided by the symbiotic job scheduler, we compare four different schedulers:

1. Random scheduler: Applications are randomly distributed across the cores. Every time slice, a new schedule is randomly determined.
2. Linux scheduler: the default Completely Fair Scheduler (CFS) in Linux.
3. L1-bandwidth aware scheduler [8]: 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 an alternative scheduler in Linux.
4. Symbiotic job scheduler: our proposal.

Figure 5 presents the speedups of the system throughput achieved by the Linux scheduler, the L1-bandwidth aware scheduler and the proposed symbiotic job scheduler relative to the random scheduler. Each set of bars represents an individual workload (i.e., combination of applications), and the workloads are grouped by the number of available cores. The workloads comprise twice as many applications as the number of available cores (e.g., the workloads evaluated with five cores are composed of ten applications). The results *include* the overhead of the schedulers, i.e., the time needed to gather the event counts from the performance counters and update the scheduling variables, and the time needed to take the scheduling decisions. For our symbiotic job scheduler, they also include the time consumed in the single-thread-per-core sampling phases. The speedups shown for each workload and scheduler represent the average speedup for a set of 15 runs, plotting 95% confidence intervals.

The symbiotic job scheduler clearly outperforms all other schedulers across all thread counts. Considering all the workloads, it performs on average 10.3% better than the random scheduler, 4.7% better than the default Linux scheduler, and 4.1% better than the L1-bandwidth aware scheduler. If we only consider the *middle* workloads (from 5-core workloads to 7-core workloads) the speedup over the Linux scheduler is even 7.0%. Compared to smaller workloads, the middle workloads offer a much higher number of possible combinations of applications, which increases the difference between the best and worst coschedules, allowing the symbiotic scheduler to find better coschedules. For large core counts, the overall performance is mainly determined by off-core interference (shared cache, memory controller), and less by selecting symbiotic job schedules. As we will explain below, the Linux scheduler is apparently good at controlling memory contention, which increases its performance compared to the random scheduler, decreasing the gap with our symbiotic scheduler.

The speedup of the Linux scheduler over the random scheduler tends to be higher for workloads with a higher number of cores (see also Figure 7, which shows the average speedup per core count). We surmise that the Linux scheduler somehow monitors memory behavior and tries to reduce memory contention, which is more beneficial when there are more applications and therefore more possible contention. This guess is further supported by the fact that the Linux scheduler sometimes decides to pause threads, especially on cores that seem to have a lower memory performance (see Section 6.2.5) and when there are a lot of memory-intensive applications. Our symbiotic scheduler also takes into account memory contention (through the miss component interference), and it does not put two memory-intensive applications on the same core. However, it also considers interference in the other components, which explains its superior performance for lower core counts.

The speedups achieved by the L1-bandwidth aware scheduler are similar to those of the Linux scheduler. Nevertheless, they tend to higher than the Linux speedups for low core counts. When the number of running applications is low,

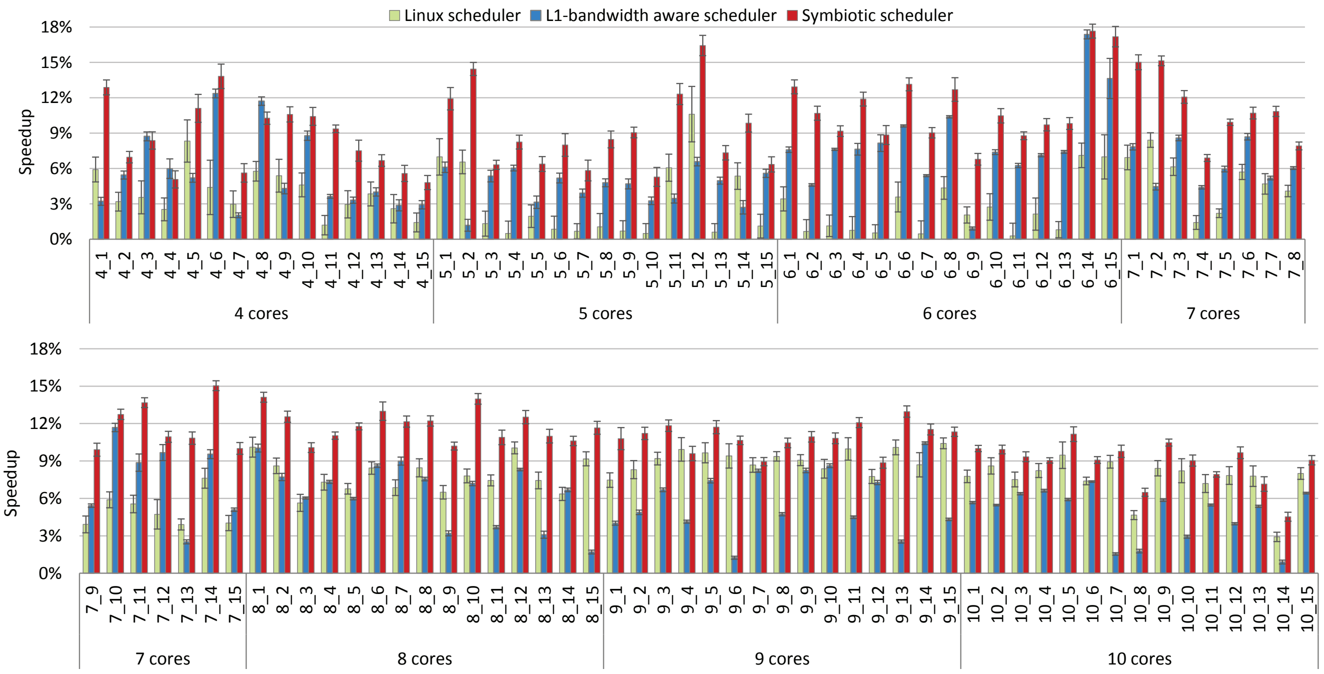

**Figure 5: Speedup of the Linux, L1-bandwidth aware and our symbiotic scheduler relative to a random scheduler.**

memory interference is less significant and L1 interference plays a more important role on the performance degradation. However, with higher number of applications, memory contention becomes the main bottleneck and the effectiveness of the L1-bandwidth aware scheduler is reduced. Although we faithfully implemented the L1-bandwidth aware scheduler as described in [8], we do not see as big performance improvements as in the original paper. We attribute this to the different hardware setup. For instance, the IBM POWER8 has double the L1 cache size of the Intel Xeon processor, which was used in their experiments. The larger cache should significantly reduce the pressure on this resource.

### 6.2.2 *Fairness*

Although the main goal of the job symbiosis scheduler is to maximize the system throughput, we also evaluate its impact on fairness. Fairness quantifies how evenly the performance benefits (or losses) are distributed across all the applications of a workload. Unfair schedules can lead to priority inversion or even starvation for a single application, even if they increase total throughput. We calculate the *un*fairness of a schedule as the maximum slowdown (compared to ST execution) divided by the minimum slowdown across all the applications of the workload. An unfairness equal to 1 means that the system is completely fair.

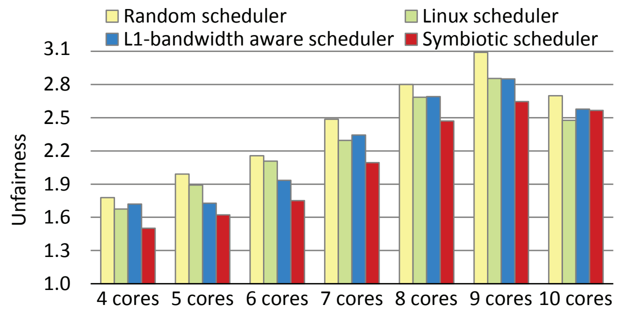

**Figure 6: Average unfairness (lower is better) of the symbiotic and Linux schedulers.**

Figure 6 depicts the unfairness achieved by the four evaluated schedulers. The different bars represent the average unfairness across all the fifteen workloads evaluated for each number of cores. The figure shows that the symbiotic scheduler reaches the lowest unfairness for workloads ranging from 4 to 9 cores, followed by the L1-bandwidth aware and Linux schedulers. As expected, the worst unfairness is reached by the random scheduler. With respect to Linux, the unfairness decrease of the symbiotic scheduler is quite significant. For instance, for 6-core workloads the symbiotic and Linux schedulers have an unfairness of 1.75 and 2.11, respectively, which means that Linux is 20% more unfair than our scheduler. Our scheduler tries to reduce interference as much as possible, and therefore, as a side effect, it helps reduce unfairness.

For 10 cores, unfairness reduces for all schedulers. Because the processor is designed for 10 cores, we guess that using all cores leads to the most balanced usage of the off-core resources (memory controllers, various buffers, etc.), resulting in better fairness. At this core count, our scheduler is slightly less fair than the Linux scheduler, supposedly because the Linux scheduler is better at controlling memory contention.

### 6.2.3 *Scheduler overhead*

Our scheduler has to perform a non-negligible amount of computations at the beginning of each time slice: it has to apply the inverse model to estimate the single-threaded cycle stacks, and then it has to search for the optimal schedule using the forward model. These computations are clearly

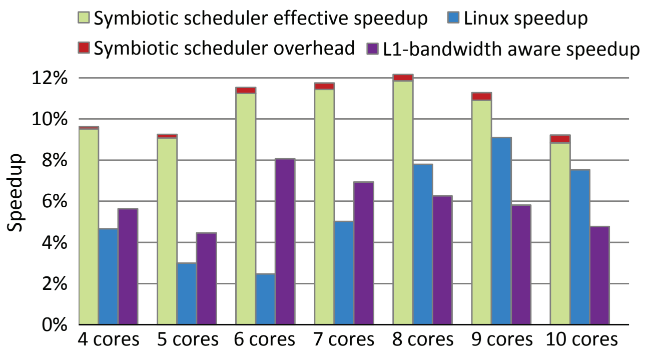

**Figure 7: Effective average speedup and overhead of the symbiotic scheduler, the Linux scheduler, and L1-bandwidth aware scheduler, relative to the random scheduler.**

more complex than those of the other schedulers, introducing more scheduling overhead. Figure 7 shows the speedup of the symbiotic scheduler over the random scheduler, *without* taking the overhead into account, i.e., we assume zero scheduling overhead (top of the stack). The top component of the stack shows by how much the speedup is reduced when considering the overhead, i.e., the bottom part is the speedup including the overhead (which is equal to the results shown in the previous section). The figure also shows the speedups of the Linux and L1-bandwidth aware schedulers, which have no noticeable overhead for the evaluated quantum length. The results are averaged over all evaluated workloads per core count. It shows that the overhead of the symbiotic scheduler is very small, and has only a marginal impact on the speedup. The overhead slightly increases with the core count, but scales relatively well due to the polynomial time complexity of the matching algorithm.

### 6.2.4 *Symbiosis patterns*

The symbiotic scheduler constantly re-evaluates the optimal schedule, which means that it adapts to phase behavior, updating the couples of applications that are run together. If there is no phase change behavior, a static schedule would suffice, avoiding the overhead of recalculating the schedule. Figure 8 presents a frequency matrix of the job coschedules for two different 5-core workloads. The symmetric matrix representsthe percentage of quanta where each combination of jobs is coscheduled on one core. The darker the color of the cell, the more frequently the associated pair of applications runs together on the same core.

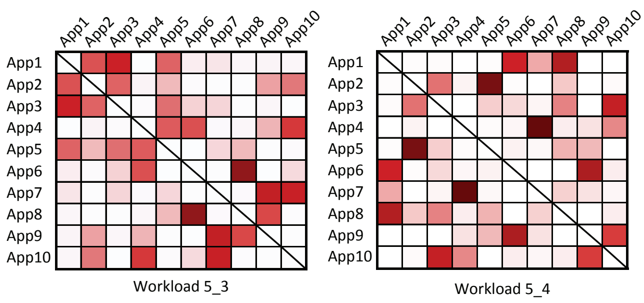

**Figure 8: Frequency matrices for two 5-core workloads.**

The two matrices represent two distinct behaviors that we have observed in the symbiotic scheduling runs. The frequency matrix of workload 5_4 (at the right) shows a workload where two couples are scheduled very frequently (App2 is coscheduled with App5 and App4 with App7, 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. The opposite behavior is observed in the matrix of workload 5_3. In this case, 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.

### 6.2.5 *Core asymmetry*

During our experiments, we notice that the characteristics of the cores on our POWER8 processor are not homogeneous. In particular, 4 applications (*mcf*, *milc*, *gemsFDTD*, and *lbm*) out of the 21 SPEC CPU 2006 benchmarks have a clearly higher performance on the first 5 cores (cores 0 to 4) than on the last 5 cores (cores 5 to 9). For the other applications, single-threaded performance is approximately equal across all cores. The applications with different performance are all memory-intensive, so we conjecture that the asymmetry is in the memory subsystem. We could not find any cause for this behavior in publications or in the processor documentation. We guess that the last 5 cores might be further away from the memory controller(s), incurring a longer memory latency, or that there is some (possibly unintentional) priority mechanism implemented in the centaurus chip [20] that acts as memory controller and handles the access sequence of the DRAM requests.

The Linux scheduler seems to be somehow aware of this asymmetry in the memory resource division (or at least, aware of the memory performance of the applications), as it more frequently pauses one or more threads on the last 5 cores when memory pressure is high. In contrast, our scheduler does not pause threads, but it still achieves high throughput by spreading the memory-intensive applications across cores. In addition, one could exploit this asymmetry to optimize performance even more by putting the most memory-intensive applications on the first 5 cores. Figure 9 shows the performance of our proposed scheduler with and without

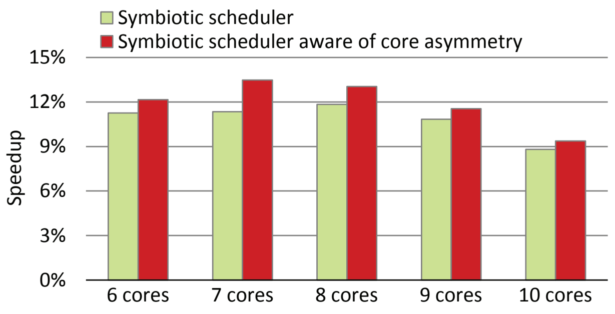

**Figure 9: Avg. speedup of the Symbiotic scheduler not aware and aware of asymmetry over a random scheduler.**

this machine-specific optimization. The performance indeed increases a little bit on average (about 1%), but the small difference shows that our scheduler is general enough that it does not need to rely on these machine-specific features to obtain good performance.

## 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, and it does not require extensive sampling. Experiments on an IBM POWER8 server show that our scheduler improves throughput by 11.0% and 8.8% versus the random and Linux kernel built-in schedulers, respectively, for workloads composed of 12 applications, and by 10.3% and 4.7%, respectively, across all evaluated workloads. 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 can be adapted to other CMP architectures with SMT cores that provide a similar cycle accounting mechanism, e.g., an Intel Xeon server [14]. This only requires a one-time training step. The scheduler can also support heterogeneous architectures, by creating different models for the various core types.

## 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.

## 8. 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] J. Edmonds. Maximum matching and a polyhedron with 0, l-vertices. *J. Res. Nat. Bur. Standards B*, 69(1965):125–130, 1965.

[4] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. 1997.

[10] 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.

[11] 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.

[12] C. Luque, M. Moreto, F. J. Cazorla, and M. Valero. Fair CPU time accounting in CMP+SMT processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):50, 2013.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] W. Starke, J. Stuecheli, D. Daly, J. Dodson, F. Auernhammer, P. Sagmeister, G. Guthrie, C. Marino, M. Siegel, and B. Blaner. The cache and memory subsystems of the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):3:1–3:13, Jan 2015.

[21] 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.

[22] 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.

[23] 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.