Hy-Sched: A Simple Hyperthreading-Aware Thread to Core Allocation Strategy

Marta Navarro, Lucia Pons, Julio Sahuquillo, *Member, IEEE.*

Abstract—Simultaneous multithreading processors are dominating the High Computing Performance market. Among these processors, those supporting only two threads are being the most widely deployed in current systems, thus, only two threads compete at run-time for intra-core resources. The performance of these processors can be boosted by selecting *symbiotic* applications to be executed on the same core, which reduces the inter-application interference considerably. In this paper we propose Hy-Sched, an scheduling algorithm that exploits symbiosis to make pairs of applications to be launched on the same physical core. The proposed approach lies on the categories of the Top-Down Method for Performance Analysis. Different variants of the algorithm are explored. Experimental results show that Hy-Sched outperforms Linux on average by 15% in the studied workloads.

Index Terms—Simultaneous Multithreading, symbiotic applications, intra-core interference.

1 Introduction

Multithreaded processors provide hardware support for the concurrent execution of multiple threads on the same core. Among these processors, simultaneous multithreading (SMT) [1] is the only paradigm that allows instructions from different threads or applications to be issued for execution at the same time. This design feature allows to improve the processor throughput over their single-threaded cores counterparts, with little extra hardware.

Because of this fact, most processor manufacturers like IBM, Intel or AMD include SMT processors among their products, and these throughput oriented processors are being commonly deployed in the server segment of the market.

Commercial SMT cores support multiple (ranging from 2 to 8) threads per core. In these cores, the co-running applications compete among them for intra-core components like the ROB, physical registers, or instruction queues. As a consequence, inter-application interference is introduced in the cores between the co-running applications, which can make the performance of individual applications to significantly drop over stand alone execution. The inter-application interference depends on the contending applications. To deal with this shortcoming, some research has focused on interference-aware thread-to-core (T2C) allocation strategies within the system scheduler. The T2C policy is aimed to select applications showing some symbiosis [2] between them; that is, applications that minimize the intra-core interference thanks to the fact that they use complementary core resources.

Some previous research has focused on cores supporting a high number of threads (e.g., 8). These solutions [3] use mathematical models to estimate performance of possible combinations of applications running on the same core, which introduces a high complexity and overhead. Nevertheless, these solutions do not put special focus in SMT2 processors, which are most of the installed high-end processors and servers. In addition, mathematical model based approaches depend on characteristics of the running workloads and underlying hardware (e.g., adding more DRAM would make the model to introduce error deviations). In contrast, in this paper we focus on an universal approach.

In this paper we propose Hy-Sched (hyper-threading aware), a simple and effective scheduler for SMT-2[1] processors. We implemented the devised policy on an Intel Xeon E5-2620 v4. The devised policy is based on the categories discussed in *a Top-Down Method for Performance Analysis* by A. Yasin [4]. This method breaks down the execution time of a given application into categories according to where processor stalls rise. This allows programmers to identify possible performance bottlenecks of their applications. For instance, a high value in the *backend-bound* category may indicate that L3 or main memory is performing poorly. Since categories identify processor stalls in different processor components, we use them to look for symbiosis. This paper presents two main findings: i) performance improves when the applications co-running on the same physical core belong to distinct categories, and ii) in case there are many applications from the same category, couples of applications must be made in order to balance IPC.

Hy-Sched algorithm works on the two aforementioned findings. In this paper, we study four main variants of the algorithm (though it can be applied to any set of categories), starting from the four top-level categories, and descending to the second and third level by splitting categories of the upper levels.

This paper makes two main contributions. We propose a simple and low-overhead hyper-threading aware scheduler that improves existing state-of-the-art approaches. We found that, in order to achieve the best performance, categories from all three levels need to be considered.

2 Related Work

Some research works focused on symbiotic approaches for generic SMT processors using simulation. In [2], Snavely and Tullsen implemented SMT capabilities in a single-threaded single-core model simulating the Alpha 21264. Different multithreading levels, ranging from 2 to 6 were analyzed. This approach employs sample phases to collect inter-application interference. In these phases, the approach permutes the schedule periodically to predict the best schedule. This approach, however, can not be applied to commercial processors since the scheduler is based metrics not available in existing processors.

• *M. Navarro, L. Pons and J. Sahuquillo were with the DISCA Departament, at Universitat Politècnica de València, Spain .*
E-mail: marnaed@inf.upv.es

Manuscript received June XX, 2020; revised XXX, 2020.

1. 2 supported threads

Some works [3] have focused on the highly-threaded IBM POWER8. As in [2], this approach predicts the best performing schedule. To this end, a mathematical predictor having as input IBM performance counters estimates the performance of each application with every possible co-runner. In contrast, our approach does not perform any estimate and no *prediction* is made for the pairs of applications to be launched together but pairs are made only based on the categories the applications belong to. An improvement of this work to support more efficiently a higher multithreading level is presented in [5].

In [6] a simple and effective approach is presented. Feliu et al. show that, by evenly distributing L1 D-Cache accesses, significant performance gains can be achieved in an Intel Xeon E5645. Like ours, this approach performs on Intel processors with similar performance counters, thus we will compare our approach to this one.

3 Background

The Top-Down Method for Performance Analysis proposes a Counters' Architecture consisting of basic performance counters that allow identifying performance bottlenecks. The key idea is to apply a *high-level approach* that avoids the high learning curve of microarchitectural details.

To identify performance bottlenecks of an application at run-time, the analysis is done at the issue stage, at issue slot granularity. The top level includes four main categories, each one identifying a different source of performance loss. Below we summarize these categories.

Backend category accounts for issue slots in which no instruction is issued due to a *backend event*. For instance, due to a D-Cache miss blocking the ROB head, causing this structure to fill up and preventing instructions from being issued. This category splits into two categories in the second level: memory bound and core bound. The former refers to memory events that cause the processor to stall, while the latter refers to issue stalls that raise mainly due to structural hazards in the arithmetic functional units. The memory bound category in turn splits into new subcategories in the third level that identify the contribution of each cache level and main memory to performance drops.

Frontend category accounts for issue slots in which no instruction was issued due to a frontend event. For instance, a I-Cache miss that caused that the frontend was not able to deliver instructions to the execution pipeline.

Bad speculation category accounts for those issue slots wasted due to mispeculations (e.g., mispredicted branches). Finally, **retiring category** refers to instructions issued that were successfully executed and committed.

In summary, the top level consists of four categories, each one splits in subcategories in the second level, and similarly, each of them in new subcategories in the third level. Considering the categories of the third level, this method accounts for 23 categories in [4]. However, due to implementation constraints we consider up to 18 categories. We would like to emphasize that the main aim in [4] is to help software developers analyze performance bottlenecks. As opposite, this work uses the proposed categories to help improve scheduling performance.

4 Looking for Symbiosis: Fundamentals

Looking into the performance counters used in the Top-Down Method [4] to categorize applications in Intel processors, we found that most of the used hardware events are closely related to those used by Feliu *et al.* in [3] in the design of a scheduler for symbiotic applications in IBM processors.

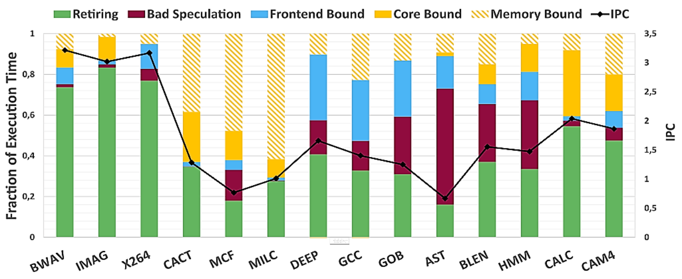

Fig. 1: Workload categorization example.

In [3], these events were used in a mathematical model implemented in the scheduler to predict the performance of any combination of applications. The number of predictions of the scheduler, and therefore its overhead in time, exponentially grows with the number of applications in the workload.

Motivated by the large set of common hardware events used in both approaches, this paper pursues to design a simple, low-overhead symbiotic scheduler. To this end, in this paper we first analyze if there exists potential symbiosis among applications belonging to the same or distinct categories.

4.1 Static Approach

The first step to study symbiosis is to define the categories to be considered, and then, look for performance counters to measure the fraction of time each application elapses in each category.

The Top-Down method proposes three levels, opening a wide design space. We can study symbiosis considering all the categories of a given level (first, second, or third), or use a hybrid multilevel approach, taking categories from two or even three distinct levels. As example, Figure 1 presents the execution time[2] of 14 SPEC CPU applications. The results correspond to a multilevel approach that considers three top-level categories (i.e., *retiring*, *frontend*, *bad speculation*), and splits the *backend* top-level category in its second level categories (i.e. *memory bound* and *core bound*). In addition, the IPC is also plotted to show that it is strongly related with the retiring category.

We tag each application according to the category that dominates its execution time to explore potential symbiosis. For instance, the first three applications *bwaves_r, imagick_r, and x264_r* are tagged as *retiring applications*, and the next three *cactusBSSN_r, mcf, and milc* as *backend applications*.

To this end, two main studies have been performed looking for symbiosis. In the first study, we analyze the behavior of pairs of applications running on the same core (i.e., only one core is considered). We measure the performance of the pairs of applications, exploring all possible combinations. Performance has been evaluated in terms of system throughput (STP) [7], also called individual speedup. We found that most performance drops appear in pairs belonging to the same category. This was expected since they tend to use the same resources (e.g., caches or arithmetic units), and consequently, they rise structural hazards in these resources.

In the second study, we analyzed how the results of the previous study could be used to improve performance when executing applications in more than one core. To provide insights in this direction, we considered four applications launched in two physical cores (i.e., four logical cores). The experimental results of the previous single-core study were used to select (off-line) symbiotic pairs (i.e., those couples achieving the best

2. See Section 6 for experimental framework details

performance) and pin them to cores. Applications, however, were forced to remain in the same core for the entire execution. We refer to this approach as *Static* since applications are not allowed to move to another core.

The results of the second study have been compared to the Linux native scheduler Ubuntu 16.04.1 LTS. For illustrative purposes, we consider only the four top-level categories. Table 1 shows the STP results for three four-benchmark mixes. Each mix consists of four applications (two instances of two different benchmarks). There are only two options to pin the benchmarks: executing two different benchmarks on the same core (*combined*) or the two instances of the same benchmark on the same core (*same*). The column *ph. core* refers to the physical core where benchmarks are initially pinned to. It can be observed that in all cases *Static* performs worse when two instances of the same application (row mix with suffix `s`, *same*) are pinned to the same physical core. Remember that *Static* does not allow applications to move since they are forced to remain in the same core across the entire execution. In contrast, this rule does not apply to Linux. In this case, Linux is able to launch applications in any of the 2 cores in hyper-threading mode, so it is allowed to dynamically move applications between cores each execution quantum. The order of benchmarks in each row of Table 1 indicates the order in which the Linux scheduler initially reads them. For instance, in mix1-c benchmarks are read in the order `mcf, calculix, mcf, calculix`.

This simple example illustrates two main downsides of Linux OS. First, the reading order can significantly affect the Linux's performance. Therefore, a weakness we found in Linux OS is that its performance does not only depend on the applications to be scheduled, but on the order in which the scheduler reads them. Second, despite Linux migrates applications between cores, its performance is below the best performance of that pair (i.e., combined) achieved by *Static*. For instance, *mix2-c* outperforms Linux by around 4.6%. These results take special relevance due to the simplicity of *Static* approach where benchmarks are not allowed to move to other cores.

This example has illustrated two main cases. Mix1 and mix2 contain two benchmarks from different categories (e.g., *mcf* and *calculix* in mix1 belong to backend and retiring categories (see Figure 1). Performance improves when launching two benchmarks of different categories. However, in mix3 both benchmarks belong to the same category. In this case, performance improves because when launching two distinct benchmarks on the same core. Even though they belong to the same category, we balance the overall IPC among the physical cores. A more refined criteria diving in the use of all the core resources could be used to provide further details about this behavior; however we found, as experimental results will show, that IPC works well for symbiosis.

4.2 Findings

This section summarizes the key findings that define the criteria used by the dynamic scheduler to choose pairs of applications to be pinned to the same core.

Mix	Ph. Core 0	Ph. Core 1	Static	Linux
mix1-c	mcf , calculix	mcf , calculix	2.836	2.670
mix1-s	mcf, mcf	calculix, calculix	2.628	2.825
mix2-c	cam4_r, hmmer	cam4_r, hmmer	2.817	2.757
mix2-s	cam4_r, cam4_r	hmmer, hmmer	2.574	2.694
mix3-c	imagick_r, calculix	imagick_r, calculix	2.676	2.566
mix3-s	imagick_r, imagick_r	calculix, calculix	2.460	2.627

TABLE 1: STP results of four four-benchmaks mixes, where c and s suffixes refer to *combined* and *same*, respectively.

Algorithm 1 Hy-Sched Algorithm: STEP 3 MAIN LOOP

```
1: while apps_list not empty do # CRITERION 1
2:     Take category from the application-tail
3:     while walk apps_list from current position to head do
4:         if category[tail appl.] ≠ category[current appl.] then
5:             Remove applications from apps_list
6:             Allocate the pair to a physical core
7:         end if
8:     end while
9: end while
10: while v_apps not empty do # CRITERION 2
11:     Take applications from tail and head to make a pair
12:     Remove applications from apps_list list
13:     Allocate the pair to a physical core
14: end while
```

Combine categories. We found that, in general, minor or no performance gains are achieved when combining two applications of the same category, thus performance improves when two benchmarks from different categories are launched in the same physical core.

Balance IPC. When there is a high percentage of applications of the same category, then the *combine categories* criterion cannot be applied. In this case, the *balance IPC* criterion will be used instead. For instance, sort applications in ascending order of IPC, and choose applications from both ends (e.g., first and last, second and next-to-last, and so on) to make pairs.

5 Hy-Sched Dynamic Algorithm

This section discusses the implementation of the proposed approach. So far we have focused on simple scenarios to illustrate the main rules that the scheduler will apply at run-time regardless of the number of cores. The studied scenarios, however, introduce over-conservative performance results since three main constraints were considered: i) the application belonged to the same category through the entire execution that consisted of a wide set of execution quanta, ii) applications were not allowed to move to other cores, and iii) only top level categories were considered by *Static*. This section introduces *Hy-Sched*, a dynamic symbiotic oriented approach which addresses all three performance constraints. Our proposal checks the category of each application at the end of each quantum (e.g., 500ms) and, based on the categories identifies, makes the new pairs for the next execution quantum.

Hy-Sched consists of three main steps that applies, at run-time, at the end of each quantum. In the first step, performance counters are read, and this data is used to calculate the necessary metrics (e.g., IPC) and obtain the dominant category. In the second step, applications for each category are sorted in ascending IPC order. In the third step depicted in Algorithm 1, applications' pairs are made starting from the first core, and applying the two main criteria discussed in Section 4.2. This step consists of two loops. In the first loop, applications belonging to different categories are paired, and then, the second loop makes couples of applications belonging to the same category applying the *balance IPC* criterion.

Hy-Sched works regardless of the number of categories. This paper analyzes four main variants of Hy-Sched: i) *TopL:* considers only the four top-level categories; ii) *TopB2L:* starting with the previous variant, it replaces the top-level *backend* category by its second level categories: *core bound* and *memory bound* (it has 5 categories); iii) *TopB2M3L:* 9 categories are included starting with the previous variant, it replaces the *memory bound* category by its third level categories (i.e. *store bound*, *L1 bound*, *L2 bound*, *ext. memory bound*); and iv) *All-3L:* this variant considers 18 categories.

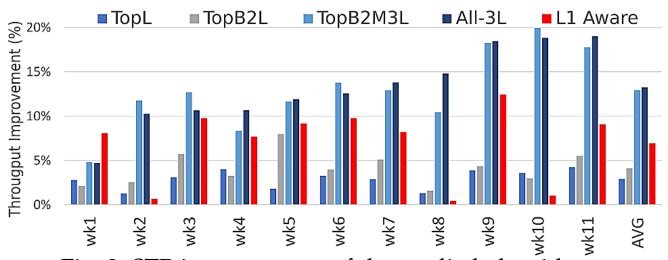

Fig. 2: STP improvement of the studied algorithms.

6 Performance Evaluation

Experiments have been conducted in a machine with an Intel Xeon E5-2620 v4 processor with 8 SMT cores running at 2.20GHz. Each core has private 8 way-32KB L1 caches and a 8 way-256KB L2 cache. There is a 20 way-20MB L3 cache, and 1.5TB DDR4 main memory. To conduct the experiments, we have developed a framework that executes the application mixes, reads the performance counters and reports these values. We built a library based on *Linux Perf* [8] to read performance counters, which allows to monitor all the Intel PMU events available to the Broadwell architecture.

Experiments were carried out with mixes consisting of 8 applications, randomly picked from the SPEC CPU2006 and SPEC CPU2017 benchmark suites. During the mix execution, each individual application commits the same number of instructions that it commits when running alone for 60s (120 500ms quanta). When executing Hy-Sched, 25 hardware events are read in each quantum in a multiplexed way in order to keep the overhead low. The execution of the mix ends when the slowest executing application finishes. If an application finishes earlier, it is relaunched, but only the data collected at the end of the first execution are considered in the results.

6.1 Performance Results

This section evaluates the performance of Hy-Sched variants compared to Linux and the L1-aware [6].

Figure 2 shows the STP improvements of the studied approaches over the Linux scheduler. It can be observed that, in general, for the studied TopL, TopB2L, and TopB2M3L, the higher the number of categories of the algorithm the higher the throughput benefits, since more symbiosis is exploited. An interesting observation is that the TopB2M3L with only 9 categories reaches similar performance to All-3L. On average, TopB2M3L reaches performance improvements by 14%, reaching by 20% in some workloads. On the other hand, it can be observed that L1-aware poorly outperforms Linux in those workloads (3 out of 13) workloads where the L1 is not the major performance bottleneck, reaching on average half the performance improvements of TopB2M3L.

We looked into the reasons that explain the high disparity in performance among the studied variants. We found that categories are not uniformly distributed at run-time but the Backend Bound category clearly dominates while other categories like Bad Speculation or Frontend Bound rarely appear.

As example, each cell (*category i, category j*) in Table 2 shows the percentage of pairs made across the execution time including an application from *category i* and another from *category j*. Results correspond to TopB2M3L when running workload *wk10*. Those categories having all their cells in the row and column less than 0.5% have not been included. Note that only Retiring and Backend Bound behaviors are exhibited. This the common trend, and although we found Bad Speculation behavior in very few workloads, its percentage is below 7%. Retiring behavior is shown by only one application of the pair in 8.78% (see retiring row) of the pairs made; and all the remaining cases exhibit Backend Bound subcategories (Store Bound, L1 Bound and Ext. Memory). This fact explains why descending only to the second level of the Backend Bound category results in minor performance benefits compared to the top-level Hy-Sched variant, as shown in Figure 2.

Category	**Ext. Memory**	**L1**	**Stores B.**	**Core B.**	**Retiring**	TOTAL row
Retiring	2.70%	1.35%	0.68%	4.05%	0.00%	*8.78%*
Core Bound	21.62%	10.81%	0.68%	29.75%		*62.86%*
Stores Bound	0.00%	0.68%	0.00%			*0.68%*
L1 Bound	17.57%	0.68%				*18.25%*
Ext. Memory	9.46%					*9.46%*
TOTAL column	*51.35%*	*13.52%*	*1.36%*	*33.80%*	*0.00%*	*100.00%*

TABLE 2: Categories of the two applications of the couples made (in %) in wk10.

Effects of more categories. We found that the retiring category fraction increases with the two approaches introducing more categories "finer granularity", which explains why these approaches are the best performing ones. On the other hand, we also found that only including over 9 categories (e.g., TopB2M3L or All-3L) results in a high number of category changes (around 40% of the execution quanta).

7 Conclusions

This paper has proposed Hy-Sched, a low-overhead scheduler that exploits symbiosis in 2-SMT processors. We have shown that the categories identified in [4] behave by design as symbiotic among them. Results, have shown that most of the symbiosis can be highly exploited by using only 9 categories. Experimental results show that Hy-Sched outperforms Linux on average by 15% across the studied workloads, doubling the performance improvements reached by the state-of-the-art L1-aware approach. Hy-Sched has been implemented in an Intel processor, but it can be applied to any SMT-2 processor.

Acknowledgments

This work has been partially supported by Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, and Generalitat Valenciana under Grant AICO/2019/317.

References

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.
[2] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," 2000.
[3] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic job scheduling on the IBM POWER8," in *2016 IEEE International Symposium on High Performance Computer Architecture*, 2016, pp. 669–680.
[4] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 35–44.
[5] J. Feliu, S. Eyerman, J. Sahuquillo, S. Petit, and L. Eeckhout, "Improving IBM POWER8 Performance Through Symbiotic Job Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2838–2851, 2017.
[6] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, C. Fensch, M. F. P. O'Boyle, A. Seznec, and F. Bodin, Eds., pp. 123–132.
[7] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 93–96, 2014.
[8] T. Gleixner, I. Molnar, "Performance counters for linux," 2009.