Thread Isolation to Improve Symbiotic Scheduling on SMT Multicore Processors

Josué Feliu, Julio Sahuquillo, *Member, IEEE*, Salvador Petit, *Member, IEEE*, and Lieven Eeckhout, *Fellow, IEEE*

Abstract—Resource sharing is a critical issue in simultaneous multithreading (SMT) processors as threads running simultaneously on an SMT core compete for shared resources. Symbiotic job scheduling, which co-schedules applications with complementary resource demands, is an effective solution to maximize hardware utilization and improve overall system performance. However, symbiotic job scheduling typically distributes threads evenly among cores, i.e., all cores get assigned the same number of threads, which we find to lead to sub-optimal performance. In this paper, we show that asymmetric schedules (i.e., schedules that assign a different number of threads to each SMT core) can significantly improve performance compared to symmetric schedules. To leverage this finding, we propose *thread isolation*, a technique that turns symmetric schedules into asymmetric ones yielding higher overall system performance. Thread isolation identifies SMT-adverse applications and schedules them in isolation on a dedicated core to mitigate their sharp performance degradation under SMT. Our experimental results on an IBM POWER8 processor show that thread isolation improves system throughput by up to 5.5 percent compared to a state-of-the-art symmetric symbiotic job scheduler.

Index Terms—Simultaneous multithreading (SMT), symbiotic job scheduling, thread isolation

1 Introduction

Simultaneous multithreading (SMT) processors improve hardware resource utilization and system throughput over single-threaded processors by co-executing distinct threads on the same core, i.e., instructions from different threads may execute in the same cycle [1]. In an SMT processor, most of the core execution resources, including the L1 caches, reorder buffer, issue queues, functional units, physical register file, etc. are shared among co-running threads. The number of core resources that need to be replicated is limited. Because of the dramatically improved system throughput at low additional hardware cost, major processor manufacturers such as Intel, AMD and IBM offer high-performance SMT processors as their trademark products.

Resource sharing is a key aspect of an SMT processor design since threads co-running on a core share resources at fine granularity. Basically, two main design approaches can be employed to share a resource among co-executing threads: static partitioning versus dynamic sharing. Partitioning splits a resource in as many fixed-size parts as there are supported threads. For instance, a 180-entry reorder buffer (ROB) can be split into two 90-entry ROBs to support two simultaneous threads, or in four 45-entry ROBs to support four threads. To preserve high single-thread performance, the entire ROB is still available when only a single thread is running. In contrast, dynamic sharing allows different threads to compete for and use a distinct portion of a shared structure according to their requirements and the management logic. For example, concurrent threads dynamically compete for L1 cache space. Note that some processor structures may be partitioned whereas others may be dynamically shared.

When running in SMT mode, resource sharing harms individual per-application performance in two ways. First, for partitioned resources, a thread can only use its assigned share. As a result, its performance will be inferior compared to running in isolation. Second, for the shared resources, threads interfere with each other when competing through dynamic sharing. In particular, resource sharing makes the performance of individual threads and, consequently, the throughput of an SMT core, strongly dependent on the characteristics of the co-executing threads. Co-scheduling applications that do not stress the same shared resources minimizes interference and maximizes SMT throughput.

For this reason, scheduling applications on SMT cores is key to achieving high overall system performance, i.e., determining which applications to co-run has a severe impact on performance. Symbiotic job scheduling or intelligently selecting which applications to co-run on an SMT core, has been widely explored, see for example [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Almost all previous work on job symbiosis propose symmetric schedules where the same number of applications is mapped to each core. Only the works by Gomaa et al. [5] and De Vuyst et al. [7] explore

- *J. Feliu, J. Sahuquillo, and S. Petit are with the Department of Computer Engineering, Universitat Politécnica de Valéncia, Valéncia 46022, Spain. E-mail: jofepre@gap.upv.es, {jsahuqui, spetit}@disca.upv.es.*
- *L. Eeckhout is with the Department of Electronics and Information Systems, Ghent University, Gent 9000, Belgium. E-mail: lieven.eeckhout@ugent.be.*

Manuscript received 5 Sept. 2018; revised 31 July 2019; accepted 8 Aug. 2019. Date of publication 14 Aug. 2019; date of current version 26 Dec. 2019. (Corresponding author: Josué Feliu.) Recommended for acceptance by C. Krintz. Digital Object Identifier no. 10.1109/TPDS.2019.2934955

asymmetric schedules but, unlike our work, the focus of their works is on thermal and energy efficiency aspects rather than performance. Gomaa et al. [5] use asymmetric temperature-aware schedules to favor cool cores and let hot cores cool down. De Vuyst et al. [7] devise asymmetric schedules that, combined with the ability to power down idle cores, provide significant improvements in energy-delay product (EDP) compared to symmetric schedules.

Unlike prior work, in this paper we propose asymmetric scheduling to improve SMT throughput. Our proposal is based on the finding that the performance gap between single-threaded (ST) mode, i.e., an application runs in isolation, versus SMT mode, i.e., an application co-runs with other applications, varies widely across applications. In other words, some applications are SMT-friendly whereas others are SMT-adverse, i.e., they see their performance significantly degrade under SMT execution. Considering all applications equally under symbiotic SMT scheduling therefore leads to suboptimal performance.

To leverage the previous finding, we propose *thread isolation*, a technique to improve the performance of symmetric schedules by turning them into asymmetric ones. Thread isolation works on top of a state-of-the-art symbiotic scheduler, which obtains the best symmetric schedule by mapping application pairs per core. Thread isolation then identifies the application that experiences a sharp performance degradation when running in SMT mode. Pairs of applications that include such an SMT-adverse application are broken down and the SMT-adverse application is scheduled to run in isolation on a dedicated core. The other application is added to a different pair, forming a 3-application combination that will run on another SMT core. This leads to an asymmetric schedule in which one application runs on a dedicated core in ST mode and the other three applications co-run in SMT mode on another core. These operations are driven by SMT interference models to ensure that the asymmetric schedule devised by thread isolation outperforms the original symmetric schedule.

The experimental evaluation, carried out on an IBM POWER8 processor, shows that thread isolation significantly improves system throughput compared to a symbiotic scheduler when the workload composition is favorable to asymmetric schedules. The maximum speedups for 6-application, 8-application, 10-application workloads amount to 5.5, 4.8, and 4.2 percent, respectively, compared to a state-of-the-art symbiotic scheduler that only devises symmetric schedules. When the workload is not suitable to asymmetric schedules, thread isolation is not applied and the achieved performance matches that of the symbiotic scheduler.

To sum up, we make the following contributions in this work:

- We analyze the performance of the SPEC CPU2006 applications when scheduled in sets of two and three applications on the same SMT core. The results shows that performance degradation widely varies depending on the application and SMT level. While SMT-adverse applications suffer a severe performance degradation, the performance of SMT-friendly applications reduces moderately. This observation stresses the interest in asymmetric scheduling.
- We study the performance of asymmetric schedules and find out that they can achieve higher system throughput than state-of-the-art symmetric symbiotic schedules. Asymmetric schedules run SMT-adverse applications in ST mode on a single core and allocate SMT-friendly applications together on other cores. The performance benefit obtained by the SMT-adverse applications exceeds the performance degradation suffered by the SMT-friendly ones.
- We propose *thread isolation*, an new scheduling approach that turns symmetric symbiotic schedules into asymmetric ones yielding higher overall system performance.

The rest of the paper is organized as follows. Section 2 analyzes the performance degradation when two and three applications are co-scheduled on an SMT core. Section 3 studies the potential for asymmetric scheduling and identifies the application that benefit more when running in isolation. Section 4 explains the proposed thread thread isolation algorithm. Section 5 describes the experimental setup. Section 6 presents the experimental evaluation of the proposal. Finally, Section 7 discusses related work and Section 8 presents concluding remarks.

2 Performance Characterization in SMT Execution

Before quantifying the impact of resource interference on SMT performance, we first revisit how SMT core hardware resources are shared among co-executing threads.

2.1 SMT Resource Sharing

Resource sharing is a critical design aspect of SMT processors. As introduced before, each resource can be implemented to be statically partitioned or dynamically shared when multiple threads run simultaneously. Partitioned resources are easier to implement and provide two interesting features: performance isolation and predictability. However, dynamically shared resources potentially provide higher system performance since they can adapt to the varying requirements that threads experience along their execution. Modern SMT processors combine both flavors of resource sharing for different processor structures to provide the best of both worlds.

In the IBM POWER8 (our experimental platform), the L1 caches (both the instruction and the data cache), the (unified) L2 cache, the reorder buffer, the rename registers, as well as some issue queues and execution pipelines are dynamically shared among the co-running threads. However, it follows a hybrid approach for the main issue queue and execution pipelines (e.g., fixed-point, floating-point and load/store units). In SMT mode, threads are divided into two subsets and each subset is assigned to an issue queue half and its associated execution units. When co-running only two threads, these resources behave as statically partitioned resources. However, when the SMT degree increases, several threads are assigned to each thread subset, and these threads dynamically share the respective assigned issue queue half and execution pipelines.

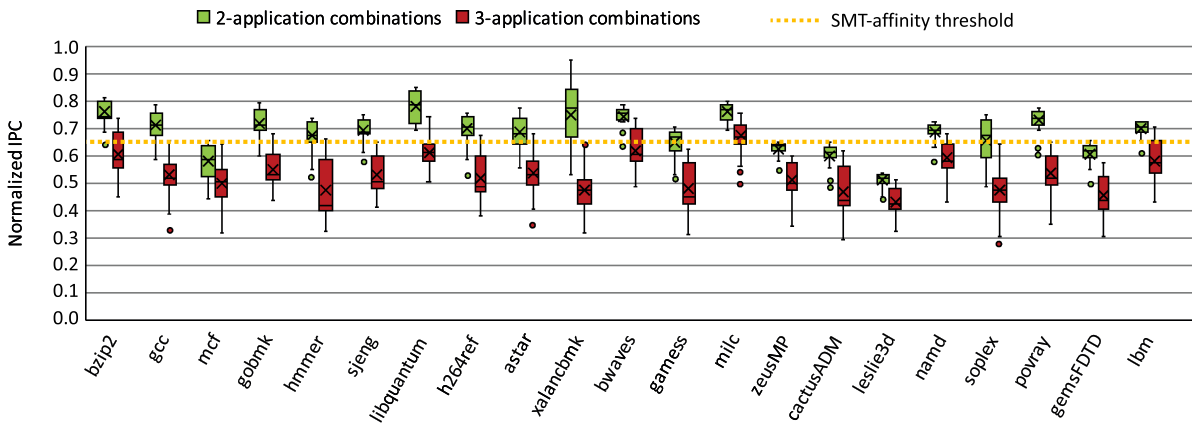


Fig. 1. Box-and-whisker chart showing the distribution of the normalized IPC for the SPEC CPU2006 benchmarks in SMT mode for all possible two-application and three-application combinations.

2.2 Resource Interference

This section analyzes the sensitivity of per-application performance when running in SMT mode relative to ST mode. Fig. 1 shows normalized IPC for the SPEC CPU2006 benchmarks when running on an SMT core of the IBM POWER8 for any possible two-application and three-application combination. Data is presented as a box-and-whisker plot, which shows the distribution of the values into quartiles. The box, divided by the median value, represents 50 percent of the data around the median (25 percent above and below the median). The remaining data points fall within the whiskers. Values that differ more than 1.5 times the inter-quartile range from the whiskers (above it for the top edge and below it for the bottom edge) are considered outliers and are represented as dots. The median value is highlighted with an *X* mark. We now discuss how resource sharing affects SMT performance for the two- and three-application combinations.

2.2.1 Two-Application Combinations

We start the analysis by studying how pairs of applications perform under SMT. The analysis provides hints as to how applications react differently to SMT execution due to reduced allocation of partitioned resources versus interference with co-runners in shared resources. Intuitively, the height of the box-and-whisker relates to whether the performance degradation of an application is mainly caused by partitioned versus shared resources.

Focusing on *xalancbmk*, we observe that its box-and-whisker is tall, which indicates that performance is highly sensitive to the interference caused by its co-runner. SMT execution with some co-runners leads to a slight performance degradation (normalized IPC of 0.95); in contrast, SMT execution with other co-runners leads to a large performance drop (normalized IPC below 0.53). On the other hand, we observe a short box-and-whisker for other applications such as *zeusMP* with a normalized IPC ranging between 0.58 and 0.65. A short box-and-whisker suggests that performance degradation primarily comes from partitioned resources since all co-runners affect an application's performance similarly.

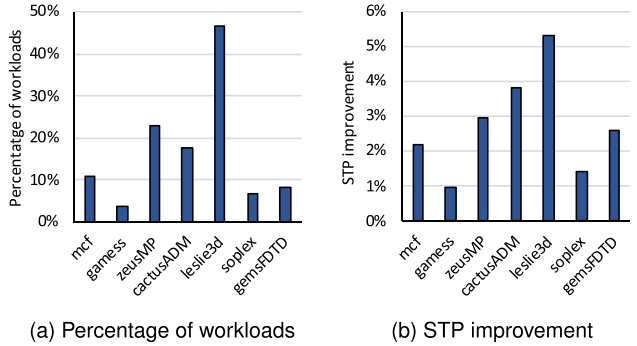
The most interesting finding, however, is to observe that the performance degradation due to resource sharing varies widely across applications. On the one hand, applications such as *milc* or *bwaves* do not suffer from a severe performance degradation as its normalized IPC under SMT execution is above 0.7. On the other hand, applications such as *leslie3d*, *cactusADM* and *zeusMP*, see their performance significantly degrade irrespective of the specific co-runner. For *leslie3d*, normalized performance can be as low as 0.45.

2.2.2 Three-Application Combinations

With three applications in a workload mix, the allocated partitioned resources are further reduced and interference in the shared resources potentially grows. A first observation is that, as we observed for the two-application combinations, while some applications are more sensitive to partitioned resources (short box-and-whisker), others are more sensitive to dynamically shared resources (tall box-and-whisker).

Nevertheless, note that applications such as *libquantum* or *xalancbmk* move from a relatively tall box-and-whisker under two-application combinations to a shorter box-and-whisker under three-application combinations. This behavior is explained by the fact that, as more applications run simultaneously on an SMT core, each application receives a smaller fraction of the partitioned resources. When a partitioned resource becomes the main performance bottleneck, the box-and-whisker gets shorter. From a symbiotic scheduling perspective, the most interesting finding is that applications such as *mcf* and *milc* do not suffer from a significantly higher performance degradation when running in three-application combinations versus two-application combinations.

Taking into account the widely different performance degradations observed for each application when running two- and three-application combinations compared to isolated execution, an intelligent scheduling policy should: (i) isolate the applications that suffer the highest performance degradations under SMT, and (ii) co-schedule applications whose performance degradation is similar under two- and three-application combinations, on a single core. Such an asymmetric schedule can minimize the high performance degradation that some applications suffer from under SMT. Doing so will improve overall system throughput. This is the case for example for a four-application workload comprised of *leslie3d*, *mcf*, *zeusMP* and *lbm*, for which an asymmetric schedule in which *leslie3d* runs in isolation on a



(a) Percentage of workloads (b) STP improvement

Fig. 2. Percentage of workloads including the applications on the horizontal axis that improve performance in an asymmetric schedule compared to the symbiotic symmetric schedule and their average performance improvement.

separate core, while the other three applications (*mcf*, *zeusMP* and *lbm*) co-run on another SMT core, improves system throughput by 9.1 percent compared to a symbiotic symmetric schedule in which the most frequent schedule co-runs *leslie3d* with *mcf* on one core and *zeusMP* with *lbm* on another core.

3 Asymmetric Scheduling

3.1 Potential for Asymmetric SMT Schedules

To illustrate the potential benefits of asymmetric scheduling on SMT cores, we devise the following experiment. Taking the SPEC CPU2006 benchmarks, we build 1400 random four-application workloads. Each workload includes at least one of the following benchmarks: *mcf*, *gamess*, *zeusMP*, *cactusADM*, *leslie3d*, *soplex* or *gemsFDTD*. (These are the seven benchmarks with the highest performance degradation when running in two-application combinations, as shown in Fig. 1.) We will refer to these benchmarks as the target applications. We evaluate the performance of each workload running on two cores using two scheduling algorithms: (i) a state-of-the-art symbiotic scheduler that dynamically selects the best predicted symmetric schedule [11], [12], which we refer to as the symbiotic symmetric schedule; and (ii) a scheduling policy that assigns the target application to a core in isolation and the three remaining applications together to the other core — we refer to this schedule as the *asymmetric schedule*.

Fig. 2 presents the results of this experiment. Fig. 2a shows the percentage of workloads for which the asymmetric schedule outperforms the symbiotic symmetric schedule. The figure shows that workloads including *leslie3d*, *zeusMP* and *cactusADM* perform better with the asymmetric schedules for 46, 23 and 18 percent of the evaluated workloads, respectively. These results confirm that thread isolation can indeed improve the performance of workloads including SMT-adverse applications. The percentage is somewhat lower for the workloads that include *mcf*, *gemsFDTD*, and *soplex* but still reach higher performance under asymmetric schedules for 11, 8, and 7 percent of the evaluated workloads, respectively. Finally, workloads including *gamess* only achieve higher performance when *gamess* is isolated on a core in an asymmetric schedule in 3.7 percent of the workloads.

To complement the previous observation, Fig. 2b on the right reports the average system throughput improvements that the asymmetric schedule achieves over the symbiotic symmetric schedule. These results only consider the workloads where the asymmetric schedule outperforms the symbiotic symmetric schedule. For instance, the figure indicates that for the workloads that include *leslie3d*, the average improvement achieved by the asymmetric schedule amounts to 5.3 percent, if we only take into account the 46 percent workloads where the asymmetric schedule effectively outperforms the symbiotic symmetric schedule. Similarly, the average performance benefits that thread isolation provides for workloads including *cactusADM* and *zeusMP* amount to 3.8 and 2.9 percent, respectively. Workloads including *GemsFDTD* and *mcf* obtain lower speedups of 2.6 and 2.2 percent, respectively. Note that these workloads do not benefit as much from asymmetric schedules, as previously discussed. Finally, as discussed above, workloads including *soplex* and *gamess* seldomly witness a performance improvement with asymmetric schedules. On average, across the workloads for which performance is improved, this benefit amounts to 1.3 percent for *soplex* and to 1.0 percent for *gamess*.

In summary, these experiments show that there is a potential performance benefit through thread isolation. However, not all workloads benefit equally. In general, as normalized performance of the application to be isolated in the asymmetric schedule is higher, the frequency at which asymmetric schedules outperform the symbiotic symmetric ones reduces, and so does the average performance benefit achieved. Consequently, only when a workload includes one of the applications that suffer the highest performance degradation when running in two-application combinations is there high potential for thread isolation. Therefore, it is important to identify those applications to provide overall system throughput benefits.

3.2 Identifying SMT-Adverse Applications

To classify applications as SMT-adverse or SMT-friendly, we devise a new threshold referred to as the *SMT-affinity threshold*. This threshold is based on the normalized IPC that applications experience when running in two-application combinations. It only considers this IPC because the goal of the threshold is to estimate when an application running in a symmetric schedule can greatly benefit from running in isolation on a core.

We set the SMT-affinity threshold for our experimental platform as follows, but it can be easily tuned towards other systems and applications. To determine the threshold, we first take the upper quartile (the value that divides the upper box and the whisker) for each application in the two- and three-application combinations and we calculate the *upper quartile average (uqa)* performance of the applications in the two- and three-application combinations. The upper quartile denotes the combinations that suffer the least from SMT interference; in other words, these are the best combinations and the ones selected by the symbiotic symmetric scheduler. On our platform, uqa performance equals 0.72 and 0.60 for the two- and three-application combinations, respectively.

We compute the SMT-affinity threshold using the two- and three-application uqa's, as follows. We compute what the best average normalized performance would be if we were to execute three applications simultaneously and the

fourth in isolation, i.e., this amounts to $(3 \times 0.60) + 1$. And we compare this against the average best performance if we were to execute the applications in pairs, i.e., this amounts to $(2 \times 0.72) + (0.72 + Perf_{SMT2/ST})$, with $Perf_{SMT2/ST}$ the application's normalized performance in a symmetric schedule. This comparison can be expressed as follows:

$$\begin{aligned} (3 \times 0.60) + 1 &> (2 \times 0.72) + (0.72 + Perf_{SMT2/ST}) \\ Perf_{SMT2/ST} &< 0.64. \end{aligned} \tag{1}$$

In other words, this means that if an application's normalized performance in a symmetric schedule is less than 0.64, we classify the application as SMT-adverse. We conservatively increase the SMT-affinity threshold to 0.65 because it is only used to determine whether thread isolation should be evaluated, but the performance predicted by the SMT interference models in the end determines whether thread isolation should be engaged or not.

The dotted line in Fig. 1 plots the SMT-affinity threshold of 0.65 for our platform. Looking at the normalized IPC of applications when running in two-application combinations, we can infer that *mcf*, *zeusMP*, *cactusADM*, *leslie3d*, and *GemsFDTD* will be classified as SMT-adverse most of their execution time because their normalized performance seldomly exceeds 0.65 with any co-runner. Note that these are the five benchmarks that benefit more from isolated execution, as shown in Fig. 2. Nevertheless, the proposed scheduler does not classify benchmarks statically, but based on the normalized performance that they achieve dynamically at runtime and thus, phase behavior can make some benchmarks move between SMT-adverse and SMT-friendly categories along their execution.

There are other benchmarks that do not frequently benefit from asymmetric scheduling such as *gamess* but could still be classified as SMT-adverse based on their normalized IPC when running with some co-runners. This situation, however, is unlikely to occur when running with the symbiotic scheduler because applications usually run with *good* co-runners to maximize performance. Furthermore, even if they are classified as SMT-adverse, the interference models (see Section 4.1) should detect that an asymmetric schedule isolating such applications would not improve performance over a symbiotic symmetric schedule and, consequently, it should keep the symmetric schedule.

By identifying the SMT-adverse applications, we avoid evaluating asymmetric schedules by isolating all applications in the workload, which reduces the scheduling overhead. Note that a higher threshold requires evaluating thread isolation with more applications. In addition, the SMT-adverse threshold prevents that model deviations end up scheduling asymmetric schedules that perform worse than the symmetric combinations. This situation is not frequent but can occur at the beginning of the execution, where correction factors (see Section 4.4) are not warmed up yet.

4 Thread Isolation

Thread isolation is applied on top of the symbiotic scheduler previously proposed by Feliu et al. [11], [12]. To adapt to time-varying execution behavior, the symbiotic scheduler estimates the optimal symmetric schedule, for each quantum. Once the symmetric schedule has been chosen, the newly proposed thread isolation technique evaluates whether any of the applications would benefit from an asymmetric schedule. If so, the identified application is isolated to run on a dedicated core; the other applications are then co-scheduled to run on SMT cores. Otherwise, the symmetric schedule is maintained.

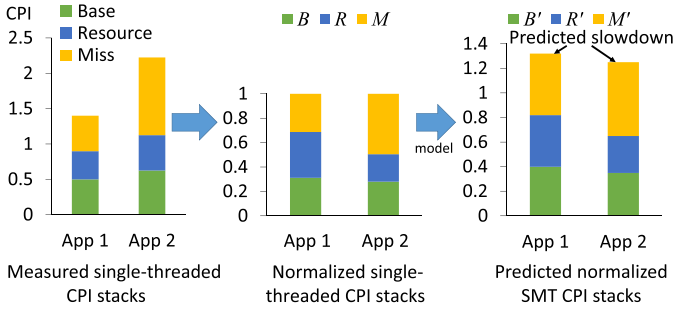


Fig. 3. Overview of the symbiosis model: first, measured CPI stacks are normalized to obtain probabilities; then, the model predicts the increase of the components and the resulting slowdown under SMT.

To understand how thread isolation works, we first provide a brief background about symbiotic scheduling. We next discuss the thread isolation proposal.

4.1 Symbiotic Scheduling

Symbiosis Models. The baseline symbiotic scheduler used in this paper is the one developed by Feliu et al. [11] [12]. This scheduler uses SMT interference models, which leverage CPI stacks to estimate job symbiosis. The left part of Fig. 3 shows a simplified CPI stack with only three components: the *base* component plus two stall components, namely *resource* and *miss*, which account for cycles during which no instructions are committed. SMT symbiosis models predict the slowdown of each application in a workload mix if this mix would be scheduled to run on an SMT core. To this end, the CPI stack of each application in single-threaded mode (i.e., when executed alone on the core) is normalized and interpreted as a probability distribution. For instance, there is approximately a 30 percent chance that application 2 executes instructions on a cycle (base component), a 25 percent chance of suffering a resource stall, and a 45 percent chance of suffering a miss stall. Then, the normalized stacks are used to calculate the probabilities of the events to generate interference if the applications would run concurrently on an SMT core. This interference introduces some performance degradation and increases the components of the SMT CPI stacks. Finally, the sum of the predicted SMT CPI components for the SMT execution is a prediction for the application's performance under SMT. The ratio of the predicted SMT CPI and the isolated CPI is a prediction for the per-application SMT slowdown.

We use regression models to predict the components of SMT CPI stacks based on the ST CPI stack components. The models follow the equation

$$C_i' = \alpha_C + \beta_C C_i + \gamma_C \sum_{j \neq i} C_j + \delta_C C_i \sum_{j \neq i} C_j, \tag{2}$$

in which C_i refers to the C component for application i in the ST stack and C_i' refers to the same component when

application i co-runs with the other applications in SMT mode. The j variable iterates over all applications that form the evaluated schedule except i, and C_j refers to the C component in the ST stacks of application j. Hence, when evaluating a two-application combination i refers to the application of which the model is estimating the slowdown and j refers to the co-runner. In the case of a three-application combinations j would iterate over the two co-runners of application i. The parameters α_C, β_C, γ_C and δ_C capture a component's specific behavior when multiple applications run simultaneously on the same SMT core.

We obtain an SMT interference model for each number of applications co-scheduled in the same core. The models are trained offline using experimental training data of the applications when running alone, and in two- and three-application combinations. We use linear regression to obtain the parameters for each component α_C, β_C, γ_C and δ_C from the ST and SMT CPI stacks of the applications. Note that the parameters are tied to CPI components and SMT level but not to applications. Therefore, as long as the training set is representative, the model can be trained once and used to schedule any set of applications. We refer the interested reader to [11] and [12] for further details on the symbiosis models.

Algorithm 1. Thread Isolation Algorithm

```
   INPUT:
   Optimal symbiotic schedule: couples of applications (C_1 =
   {A_1, A_2} to C_N ={A_2N-1, A_2N})
 1: do {
 2: Max_Asym_benefit = 0
 3: Thread_isolation_applied = FALSE
 4: for all C_i, C_j : i ≠ j      ∧
          ∃ A∈C_i∪C_j : Perf_SMT2/ST(A) < 0.65  do
 5:    STP_Symb = STP_couple(C_i) + STP_couple(C_j)
 6:    {STP_Asym, Asym_combination} =
                 Find_best_asym_combination(C_i∪C_j)
 7:    Asym_benefit = STP_Asym − STP_Symb
 8:    if Asym_benefit > Max_Asym_benefit then
 9:       Max_Asym_benefit = Asym_benefit
10:       Max_Asym_combination = Asym_combination
11:    end if
12: end for
13: if Max_Asym_benefit > 0 then
14:    Apply_to_schedule(Max_Asym_combination)
15:    Thread_isolation_applied = TRUE
16: end if
17: } while (Thread_isolation_applied)
```

Scheduling Algorithm. Estimating the performance for each possible co-schedule is a computationally challenging problem because the number of possible schedules quickly grows with the number of cores and applications. To efficiently cope with the large number of possible schedules, the symbiotic scheduler uses the technique proposed by Jiang et al. [13]. The scheduling problem is modeled as a minimum-weight perfect matching graph problem. Applications are represented as graph nodes and the weight of each edge connecting every two nodes represents the slowdown that the two connected applications will suffer if they run simultaneously on a SMT core. Hence, the perfect matching graph with minimum weight represents the schedule with the lowest performance degradation. Modeled as a graph, the scheduling problem can be solved in polynomial time using the blossom algorithm [14]. Again, see [11] and [12] for further details.

4.2 Thread Isolation Algorithm

The symbiotic scheduler, as just described, predicts the best symmetric schedule for each quantum. However, as shown in Section 3, this schedule is not always the optimal one. Depending on the workload, an asymmetric schedule could possibly lead to higher overall system performance. Therefore, after obtaining the symmetric schedule, we propose to evaluate the potential performance benefit from thread isolation. The thread isolation algorithm checks, for each possible pair of applications, whether an asymmetric schedule in which we isolate one application to a dedicated core while consolidating the other three applications on another SMT core, outperforms the predicted best symmetric schedule. If it does, the asymmetric schedule is applied.

Algorithm 1 presents pseudocode for the thread isolation algorithm. It takes pairs of applications as input; these pairs were previously selected through symbiotic scheduling.[1] The algorithm consists of a main loop (lines 1 to 17), which is repeated either until no improvement can be obtained by applying thread isolation, or there are less than two couples of applications remaining and thus thread isolation cannot be further applied.

The algorithm iterates over each possible pair of application couples C_i and C_j, and for any application A in any of these couples that potentially benefits from thread isolation, the algorithm looks for the asymmetric combination that provides the highest performance benefit among all the possible pairs of couples (see lines 5 to 11). Once the best asymmetric combination is found, it is applied (lines 13 to 16) and the main loop starts again with the remaining application couples. Finally, the algorithm returns the new schedule. This schedule can be formed by applications running in isolation, couples of applications mapped to the same core, and triplets of applications consolidated to run together.

As discussed in Section 3, the workload mixes for which thread isolation can improve performance are limited. Only when the mix includes an SMT-adverse application does thread isolation have a good chance of improving performance. To identify SMT-adverse applications (Algorithm 1, line 4) the SMT-affinity threshold is used (see Section 3.2). Every quantum, the scheduler updates for each application $\text{Perf}_{SMT2/ST}$, the performance of the application in two-application combinations normalized to its isolated performance. If $\text{Perf}_{SMT2/ST}$ is below 0.65 the application is considered SMT-adverse for the next quantum.

$\text{Perf}_{SMT2/ST}$ is calculated using the IPC that each application achieved during the last quantum it ran in any two-application combination and in isolation, respectively. These values are dynamically updated by the scheduler every quantum each application runs in either mode. In addition, a sampling phase is periodically triggered (see

1. We use a symbiotic scheduler to select the best pairs of applications but thread isolation can also be applied when these pairs are selected following any other criteria.

Section 5.4) to ensure that both the performance in isolation and in two-application combinations are recent enough to properly classify applications. At runtime, *mcf*, *zeusMP*, *cactusADM*, *leslie3d* and *gemsFDTD* are frequently classified as SMT-adverse. Other applications such as *soplex* are also classified as SMT-adverse depending on their execution phase and workload. Note that limiting thread isolation to sets of applications where there is good probability of improving performance reduces the overhead of the algorithm, since evaluating all the possible sets of four applications can become too costly for workloads with a high application count.

If a four-application combination includes an SMT-adverse application, the performance of the best asymmetric schedule and the symbiotic symmetric schedule are compared (line 7). The performance (STP) of the symmetric combination is computed with the STP estimated for the C_i and C_j couples proposed by the symbiotic scheduler (line 5), whereas the best asymmetric schedule and its estimated performance are obtained through the *Find_best_asym_combination* function (line 6). Algorithm 2 presents the pseudocode for this function. The function receives as input a four-application workload (i.e., 2 applications from each couple C_i and C_j) and just evaluates the asymmetric schedules that isolate an SMT-adverse application. The predicted STP for these asymmetric combinations is calculated as the predicted STP of the application running alone plus the STP of the triplet composed of the remaining applications running on the same core. To estimate the performance of triplets, we leverage the interference model for three-application combinations. The model estimates the slowdown of an application when it is co-scheduled on the core with two co-runners using the ST CPI stack of each application. We obtain the performance of each application in the triplet individually and the STP of the three-application combination is the sum of all of them.

Algorithm 2. Find_best_asym_combination Function

INPUT:
Set of 4 applications $S = \{A_1, A_2, A_3, A_4\}$
1: $STP_{Max} = 0$
2: **for all** $A \in S$: $A_{norm\_IPC} < 0.65$ **do**
3: $STP = STP_{alone}(A_i) + STP_{triplet}(S - \{A_i\})$
4: **if** $STP > STP_{Max}$ **then**
5: $STP_{Max} = STP$
6: $Best\_Asym = (A_i, S - \{A_i\})$
7: **end if**
8: **end for**
OUTPUT:
STP_{Max} and Best_Asym

4.3 More than Two Applications per Core

Up to now, we assumed workloads with two applications per core. We now generalize thread isolation to schedule larger workloads. The main difference with the algorithm explained in Section 4.2 is that input schedules (C_1 to C_N in the input statement of Algorithm 1) will not be couples of applications, but combinations of applications to be scheduled on the same core. For example, if we consider workloads where the number of applications ranges from more than two applications per core to less than three, input schedules should be formed by couples and triplets of applications. Consequently, the *Find_best_asym_combination* function (Algorithm 2) should also be extended to additionally evaluate thread isolation on sets of five applications (a couple and a triplet). The extension of this function itself is straightforward, as it only needs to use the interference models to evaluate the performance of the possible schedules when mapping an SMT-adverse application to a dedicated core and the remaining applications to the other core.

Our initial experiments on these workloads, however, revealed that the performance degradation when scheduling four applications on the same core is high and hinders most performance benefits of thread isolation. We therefore extended Algorithm 1 to evaluate thread isolation taking combinations of three applications — Algorithm 1 takes them in pairs (line 4). In addition, we extended the *Find_best_asym_combination* function to take seven applications as input (two couples and one triplet) and find the best combinations mapping an SMT-adverse application to a dedicated core and two groups of three applications each to two different cores.

4.4 Correction Factors

The accuracy of the symbiosis models is key for the effectiveness of thread isolation. If asymmetric schedules are erroneously applied due to symbiosis model deviations, performance can be significantly degraded. To solve this issue and make the predictions more accurate, we use correction factors as in [11].

After a schedule has been executed during a quantum, the scheduler updates the correction factors. Correction factors are defined as the actual performance divided by the performance estimated by the model. The scheduler keeps one correction factor per possible combination of applications in each SMT mode. When predicting the slowdown of an application for the next quantum, we multiply the predicted performance with the corresponding correction factor. This way, we learn from previous observations and dynamically make the predictions more accurate. At the end of each quantum, correction factors are updated using an exponential moving average function, which smooths out sudden changes in execution behavior. To update the correction factors, we need to know the isolated performance for each application. Isolated performance is obtained by very sparsely executing each application in ST mode on a core (see Section 5.4 for scheduler implementation details).

5 Experimental Setup

Before presenting our experimental results, we first detail our methodology. This includes the system that we use, the workloads, the metrics and the scheduler's implementation details.

5.1 System Features

We perform all experiments on an IBM Power System S812L server, which is a POWER8 machine consisting of 10 cores in total [15]. This processor is a dual-chip module (DCM) in which two chips with 5 cores each operate as a shared-memory machine. Each core can execute up to 8 SMT

hardware threads simultaneously. The cores feature a 64 KB L1 data cache, a 32 KB L1 instruction cache, and a 512 KB L2 cache; the L1 and L2 caches are private to a core. The on-chip 80 MB last-level cache (LLC) is shared by all 10 cores. Our IBM POWER8 system has 32 GB of DRAM installed on a single memory module, and runs Ubuntu v16.04 with Linux kernel v4.4.0.

The DCM design implies non-uniform memory access (NUMA) effects, i.e., memory accesses from cores in one chip module to the memory controllers located on the other chip module incur higher latency and reach lower bandwidth than the accesses to the memory controller on the same chip module [12]. These NUMA effects arise when more than 5 cores are used in our machine. To prevent these NUMA effects from disturbing our experimental results, we restrict our experiments to at most 5 cores, i.e., we limit our experiments to a single chip module. To evaluate a higher number of cores, the proposed algorithm would need to be extended to consider NUMA effects. This is left for future work.

5.2 Workloads

We evaluate thread isolation using randomly chosen multiprogram workloads composed out of SPEC CPU2006 benchmarks with reference input sets. For each benchmark, we measure the number of instructions required to run during 120 seconds in isolated execution and save this number as the target number of instructions. This reduces the amount of variation in the benchmark execution times across the experiments. We run each multiprogram experiment until the last application in the workload mix completes its target number of instructions. When an application reaches its target number of instructions before others do, its performance number (useful instructions executed per cycle or IPC) is saved and the application is relaunched but its performance is no longer monitored. This method ensures that we compare the same part of the execution for each application, and that the effective workload is uniform during the full duration of the experiment, i.e., a constant number of applications co-run at any point in time.

5.3 Metrics

We use system throughput (STP) [16] as the target metric to optimize for. STP is equivalent to weighted speedup [2]. STP is a higher-is-better metric and measures system-level performance. To provide a more solid and insightful evaluation, we also measure average normalized turnaround time (ANTT) [16]. ANTT is a lower-is-better metric, and is a measure for average per-application performance. ANTT provides some notion of fairness as it effectively computes the average per-application slowdown.

Running experiments on real hardware leads to non-determinism, i.e., different runs of the same experiment lead to slightly different performance results. We find the 95 percent confidence intervals for our proposed scheduler to be $\pm 0.39\%$, $\pm 0.36\%$ and $\pm 0.30\%$ for the 6-, 8- and 10-application workloads, respectively. These are fairly tight confidence intervals, much smaller than the performance improvements that we report in this paper.

5.4 Scheduler Implementation

We implement the thread isolation algorithm in a user-level scheduler, on top of the symbiotic symmetric scheduler by Feliu et al. [11], [12]. This scheduler uses *libpfm-4.8.0* to set and read performance counters, and uses Linux system calls and the cpu affinity attribute of processes to control the execution and the binding of applications in the workload mix to the selected hardware contexts, respectively. The main loop of the user-level scheduler performs the following actions.

First, at the end of each quantum, the scheduler stops the running applications, reads the performance counters, and checks if any application has completed its target number of instructions. If so, the IPC of the completed application is saved and the application is relaunched to ensure that the set of available applications is constant throughout the experiment. Second, it runs the symbiotic scheduling algorithm to determine which schedule to run in the next quantum. As mentioned before, the scheduler first determines the optimum symbiotic symmetric schedule. The thread isolation algorithm subsequently determines whether there is a potential benefit from an asymmetric schedule. If so, an asymmetric schedule is installed. The assignment of applications to hardware threads determines on which core an application is run as hardware threads are pinned to cores. Finally, once the schedule is determined and installed, the user-level scheduler goes to sleep until the next quantum.

We set the quantum length of our scheduler to 100 ms, which offers a good compromise between scheduling overhead and adaptability to the phase behavior of applications [11]. To obtain the performance in ST and in a two-application combination of all benchmarks (the former one is used to update the correction factors and to classify applications as SMT-adverse or SMT-friendly), the scheduler triggers periodically a sampling phase. In this phase, the scheduler (i) runs the applications alone on a core to record its single-threaded performance, which takes two quanta assuming that the workload consists of two applications per core, and (ii) runs a symbiotic symmetric schedule (1 quantum) to update the performance of applications in two-application combinations. Quanta during the sampling phase are set to 20 ms and sampling phases are only triggered every 200 quanta (once every 20 seconds). Thus, they only account for 0.3 percent of the total execution time.

6 Experimental Evaluation

This section first analyzes the accuracy of the SMT interference models that drive thread isolation and discuss how accurate they are across the SPEC applications. Then, we evaluate the performance benefits that thread isolation achieves, study how thread isolation works on two sample workloads, and present a sensitivity study varying the workload size.

6.1 Model Accuracy

Fig. 4 presents the accuracy of the SMT interference models used by the proposed scheduler when estimating the slowdowns of each application when it is scheduled in two- (Fig. 4a) and three-application (Fig. 4b) combinations. For each model, the figure shows the raw model error and the

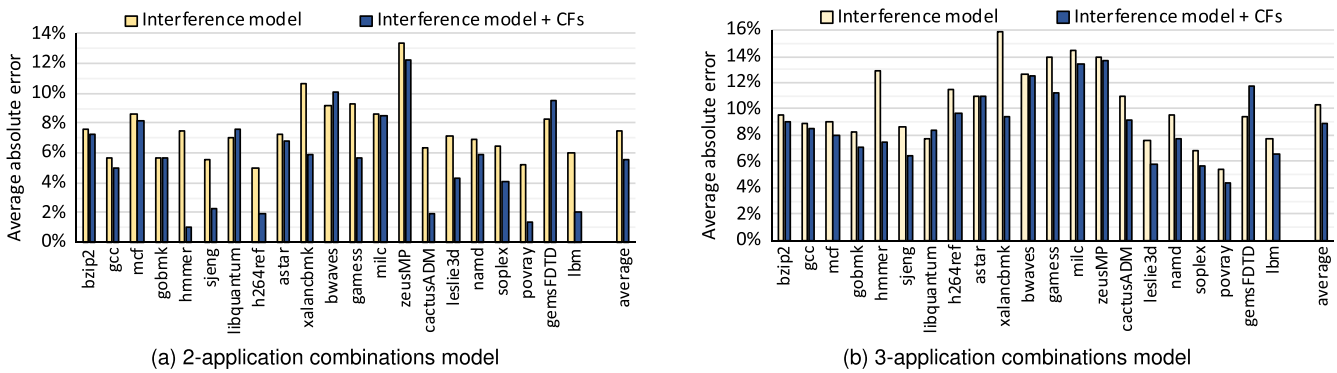


Fig. 4. Per-application average absolute error for the two- and three-application models without and with correction factors.

error when the model is used by the scheduler in combination with correction factors.

To evaluate the raw model error of the devised models, we use leave p-out cross-validation. In particular, to evaluate the accuracy of the two-application combinations model, we leave out two applications, build the interference model with the data of the remaining applications, and evaluate the model error (at multiple execution points) when estimating the slowdown of the pair of applications left out when constructing the model. These steps are repeated for all possible two-application combinations and the obtained results are aggregated to calculate the average error of the model. The same methodology is applied to obtain the average error for the three-application combinations.

The error reported for the model used in combination with correction factors is obtained from the execution of the different workloads evaluated in Section 6.3. In this case, for each quantum, we record the estimated slowdown of each application in the schedule that is going to be executed and its instruction count. After the schedule runs, we record the IPC achieved by each application. Once the workload completes, we use a profile of the IPC of the applications when running alone to obtain the actual slowdown that each application suffered in each quantum. This data is aggregated for all quanta and workloads to report the model accuracy with correction factors. Unfortunately, this accuracy results are less statistically sound than the raw model error. Note that they only consider the combinations of applications that are run (we do not know the performance of combinations that are not executed) and thus, they only take accuracy data from a subset of the combinations.

The average error for the two-application combinations model ranges from 5.0 percent (*h264ref*) to 13.9 percent (*zeusmP*), reaching an average absolute error that amounts to 7.6 percent across all evaluated applications. The average error of the 3-application combinations interference model grows for all applications. The average error ranges from 5.4 percent (*povray*) to 15.8 percent (*zeumsMP*) and reaches an average of 10.3 percent across all evaluated applications.

Correction factors effectively reduce the modeling error to average absolute errors of 5.6 and 8.9 percent for the two- and three-application combinations models, respectively. Correction factors efficacy varies across applications and mostly depends on the phase behavior of the applications, but they reduce the model error for all applications except *gemsFDTD*, *bwaves*, and *libquantum*. The performance of the former two is relatively sheer within very short intervals, which makes correction factors less meaningful. *Libquantum* has some relatively long steady phases and hence the slightly lower accuracy obtained can be a side effect of not considering all possible combinations.

6.2 Performance Evaluation

We now evaluate thread isolation in detail. We consider four scheduling policies. They are all implemented in the user-level scheduler and only differ in the scheduling policy to ensure a fair comparison. The evaluated policies include:

- *Random Scheduler.* The random scheduler obtains a random schedule for each quantum. The random scheduler serves as the baseline throughout the experimental evaluation.
- *Linux CFS scheduler.* The default scheduler in Linux is the Completely Fair Scheduler (CFS). To 'emulate' the behavior of the Linux scheduler, we allow all the applications in the workload mix to run on any of the available hardware threads. Hence, the user-level scheduler does not pin the applications to a particular hardware thread or core. By doing so, we let the OS scheduler decide which hardware thread should run which application. We also restrict the available hardware threads to two threads per core as we observe lower system throughput for Linux when the eight hardware threads of each core in our experimental platform are available for the OS to schedule applications.
- *Symbiotic Scheduler.* This scheduler, proposed by Feliu et al. [11], [12], uses symbiosis models to evaluate the performance of each possible symmetric schedule and the blossom algorithm to identify the optimal one.
- *TI-symbiotic scheduler.* Our proposed scheduler, which improves upon the symbiotic scheduler by identifying applications that benefit from thread isolation through asymmetric schedules.

6.3 System Throughput

Fig. 5 reports the system throughput improvements for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling. We consider three scenarios with 3, 4 and 5 cores. In each scenario, the number of applications in the workload mix equals twice the number of cores (i.e., 6

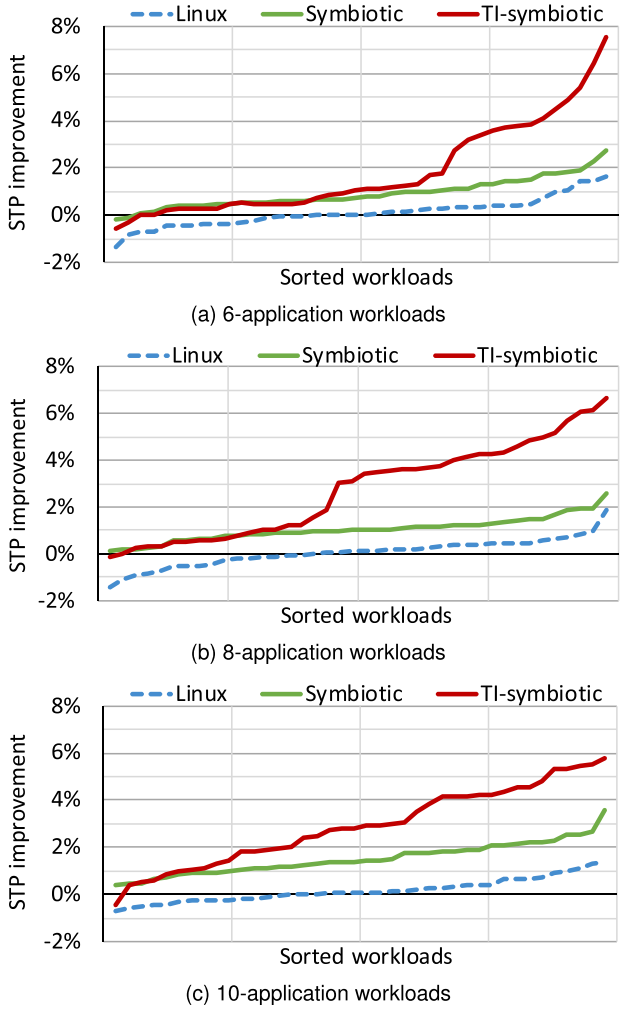


Fig. 5. STP improvement for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling.

applications for 3 cores, 8 applications for 4 cores, and 10 applications for 5 cores). Each point on the horizontal axis represents a different workload, for a total of forty workloads in each scenario. Workloads are sorted according to their normalized STP.

At first glance, the figures show that the TI-symbiotic scheduler improves STP compared to the symbiotic scheduler for a wide set of the evaluated workloads. This performance benefit comes from the ability of the TI-symbiotic scheduler to devise asymmetric schedules by isolating threads, while the original symbiotic scheduler is constrained to symmetric schedules only. We further note that both the TI-symbiotic and symbiotic schedulers outperform the Linux scheduler across all workloads. A deeper analysis of the results reveals interesting trends, which we discuss next.

For the workloads on the left (i.e., the workloads with the lowest STP improvement), the TI-symbiotic and symbiotic schedulers achieve similar performance. This suggests that these workloads do not include any of the SMT-adverse applications. Consequently, none of the threads is isolated, and as a result, the TI-symbiotic and symbiotic schedulers achieve similar performance (within statistical bounds due to non-determinism).

We note though that several workload mixes that do include SMT-adverse applications (e.g., *zeusMP*, *cactusADM* and *leslie3d*) — unexpectedly maybe — do not experience a significant STP improvement from thread isolation. The reason is that in order to improve overall system performance, the other applications in the workload mix should not be penalized too much from being consolidated on SMT cores. This limits the opportunity from thread isolation.

It is interesting to note that the number of workloads benefiting from thread isolation increases with workload size: 55 percent of 6-application workloads benefit from thread isolation versus 65% for the 8-application workloads versus 80 percent for the 10-application workloads. (These data points are identified as the points where the TI-symbiotic curve diverges from the symbiotic curve in Fig. 5.) The intuitive explanation is that the more applications in the workload mix, the higher the likelihood is to find a four-application combination for which thread isolation does improve performance. Consequently, the performance benefits of the TI-symbiotic scheduler are more significant as the number of applications in the workload mix increases.

Related to the previous observation, we note that, although the number of workloads that benefit from thread isolation increases, the maximum achieved performance benefit decreases somewhat with workload size. The maximum performance benefit from thread isolation over symbiotic scheduling (compare TI-symbiotic scheduling versus symbiotic scheduling) decreases from 5.5 to 4.8 to 4.2 percent for the 6, 8 and 10-application workloads, respectively. This is attributed to the observation that, as the number of applications in the workload increases, the relative improvement due to each application is reduced.

Overall, we report significant improvements in STP through symbiotic scheduling complemented with thread isolation. The TI-symbiotic scheduler yields improvements in system throughput up to 7.5, 6.6 and 5.8 percent for the 6-, 8- and 10-application workloads, respectively, compared to random scheduling. This is a significant improvement compared to the previously proposed symbiotic scheduler, with STP improvements of 2.7, 2.6 and 3.6 percent, respectively. The Linux scheduler on the other hand, is performance-neutral on average compared to random scheduling. On average across the workloads that include at least one SMT-adverse application, the SMT improvements of the TI-symbiotic scheduler compared to the symbiotic scheduler amount to 2.6, 3.0 and 1.8 percent for the 6-, 8- and 10-application workloads, respectively. We thus conclude that thread isolation leads to substantial improvements in system throughput. We want to re-emphasize that these results were obtained on real hardware, hence these system throughput improvements are readily available on existing systems.

6.4 Per-Application Performance

Not only does thread isolation improve overall system throughput, it also improves per-application performance. Fig. 6 reports ANTT improvement (reduction) for the TI-symbiotic, symbiotic and Linux schedulers, again relative to random scheduling. The performance trends are similar to the ones observed for STP, although the achieved improvements are smaller. For the 6- and 8-application workloads, the TI-symbiotic scheduler improves ANTT for about half

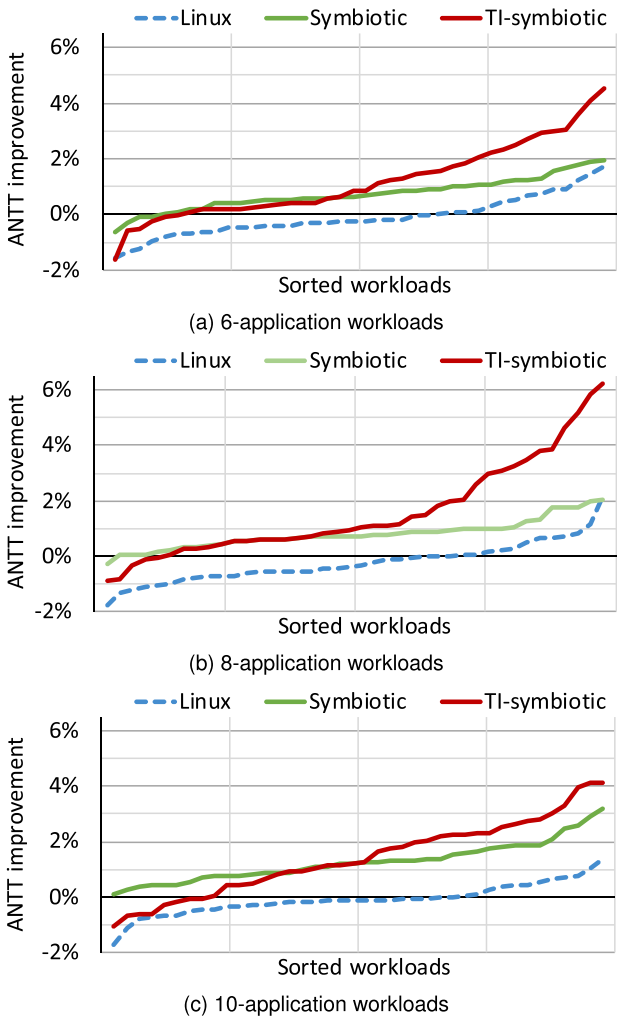


Fig. 6. ANTT improvement for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling.

of the workloads, with maximum improvements in ANTT by 4.5 and 6.2 percent, respectively, compared to random scheduling. For the 10-application workloads, the TI-symbiotic scheduler improves ANTT compared to the symbiotic scheduler for about 50 percent of the workloads, even though the symbiotic scheduler also outperforms the TI-symbiotic scheduler for around 25 percent of the workloads. Compared to the random scheduler, the TI-symbiotic scheduler improves ANTT by up to 4.1 percent.

The reason why the TI-symbiotic scheduler improves STP more than ANTT is a result of the fact that thread isolation is driven by STP, i.e., thread isolation is engaged if it is predicted to improve STP. Improving STP may in some cases lead to a (small) degradation in per-application performance. We observe that the degradation in ANTT due to thread isolation is limited — compare the TI-symbiotic versus symbiotic curves in Fig. 6. In fact, ANTT improvements are more frequent and with higher magnitude than the ANTT losses. This is a nice observation since thread isolation boosts the performance of the isolated application while adversely affecting the other consolidated applications. In other words, we conclude that the negative performance impact of thread isolation on the consolidated applications is limited.

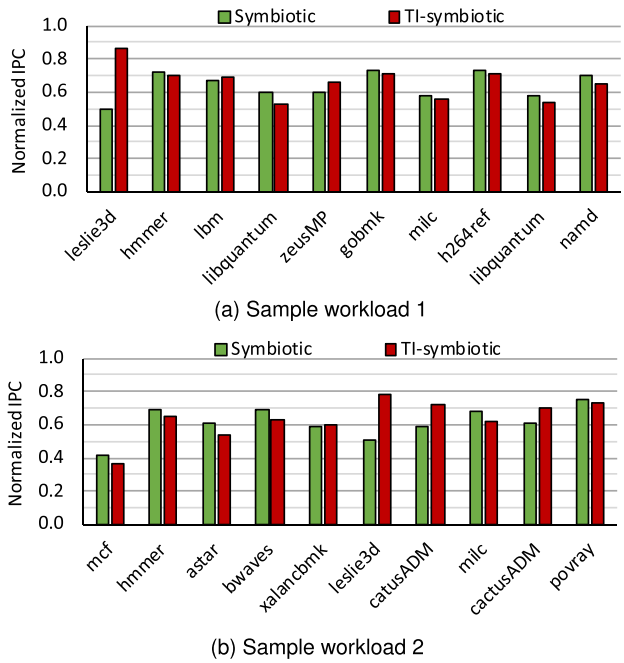


Fig. 7. Normalized IPC of individual applications when running the sample workloads under symbiotic and TI-symbiotic scheduling.

6.5 Workload Case Study

We now present a case study for two particular workloads to further gain insight into how thread isolation works and affects performance. We consider two sample 10-application workloads under the TI-symbiotic and symbiotic schedulers. Fig. 7a and 7b present the IPC that each application in the workloads achieves with either scheduler (IPC normalized to isolated single-threaded execution). Fig. 8a and 8b report the fraction of time during which the applications run in isolation, in a couple, or as a triplet under TI-symbiotic scheduling. Note that the applications always run in couples under the original symbiotic scheduler since it only devises symmetric schedules.

Focusing on sample workload 1, we observe that the normalized IPC of *leslie3d* greatly improves from 0.50 with the symbiotic scheduler to 0.86 with the TI-symbiotic scheduler. As Fig. 8a illustrates, this performance benefit is obtained by running in isolation for around 70 percent of the time. *ZeusMP* (20 percent) and *lbm* (7 percent) also run in isolation for some fraction of time, which leads to somewhat improved performance under TI-symbiotic scheduling. It is an interesting observation that, even though *zeusMP* was also classified as an SMT-adverse application, it runs more than 40 percent of the time in a triplet. This situation occurs because *zeusMP* does not suffer a much higher performance degradation when running in triplets compared to couples, as Fig. 1 shows. Consequently, when it does not run in isolation on a core, it is a good candidate to be scheduled in a triplet. The other applications run either in couples or in triplets for the entire time and suffer from a small performance degradation. The two instances of *libquantum* are the ones that more frequently run in a triplet and consequently suffer the highest, performance degradation (by 12 percent at most).

Similar observations can be made for sample workload 2. In this case, *leslie3d* runs in isolation for 75 percent of the

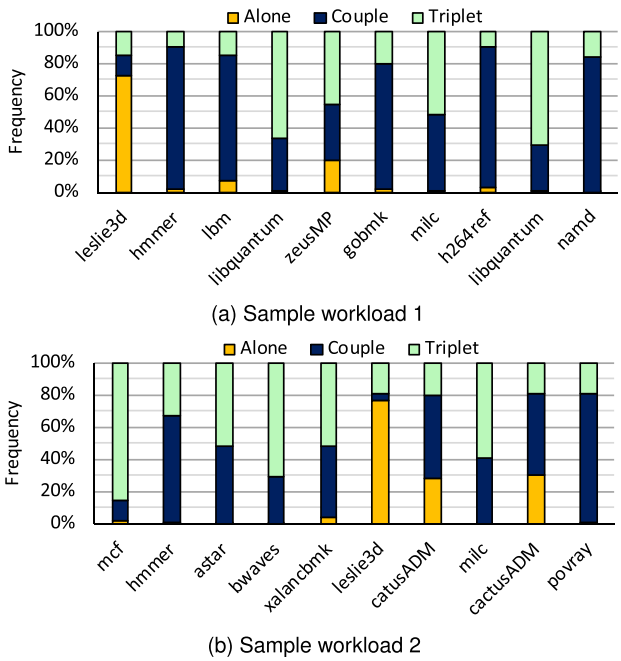


Fig. 8. Frequency of operation under TI-symbiotic scheduling for the applications of the sample workloads.

execution time and the two instances of *cactusADM* also run each one nearly 30 percent of the time alone. Note that two applications can run in isolation on different cores at the same time. Therefore, they improve their individual performance compared to the symbiotic scheduler. On the contrary, *mcf* and *bwaves* are the ones that more frequently run in a triplet and thus, suffer the highest performance degradation, although the overall degradation is lower than the total benefits obtained by *leslie3d* and *cactusADM*.

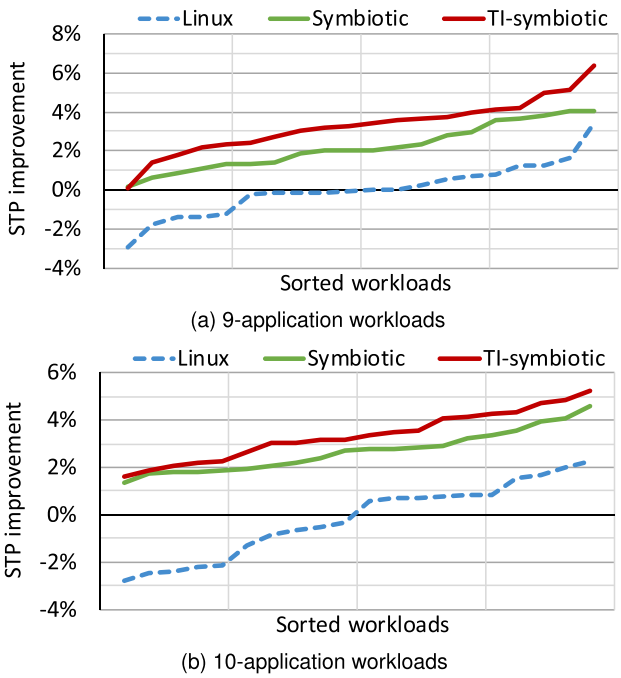


Fig. 9. STP improvement for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling running large workloads on 4 cores.

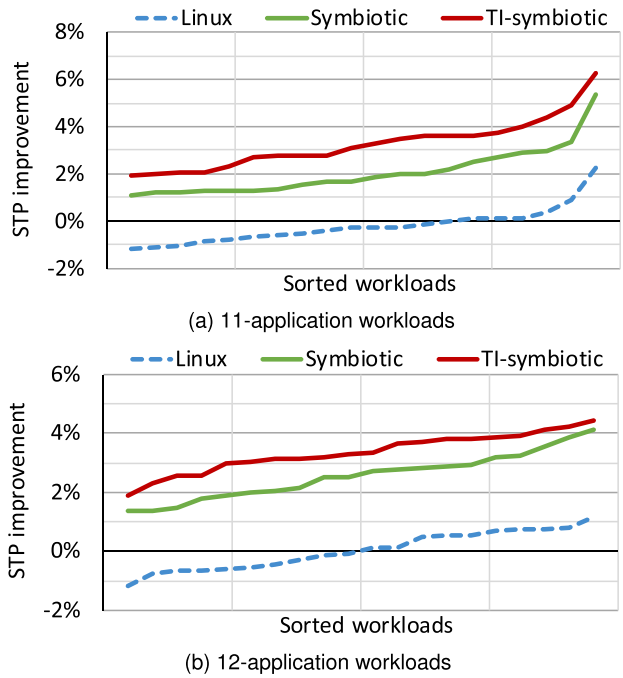


Fig. 10. STP improvement for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling running large workloads on 5 cores.

The analyzed results are not surprising. We could have expected beforehand that the applications that run in isolation would experience a performance improvement at the cost of the other applications that run in couples or triplets. However, the key point here is to observe how the performance benefit from running in isolation exceeds the performance losses from running in triplets, resulting in overall system throughput improvement. In particular for the sample workloads 1 and 2, thread isolation improves STP by 3.2 and 3.4 percent while, at the same time, also improving ANTT by 2.8 and 1.8 percent, respectively.

6.6 Workload Size Sensitivity Analysis

So far, we considered scenarios in which the number of applications is always twice the number of cores. Obviously, this may not always be the case in practice. We therefore perform a sensitivity analysis to evaluate the effectiveness of thread isolation across different workload mixes in which the number of applications exceeds the number of cores by a factor of two. Note that thread isolation could also be applied to smaller workloads with less than two applications per core. In fact, this is straightforward and leads to the same schedule as the previously proposed symbiotic job scheduler. Hence, we do not consider this case here further.

Figs. 9 and 10 report system throughput improvements for the TI-symbiotic, symbiotic and Linux schedulers relative to random scheduling when running large workloads. We evaluate workload mixes with 9 and 10 applications considering 4 cores, and workload mixes with 11 and 12 applications considering 5 cores, respectively. The figures show that thread isolation also improves STP over symbiotic scheduling for a large workload scenario. As expected, performance benefits are higher when there are fewer applications per core as the opportunity for thread isolation is higher, see 9 versus 10 applications on 4 cores as well as 11

versus 12 applications on 5 cores. As the number of applications grows, SMT-friendly applications are more likely to be already scheduled in triplets and mapped to the same core as the symbiotic schedule, which limits the ability of thread isolation to find a schedule with higher performance.

7 Related Work

Simultaneous multithreading was proposed by Tullsen et al. [1] as a way to improve the utilization and throughput of a superscalar out-of-order core by executing instructions from different threads in the same cycle. Interestingly, most of the large core resources can be shared or partitioned among the concurrent threads; this includes the reorder buffer, issue queues, functional units, physical register file, L1 caches, etc. Only a limited number of resources need replication including the program counter, global branch history, register map, return address stack, etc. Different studies report that the chip area overhead of SMT is limited to 5 to 20 percent [17], [18]. The SMT throughput benefits outweigh the chip area overheads by a significant margin. In addition to improving core utilization and system throughput, recent research has shown that SMT multicore processors are very flexible and can perform as well as or even better than heterogeneous multicores that have a fixed proportion of big out-of-order cores and small in-order cores [19]. When the active thread count is low, per-thread performance is high (a single thread can use all of the core resources); on the other hand, when the active thread count is high, high throughput is achieved by running the threads concurrently on the SMT core (at the cost of per-thread performance). Not surprisingly, the high-performance processors on the market today from IBM, Intel and AMD [20] all implement the SMT paradigm to improve system throughput.

Due to fine-grained resource sharing among co-running threads, the SMT throughput and effectiveness widely varies across workloads. If the applications in the workload balance their requirements among the shared resources, hardware utilization is high which leads to high overall system throughput. However, workloads composed of applications that severely compete for the same resources can have a substantial impact of system throughput; one such example may be cache thrashing [21], i.e., one application kicking out data of another co-running application from cache. The need to intelligently select which applications to co-run was recognized soon after SMT processors were introduced. Snavely and Tullsen [2] proposed symbiotic job scheduling, a mechanism to decide which applications to co-run on a core to maximize throughput. The proposed solution leverages sampling periods, during which all (or a subset of) the possible combinations are executed for a short duration of time to quickly identify well-performing symbiotic schedules, which are then selected to run for a longer duration. Unfortunately, this mechanism does not scale well as the number of possible combinations grows. To overcome the sampling overhead, Eyerman and Eeckhout [8] propose model-based SMT scheduling. An interference model predicts the slowdown each application would encounter when co-scheduled with any of the other applications in the workload mix, and the best performing combination is selected. However, the inputs for the model require hardware support not available in current processors. This problem was recently avoided by Feliu et al. [11], [12], who develop new interference models leveraging the CPI accounting mechanism of the IBM POWER8 processor to estimate the performance of combinations of application on current real hardware.

Many other studies have followed different approaches to maximize the throughput of SMT processors by scheduling the best combinations of applications. Parekh et. al. [22] propose thread-sensitive scheduling, a scheduling algorithm that determines the best combinations based on the applications' IPCs and memory-related metrics. Following a similar approach, Feliu et al. [9] propose to balance L1 cache bandwidth requirements across the cores to reduce interference and improve throughput. Cazorla et al. [23] guide the allocation of applications to cores based on their memory behavior and instruction-level parallelism. Other studies have explored the use of models and profiling to estimate the SMT benefit. Moseley et al. [24] use regression on performance counter measurements to estimate the speedup of SMT when co-executing two applications. Porter et al. [25] estimate the speedup of a multithreaded application when enabling SMT, based on performance counter events and machine learning. Settle et al. [26] predict job symbiosis using offline profiled cache activity maps. Mars et al. [27] use microbenchmarks called *bubbles* to measure how much an application suffers from pressure in the memory subsystem; they do so by increasing the pressure imposed by the bubble. Using this information, obtained during a characterization phase, the complexity for finding good co-schedules of applications is reduced. In follow-up work, Zhang et al. [28] 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 scheduling. Finally, Radojković et al. [29] propose a method based on Extreme Value Theory that allows for the prediction of the performance of the optimal job assignment. They state that by running a sample of the possible job assignments is enough to capture a close to optimal assignment with high probability.

As mentioned in the introduction, none of this prior work considers thread isolation to improve SMT throughput. In this work, we find that isolating SMT-adverse threads to a dedicated core while consolidating the other threads on other SMT cores can lead to significant improvements in system throughput over previously proposed symbiotic SMT schedulers.

8 Conclusions

SMT processors share most of the hardware resources among threads co-executing on the core. This design characteristic makes SMT performance strongly dependent on which applications are scheduled for concurrent execution. Co-scheduling applications with minimum interference in the shared resources reduces contention and improves performance. Many symbiotic schedulers in the literature

pursue this goal. However, as they target system throughput, previously proposed symbiotic schedulers are restricted to symmetric schedules, in which the same number of applications is mapped to each core.

In this paper, we show that asymmetric schedules, in which the number of applications per core varies, can significantly outperform symmetric schedules. Such scenarios arise due to the widely different performance degradations that applications experience under SMT. Consequently, an asymmetric schedule that intelligently isolates SMT-adverse applications to a dedicated core can provide significant performance benefits.

To leverage the previous finding, we propose thread isolation as a useful complement to state-of-the-art symbiotic (symmetric) scheduling. Thread isolation predicts whether converting a symmetric schedule into an asymmetric schedule by isolating an SMT-adverse applications to a dedicated core is going to result in higher overall system performance. If so, the asymmetric schedule is enforced. Our experimental evaluation on an IBM POWER8 server demonstrates that thread isolation improves system performance by up to 5.5 percent over previously proposed state-of-the-art symbiotic schedulers that devise only symmetric schedules.

Acknowledgments

Josué Feliu has been partially supported through a postdoctoral fellowship by the Generalitat Valenciana (APOSTD/2017/052). Additional support has been provided by the Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, as well as, by the Universitat Politènica de València through the "Ayudas a Primeros Proyectos de Investigación" (PAID-06-18) under grant SP20180140. Lieven Eeckhout's research program is supported through FWO grants no. G.0434.16N and G.0144.17N, and the European Research Council (ERC) Advanced Grant agreement no. 741097.

References

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. Int. Symp. Comput. Archit.*, 1995, pp. 392–403.

[2] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for simultaneous multithreading processor," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2000, pp. 234–244.

[3] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Proc. SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2002, pp. 66–76.

[4] G. K. Dorai and D. Yeung, "Transparent threads: Resource sharing in SMT processors for high single-thread performance," in *Proc. Int. Conf. Parallel Archit. Compilation Tech.*, 2002, pp. 30–41.

[5] M. Gomaa, M. D. Powell, and T. N. Vijaykumar, "Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2004, pp. 260–270.

[6] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, pp. 26:1–26:14.

[7] M. De Vuyst, R. Kumar, and D. M. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors," in *International Parallel Distributed Processing Symposium (IPDPS)*, 2006, Art. no. 10.

[8] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2010, pp. 91–102.

[9] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proc. Int. Conf. Parallel Architecture Compilation Tech.*, 2013, pp. 123–132.

[10] A. Morari, C. Boneti, F. J. Cazorla, R. Gioiosa, C. Y. Cher, A. Buyuktosunoglu, P. Bose, and M. Valero, "SMT malleability in IBM POWER5 and POWER6 processors," *IEEE Trans. Comput.*, vol. 62, no. 4, pp. 813–826, Apr. 2013.

[11] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic job scheduling on the IBM POWER8," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 669–680.

[12] J. Feliu, S. Eyerman, J. Sahuquillo, S. Petit, and S. Eeckhout, "Improving IBM POWER8 performance through symbiotic job scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2838–2851, Oct. 2017.

[13] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proc. Int. Conf. Parallel Architecture Compilation Tech.*, 2008, pp. 220–229.

[14] J. Edmonds, "Maximum matching and a polyhedron with 0, l-vertices," *J. Res. Nat. Bur. Standards B*, vol. 69, pp. 125–130, 1965.

[15] 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 J. Res. Develop.*, vol. 59, no. 1, pp. 2:1–2:21, 2015.

[16] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

[17] J. Burns and J.-L. Gaudiot, "SMT layout overhead and scalability," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 2, pp. 142–155, Feb. 2002.

[18] Y. Li, K. Skadron, D. Brooks, and Z. Hu, "Performance, energy, and thermal considerations for SMT and CMP architectures," in *Proc. Int. Symp. High-Perform. Comput. Architecture*, 2005, pp. 71–82.

[19] S. Eyerman and L. Eeckhout, "The benefit of SMT in the multicore era: Flexibility towards degrees of thread-level parallelism," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 591–606.

[20] T. Singh, A. Schaefer, S. Rangarajan, D. John, C. Henrion, R. Schreiber, M. Rodriguez, S. Kosonocky, S. Naffziger, and A. Novak, "Zen: An energy-efficient high-performance $\times$ 86 core," *IEEE J. Solid-State Circuits*, vol. 53, no. 1, pp. 102–114, Jan. 2018.

[21] S. Hily and A. Seznec, "Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading," Res. Rep. RR-3115, INRIA, 1997.

[22] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," Tech. Rep. 2000-04-02, Univ. Washington, 2000.

[23] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero, "Thread to core assignment in SMT on-chip multiprocessors," in *Proc. Int. Symp. Comput. Architecture High Perform. Comput.*, 2009, pp. 67–74.

[24] T. Moseley, J. Kihm, D. Connors, and D. Grunwald, "Methods for modeling resource contention on simultaneous multithreading processors," in *Proc. Int. Conf. Comput. Des.: VLSI Comput. Process.*, 2005, pp. 373–380.

[25] 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 Trans. Architecture Code Optim.*, vol. 11, no. 4, pp. 59:1–59:26, Jan. 2015.

[26] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural support for enhanced SMT job scheduling," in *Proc. Int. Conf. Parallel Architecture Compilation Tech.*, 2004, pp. 63–73.

[27] 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 *Proc. Int. Symp. Microarchitecture*, 2011, pp. 248–259.

[28] 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 *Proc. Int. Symp. Microarchitecture*, 2014, pp. 406–418.

[29] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: A statistical approach," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2012, pp. 235–248.

Josué Feliu received the MSc and PhD degrees in computer engineering from the UPV, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher with the Department of Computer Engineering, UPV. His research interests include scheduling strategies and performance modeling for multicore and multi-threaded processors. He was awarded the "IEEE TCSC Outstanding Ph.D Dissertation Award" in 2017.

Julio Sahuquillo received the BS, MS, and PhD degrees from the UPV, Spain, all in computer engineering. He is a full professor with the Department of Computer Engineering, UPV. He has taught several courses on computer organization and architecture. He has authored more than 150 refereed conference and journal papers. His current research interests include multi- and manycore processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. He is a member of the IEEE Computer Society.

Salvador Petit received the PhD degree in computer engineering for the UPV, Spain. Since 2009, he has been an associate professor with the Computer Engineering Department, UPV, where he has been teaching several courses on computer organization. He has authored more than 100 refereed conference and journal papers. His current research interests include multi-threaded and multicore processors, memory hierarchy design, GPU architecture, and resource management. He is a member of the IEEE and IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University, in 2002. He currently is a full professor with Ghent University, Belgium. His research interests include computer architecture, with a specific interest in performance analysis, evaluation and modeling, as well as dynamic resource management. He is the recipient of the 2017 ACM SIGARCH Maurice Wilkes Award, the 2017 ACM SIGPLAN OOPSLA Most Influential Paper Award, and was elevated to IEEE fellow in 2018. He served as the editor-in-chief of IEEE Micro (2015–2018).