Revisiting Symbiotic Job Scheduling

Stijn Eyerman, Pierre Michaud, Wouter Rogiest

Revisiting Symbiotic Job Scheduling

Stijn Eyerman*, Pierre Michaud†, Wouter Rogiest*
*Ghent University, Belgium
†INRIA Rennes, France

***Abstract*—Symbiotic job scheduling exploits the fact that in a system with shared resources, the performance of jobs is impacted by the behavior of other co-running jobs. By coscheduling combinations of jobs that have low interference, the performance of a system can be increased. In this paper, we investigate the impact of using symbiotic job scheduling for increasing throughput. We find that even for a theoretically optimal scheduler, this impact is very low, despite the substantial sensitivity of per job performance to which other jobs are coscheduled: for example, our experiments on a 4-thread SMT processor show that, on average, the job IPC varies by 37% depending on coscheduled jobs, the per-coschedule throughput varies by 69%, and yet the average throughput gain brought by optimal symbiotic scheduling is *only* 3%. This small margin of improvement can be explained by the observation that all the jobs need to be eventually executed, restricting the job combinations a symbiotic job scheduler can select to optimize throughput.**

We explain why previous work reported a substantial gain from symbiotic job scheduling, and we find that (only) reporting turnaround time can lead to misleading conclusions. Furthermore, we show how the impact of scheduling can be evaluated in microarchitectural studies, without having to implement a scheduler.

I. Introduction

If resources on a processor chip are shared between cores (shared memory bus, shared cache, shared core resources, etc.), the performance of a job may be impacted greatly by the behavior of the co-running jobs. Therefore, researchers have proposed symbiotic job scheduling [33]: selecting combinations of jobs such that a performance metric (throughput, latency, fairness, etc.) is optimized.

Most of these papers however only consider separate coschedules: what combination of jobs maximizes system performance, or which jobs should be coscheduled with a time-critical job to meet the timing requirements, etc. These studies do not take into account that all jobs in the system eventually must be executed, possibly resulting in a lower average performance when we are forced to execute the jobs not yet selected. A few studies evaluated symbiotic job scheduling by doing real throughput experiments with fixed workloads, where jobs enter the system at some time, and only leave when they are fully executed. These papers either measure the *turnaround time* (i.e., the time between the submission of an individual job and its completion) [10], [33], [34] or the *makespan* (i.e., how long it takes to execute a set of jobs from beginning to end) of a *small* set of jobs [32], [39]. To our knowledge, no previous study has considered the impact of symbiotic job scheduling on the long-term maximum throughput that can be achieved on a fixed workload.

In this paper, we study the impact of symbiotic job scheduling on maximum throughput, i.e., can the throughput of a fully loaded system be boosted by symbiotic job scheduling, assuming a fixed workload? In particular, we find that

- The maximum throughput increase of symbiotic job scheduling can be theoretically evaluated, without actually implementing a scheduler and performing a throughput experiment.
- The impact of symbiotic job scheduling on maximum throughput is small compared to the variance in per-job performance and the variance in per-coschedule instantaneous throughput.
- For highly-loaded systems, slightly increasing maximum throughput can result in a large reduction of the turnaround time, confirming earlier results about symbiotic job scheduling.

We conclude that the impact of symbiotic job scheduling on maximum throughput is smaller than what one intuitively expects based on the job performance sensitivity to the co-running jobs. It is also smaller than expected from results in prior work, either because they do not evaluate a long enough fixed workload, or because they report turnaround time, which is heavily impacted by jobs inter-arrival times.

The next section discusses related work. Sections III and IV introduce our framework for calculating the optimal throughput by using symbiotic job scheduling. Section V shows quantitative results for two different configurations: a four-way simultaneous multi-threading (SMT) core, and four cores sharing a cache. We find that the throughput increase is minimal and we provide insight into why this is the case. Section VI shows that turnaround time can be a misleading metric for assessing performance improvements. Section VII discusses using the optimal throughput as a metric for microarchitecture studies to incorporate scheduling in the results. We conclude in Section VIII.

II. Related Work

Resource Allocation Scheduling: There is abundant literature on the problem of scheduling several independent *parallel* jobs on multiprocessors [14]. The problem is how to best share computing resources (in space and/or time) between jobs. Whether parallel jobs demand a fixed number of processors, or can vary their demand at run time, changes the scheduling problem [14]. In the current multicore era, this problem, which used to concern high-performance computing, now concerns general purpose systems too [7].

In this study we consider only *sequential, independent* jobs. Moreover, we assume that the scheduler cannot control resource sharing directly, i.e., shared hardware resources

(SMT core, caches, memory bandwidth, etc.) do not feature programmable partitioning mechanisms.

Performance metric: The best way to schedule jobs depends on the performance metric that one seeks to optimize. For instance, Jain et al. considered symbiosis effects in an SMT processor for a soft real-time scheduling problem where the goal is to keep the number of missed deadlines below a fixed fraction [16]. In this study, our primary performance metric is the long-term average throughput, i.e., the total work executed divided by the makespan, considering a sufficiently large number of jobs.

Heterogeneous multiprocessors: The problem of scheduling independent sequential jobs on heterogeneous multiprocessors (i.e., containing different sorts of processors) has been studied extensively, mostly from a theoretical point of view [4]. In particular, Lawler and Labetoulle showed that, when preemption is allowed on a heterogeneous multiprocessor, a schedule minimizing makespan can be obtained by linear programming [20]. Miller used linear programming to find optimal schedules for heterogeneous multiprocessors under the assumption, similar to ours, that the workload consists of several different job types, with a fixed quantity of work per job type [23]. The problem of scheduling for heterogeneous multiprocessors has resurfaced in the multicore era with heterogeneous multicores. Studies advocating heterogeneous multicores for performance generally try to maximize some instantaneous (i.e., per-coschedule) throughput metrics, such as total IPC (instructions per cycle) or *weighted speedup*[1]. Scheduling heuristics that try to maximize instantaneous throughput reduce the scheduling problem to a *job-to-core assignment* problem, i.e., finding a good mapping of K jobs (or less than K) onto K cores [2], [18], [19], [24], [29], [38].

Job-to-core Mapping: Job-to-core assignment is a problem not only for heterogeneous multicores, but also for *partially symmetric* homogeneous multicores, e.g., multicores with identical cores and partially shared resources. Typical examples of partial symmetry are multi-chip nodes and multicores with SMT cores. Many studies have looked at the job-to-core assignment problem for homogeneous multicores, where the scheduling algorithm exploits application symbiosis for trying to maximize instantaneous throughput, or some other instantaneous metrics (geometric mean speedup, harmonic mean speedup, energy efficiency, etc.) [1], [3], [8], [9], [17], [21], [22], [25], [28], [40]. Tian et al. [35] studied the theoretical problem of finding optimal job-to-core assigments for minimizing makespan, assuming a number of jobs equal to the total number of cores (i.e., as shortest jobs complete, some cores are left idle).

Symbiotic Job Scheduling: On *fully symmetric* multicores, the job-to-core assignment problem does not exist: all job-to-core assignments are equivalent (except possibly when cache and branch predictor affinity matters). Researchers who studied the scheduling problem for symmetric multicores or SMT processors generally consider that there are more jobs in the system than available logical cores. The problem is reduced to selecting a coschedule of K jobs (or less than K, see, e.g., [13]) to run on the K cores in each scheduling interval. There is an abundant literature, started decades ago, about scheduling independent sequential job on symmetric multiprocessors for minimizing turnaround time or makespan. Here we consider the scheduling problem when resource sharing impacts the execution time of jobs, i.e., *symbiotic* scheduling [33].

Early studies on symbiotic scheduling have shown that weighted speedup (computed over jobs present in the system) can be increased significantly by coscheduling jobs that go well together [5], [26], [33], [34], [36]. Weighted speedup is an instantaneous throughput metric: it measures the quantity of work done during a fixed time period, but that work is *variable* [11]. Though variable workloads correspond to certain realistic situations [31], in many other situations the workload is *fixed*, i.e., it is independent from processors performance and from past scheduling decisions.

Some papers have evaluated symbiotic scheduling on fixed heterogeneous workloads. Snavely et al. [33], [34] simulated throughput experiments where jobs come and go, assuming exponentially distributed inter-arrival times and job sizes. They showed that symbiotic scheduling decreases the average turnaround time significantly. Eyerman and Eeckhout [10] proposed a symbiotic scheduling method that they evaluated in a way similar to Snavely et al. Settle et al. [32] and Xu et al. [39] implemented symbiosis-aware schedulers that they evaluated by measuring the makespan of some fixed workloads consisting of small sets (8-16) of jobs. With such small workloads, the effect of idling cores cannot be neglected. For instance, Xu et al. found that, when jobs are SPEC benchmarks run *to completion*, a simple symbiosis-unaware *long-job-first* scheduler outperforms their symbiosis-aware scheduler [39].

III. Definitions and Assumptions

A. Throughput experiment

A throughput experiment is an experiment that mimics a realistic setup where multiple distinct jobs, possibly of different types, enter the system at some time, and only leave when they are fully executed. A throughput experiment assumes certain distributions of job types, job sizes, and job inter-arrival times. We distinguish two types of throughput experiment:

- *Maximum throughput experiment*: a large pool of jobs is available, such that the processor is always fully loaded (all cores/thread contexts are occupied) and a large set of job combinations can be selected (but the distribution of job types is fixed). The goal of this experiment is to obtain the maximum throughput the processor can handle. This experiment is representative for a server with batched jobs.
- *Latency experiment*: jobs arrive at a certain rate and are queued when they cannot be executed immediately. Only the jobs currently in the system can run on the cores. To prevent an evergrowing queue, the arrival rate must be less than the maximum throughput. The average throughput equals the arrival rate, so symbiotic job scheduling only impacts turnaround time and server utilization. This is similar to an interactive environment, where jobs enter following the clients' demand.

In this paper, we mainly focus on the maximum throughput experiment, although we also discuss the impact on a latency experiment.

[1]Weighted speedup is the sum of speedups of all the jobs running simultaneously [33], where speedup is typically defined relatively to isolated execution on a reference machine.

B. Unit of work

Throughput is the quantity of work per unit of time (second, cycle,...), so defining a throughput metric requires defining a unit of work. For this study we have considered two common units of work: the instruction and the *weighted instruction* [11]. Weighted instructions take into account that some instructions have a longer execution time, and thus represent a larger amount of work (e.g., floating point operations, irregular memory patterns, etc.). The weight of the instructions is determined by their execution time on a *reference core*, e.g., a single core running a single program. Jobs have the same number of weighted instructions, and thus the same quantity of work, if they have the same execution time when run alone on the reference core. *Weighted instructions per cycle* (WIPC) is defined as a job's IPC divided by its reference IPC. WIPC is equivalent to the commonly used weighted speedup metric (see Section II). Due to the limited space, we present only results for the weighted instruction as unit of work, but we checked that our qualitative conclusions also hold for the instruction as unit of work.

C. Symbiotic job scheduling

Job symbiosis is the effect that a job's performance can be impacted by the behavior of other jobs through shared resources. Job symbiosis is high when they have little impact on each other, while large negative interference in shared resources leads to bad symbiosis. Symbiotic job scheduling is the process of selecting (a) job combination(s), here called coschedules, out of a pool of jobs such that a certain performance metric is optimized. Note that the term 'symbiotic job scheduling' has been mainly used for SMT processors, but it can be broadened to any situation involving shared resources that cause interference between jobs (e.g., shared cache, shared memory bandwidth, etc.).

D. Assumptions about the workload and processor configurations

In order to focus our study and limit the number of results, we make some assumptions about the workload and the processor configurations. For the workload, we assume that

- There are N different job types. The workload contains an unlimited (in practice, very large) number of jobs of each type. Jobs of the same type have the same behavior. We believe N is usually small (e.g., $N = 4$), because servers typically execute only a small set of programs (e.g., web servers, database servers, etc.). Nevertheless we checked that increasing N does not impact our conclusions significantly. In the evaluation section, each benchmark defines a distinct job type.
- The job types are equally likely (i.e., jobs are equally distributed across the job types) and **all job types contribute the same total amount of work**. Note that this assumption is advantageous to symbiotic scheduling: if a particular job type had more weight than the other job types (by contributing more jobs, or larger jobs), it would dominate the execution, thereby limiting the possibilities to exploit symbiosis.
- The workload is fixed. When we compare different schedulers, they all execute the exact same work, i.e., the same number and types of jobs.

In this paper, 'workload' denotes both a particular combination of N job types and the sequence of jobs defined by this combination.

For the processor configuration, we assume that

- There are K cores or hardware thread contexts. We consider a relatively small K in our setup, corresponding typically to a single-chip fully symmetric multicore or SMT. It should be noted that, under our workload assumptions, symbiotic scheduling for multiple identical machines can be reduced to the problem of symbiotic scheduling for a single machine if the performance impact of network bandwidth sharing can be neglected. An optimal global schedule can be found by dividing the workload equally between machines so that all per-machine workloads are statistically identical, and by searching an optimal local schedule independently for each machine.
- We consider only fully symmetric multicores or SMT processors. Consequently, all the job-to-core assignments are equivalent. The job-to-core assignment problem has been studied extensively for heterogeneous multicores and for partially symmetric homogeneous multicores (see Section II).

IV. Optimal Throughput

In [11], Eyerman et al. develop throughput metrics that correspond to the average throughput of a throughput experiment assuming a FCFS (first-come first-served) scheduler, i.e., jobs are scheduled in the order they arrive. In this section, we elaborate a technique to calculate the theoretically maximum throughput of a processor, assuming a perfect scheduler.

We consider N job types and K cores. We consider a job queue that at $t = 0$ contains all the jobs that need to be executed. The perfect scheduler knows the characteristics of all jobs in the queue, i.e., the performance of all jobs in all possible coschedules. The scheduler can pick jobs in any order, not necessarily the queue order. We assume that the workload is fixed and that all the job types contribute the same total amount of work, as described in Section III-D.

We are looking for a perfect job schedule, i.e., one that maximizes throughput. Let x_s be the fraction of time that the machine spends executing coschedule s under the perfect scheduler. The x_s are unknown, they define the scheduler we are looking for. The instantaneous throughput for coschedule s is

$$it(s) = \sum_{b=1}^{N} r_b(s) \tag{1}$$

where $r_b(s)$ is the total execution rate (in unit of work per second, or per cycle if the clock cycle is fixed) for job type b in coschedule s. For example, if coschedule s contains two jobs of type b, $r_b(s)$ is the sum of the execution rates of these two jobs. If coschedule s contains no job of type b, then $r_b(s) = 0$. The perfect scheduler knows all the $r_b(s)$ values. The average throughput is

$$\text{throughput} = \sum_{s \in \mathcal{S}} x_s \times it(s) \tag{2}$$

where $\mathcal{S}$ is the set of all coschedules. We seek to maximize the average throughput under the following constraints:

$$\forall s, \quad x_s \geq 0 \tag{3}$$

$$\sum_{s \in \mathcal{S}} x_s = 1 \tag{4}$$

$$\forall b \in [2, N], \quad \sum_{s \in \mathcal{S}} x_s \times r_b(s) = \sum_{s \in \mathcal{S}} x_s \times r_1(s) \tag{5}$$

Equation 5 expresses the fact that all the job types contribute the same total amount of work. Maximizing (2) under constraints (3), (4) and (5) is a linear programming problem, for which efficient algorithms are available. In particular, we use the GNU linear programming kit (glpk) to solve the linear program.

The result is a set of per-coschedule time fractions x_s that maximizes throughput. These time fractions define an optimal schedule. This optimal schedule can be implemented because we assume an unlimited number of jobs of each type. An interesting property of linear programming problems is that there exists an optimal solution for which the number of non-zero variables does not exceed the number of equality constraints. In this case, the number of equality constraints is equal to the number of job types (Equation 4 and $N-1$ times Equation 5). Hence the number of coschedules in the optimal schedule that have a non-null time fraction does not exceed the number of job types.

Instead of maximizing throughput, we can also calculate the schedule that minimizes throughput. This can be useful for analysis purposes, by providing upper and lower bounds for throughput values, valid for any scheduler.

V. Evaluating Symbiotic Job Scheduling for Maximum Throughput

The metric developed in the previous section calculates the maximum achievable throughput, assuming a perfect scheduler that knows the performance of each job in each coschedule, and that can select any coschedule at any point in time, which practically means that all the jobs are available from the beginning. Although a realistic scheduler that does not have all of this knowledge will probably perform worse, it is interesting to compare this optimal scheduler to the base FCFS scheduler, which knows nothing about the workload. In this section, we evaluate and analyze the impact of symbiotic job scheduling for two processor configurations.

A. Experimental setup

The first configuration is a 4-way SMT 4-wide out-of-order core. An SMT core has a large amount of sharing (core resources, caches and memory controller), leading to large variations in job symbiosis. Furthermore, prior work has shown a considerable impact on performance by using symbiotic scheduling for SMT [10], [33], and recently, SMT cores were shown to be a competitive design point for various degrees of thread level parallelism [12].

The second configuration is a multicore consisting of 4 4-wide out-of-order cores, with a shared last-level cache and shared memory bus. This configuration has a smaller degree of sharing, but the sharing characteristics may be different from the SMT configuration, because the SMT core's performance is often dominated by core resources sharing rather than by cache and memory sharing.

TABLE I. Selected SPEC CPU 2006 benchmarks

Benchmark	Input	Benchmark	Input
bzip2	input.program	libquantum	ref
calculix	ref	mcf	ref
gcc	cp-decl.i	perlbench	diffmail.pl
gcc	g23.i	sjeng	ref
h264ref	foreman_ref_encoder_main	tonto	ref
hmmer	nph3.hmm swiss41	xalancbmk	ref

We selected 12 SPEC CPU2006 benchmarks based on their SMT and cache sharing behavior: they approximately uniformly cover the space of low- to high-interference benchmarks, see Table I. We simulated all 1,365 combinations (with repetition) of 4 benchmarks out of the 12 selected benchmarks using Sniper [6] (we use the new instruction window centric core model), on both configurations (SMT core and quad-core processor). Unless mentioned otherwise, we assume the ICOUNT fetch policy with dynamic sharing of the ROB (and all other non-architectural core resources) for the SMT configuration.

Using the performance numbers obtained by the simulations, we calculate the average throughput for the FCFS scheduler, the optimal scheduler and the worst scheduler, as described in Section IV. We evaluate the throughput for all possible workloads consisting of N job types out of the 12 different SPEC benchmarks we consider. The default value of N is 4, leading to 495 different workloads (number of combinations without repetition). We assume that all N job types are equiprobable and that all the jobs execute the same quantity of work, as explained in Section III. As mentioned before, we only report results for the weighted instruction as unit of work, which means that we assume jobs that are sized such that they all have the same execution time when run alone on the baseline 4-wide out-of-order core.

Note that for a specific workload, there are multiple possible coschedules: for a workload consisting of 4 different job types and 4 cores/thread contexts, there are 35 different possible coschedules (number of combinations with repetition of 4 out of 4; e.g., for workload ABCD, we have coschedules AAAA, AAAB, AAAC, ..., DDDD).

B. Results

Figure 1 shows the results for the 4-way SMT core and quad-core configuration, for $N = 4$ job types.

The figure show three bars. To limit the length of the discussion, we focus on the SMT configuration. The quad-core configuration has similar conclusions with slightly different numbers. The first bar shows the variance of the IPC of a job in different coschedules (within the same workload). The zero line corresponds to the average IPC of a job (over all coschedules), the positive bar shows the maximum IPC (relative to the average IPC) and the negative bar corresponds to the smallest IPC of that job. We show both the average over all job types and workloads, as the maximum and minimum across all job types and workloads. The figure shows that on average over all workloads, the IPC of a job can be 23% higher or 14% lower than the average IPC, resulting in a variation of 37%. The extreme values are +108% (hmmer's

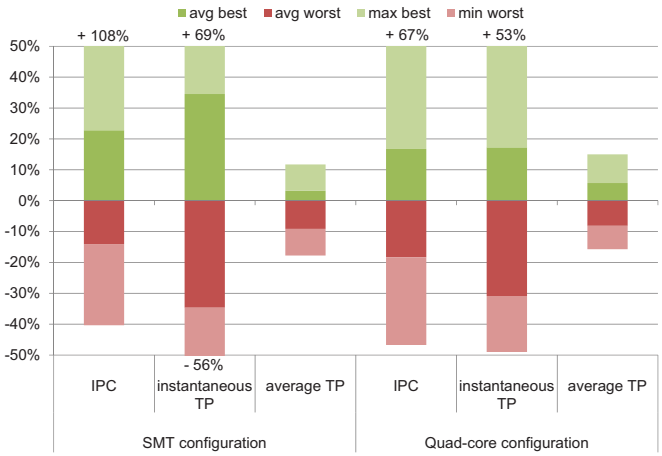


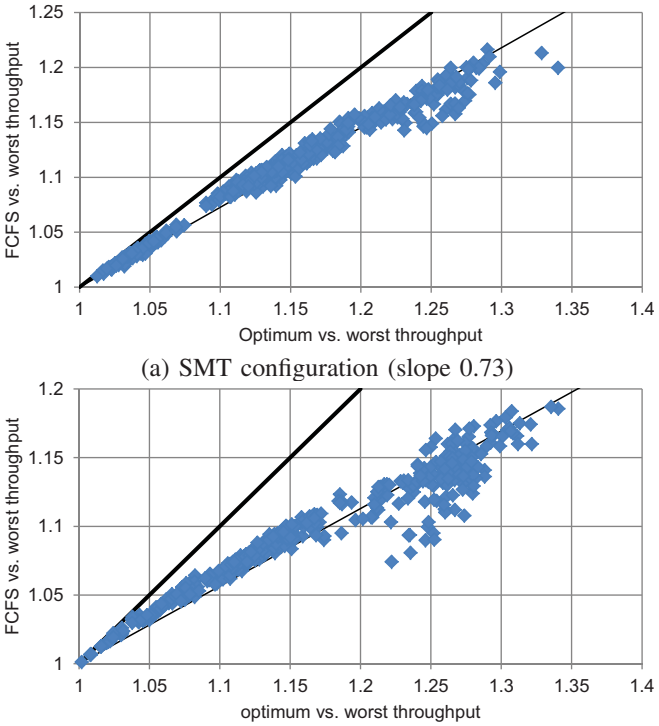
Fig. 1. Variation of the per-job IPC, the per-coschedule instantaneous throughput and the average throughput for 4 job types and both configurations.

IPC in a workload with calculix, gcc.g23 and libquantum is on average 0.31, but can be up to 0.64 depending on the coschedule) and -40% (for gcc.g23 in combination with gcc.cp-decl, sjeng and xalancbmk), so a spread of 148%. These numbers show that jobs are sensitive to the behavior of the co-running jobs.

The next bar shows the same metrics for the instantaneous throughput, i.e., the total WIPC of a coschedule. The average again corresponds to the zero line. The instantaneous throughput of a workload can be on average 35% higher or 35% lower than the average instantaneous throughput, depending on which coschedule of the workload is selected. This can go up to 69% higher or 56% lower for specific workloads. This confirms that the selection of coschedules can have a big impact on the throughput of a system, as shown by previous studies.

The last bar shows the variability of the average throughput for different scheduling policies. The zero line corresponds to the agnostic FCFS scheduler, the positive bar is the optimal scheduler (the average and the maximum over all workloads), and the negative bar is the worst scheduler (average and minimum). Surprisingly, the variance is much lower compared to the variance in the IPC and the instantaneous throughput: on average, the theoretically best scheduler only performs 3% better than the FCFS scheduler (and up to 12%), and the worst possible scheduler is only 9% worse than the FCFS scheduler (up to 18% worse). This is a surprising result. From the variance in the IPC of the individual jobs and the variance in the instantaneous throughput of the different coschedules, we expected a much larger impact of the scheduler on average throughput. We see similar results for the configuration with 4 cores with shared cache: the difference between the throughput of the FCFS scheduler and the optimal scheduler (6%) is much smaller than the variance in IPC (35%) and per-coschedule instantaneous throughput (48%). Increasing the number of different job types also has only a small impact on the optimal throughput: for 8 job types ($N = 8$), the average throughput increase of an optimal scheduler is only 4.5% for the SMT configuration. In the next section, we analyze this unexpected outcome.

In the remainder we define *variability* as the average spread (maximum minus minimum divided by average). So for example, for the SMT configuration (Figure 1), the variability of the per-job IPC is 37%, the variability of the per-coschedule instantaneous throughput is 69% and the variability of the average throughput is 12%.



(a) SMT configuration (slope 0.73)

(b) Quad core configuration (slope 0.56)

Fig. 2. Optimal versus FCFS average throughput for the both configurations. Each point is a distinct workload. Numbers are normalized to worst average throughput. Thick line is the bisector, thin line has slope 0.73 (a) or 0.56 (b).

C. Analysis of the results

We begin our analysis of the fact that symbiotic job scheduling has only a small impact on maximum throughput by comparing the optimal, FCFS and worst scheduler. Figure 2 shows scatter plots with one point per workload. The X-axis is the optimal throughput relative to the worst throughput. The Y-axis is the ratio of the FCFS throughput to the worst throughput. In theory, average throughput can be any point between the X-axis (average throughput equal to worst throughput) and the bisector (average throughput equal to optimal throughput). The figure shows that the throughput of the FCFS scheduler is close to a line with slope 0.73 for the SMT configuration and 0.56 for the quad-core configuration. On average, the symbiosis-unaware FCFS scheduler is able to bridge 76% and 63% of the throughput gap between the worst and best schedulers for the SMT and quad-core configurations respectively. We will explain later why the throughput of the FCFS scheduler is closer to the optimal than to the worst throughput, but the fact that there is a good correlation between the throughput gain of the optimal and FCFS scheduler versus the worst scheduler (i.e., the points are close to a line) simplifies our analysis to finding out why the worst and the optimal scheduler have such a small difference in throughput.

The worst and optimal throughput are both calculated by solving a linear program. A linear program consists of a some linear restrictions and a linear objective function. Due to the

linearity, the restrictions form a convex solution space, and the optimization function has it highest (and lowest) value on the boundary of the solution space. In this context, there can be two reasons why there is a small difference between the maximum and the minimum: the optimization function has small variability within the solution space (i.e., any possible solution leads to approximately the same throughput) or the restrictions limit the solution space to a very small set of possible solutions (i.e., the restrictions force the throughput to a specific value, independent of the scheduling policy).

1) Insensitivity of the average throughput: We identified two main reasons why the average throughput may have a limited variability: job insensitivity and linear bottlenecks.

a) Job insensitivity: We call a job insensitive if its performance is not dependent on the co-running jobs. If all jobs are insensitive, there is nothing to gain with symbiotic job scheduling, because we cannot improve the performance of individual jobs by intelligently selecting coschedules. For our study, about one quarter of the workloads have a low job sensitivity (both for the SMT and quad core configuration). We indeed see an average throughput variability of less than 10% for these workloads. However, Figure 1 shows that average job sensitivity is for all configurations about three times larger than the average throughput variability, so the limited throughput variability cannot be explained by small job sensitivities only.

b) Linear bottleneck: A linear bottleneck is a situation where each job has an execution rate proportional to the fraction it gets of a certain critical and fully utilized shared resource. Formally, $r_b(s) = f_b(s) \times R_b$ where R_b is the execution rate of jobs of type b when they can use the full resource, and $f_b(s)$ is the total fraction of the shared resource used by jobs of type b in coschedule s. Since the resource is fully utilized, $\sum_{b=1}^{N} f_b(s) = 1$ and for all coschedules s,

$$\sum_{b=1}^{N} \frac{r_b(s)}{R_b} = 1. \tag{6}$$

It can be shown that in this case, the average throughput is independent of the scheduling policy and equals

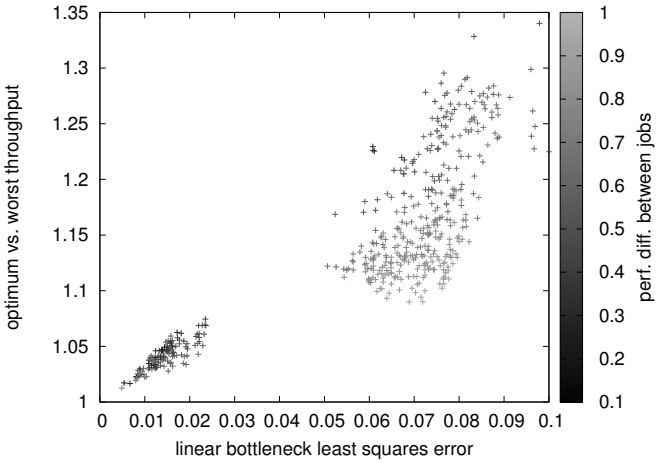
$$AT = \frac{N}{\sum_{b=1}^{N} 1/R_b}. \tag{7}$$

Examples of linear bottlenecks could be memory bandwidth, fetch bandwidth, dispatch width, a non-pipelined long-latency functional unit (e.g., a divider), etc. Note that insensitive jobs are a special case of linear bottleneck. In this case, R_b is K times the insensitive execution rate of jobs of type b, and Equation 6 holds for all coschedules.

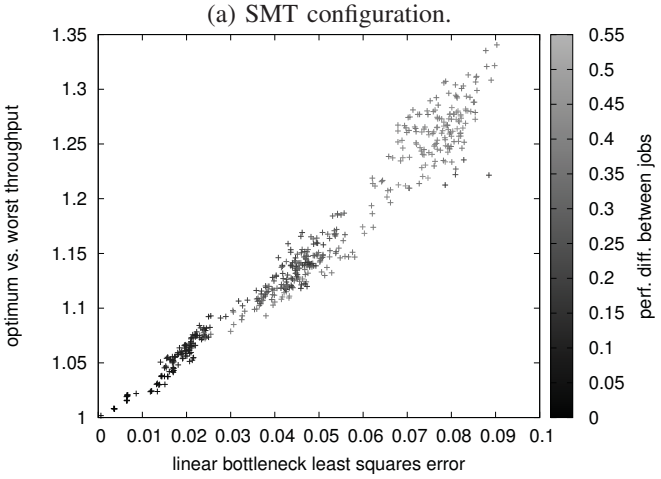
Of course, in realistic situations, Equation 6 will never be exact. To check whether we are close to a linear bottleneck, we can look for R_b's that solve the overdetermined system of equations 6 in the least-square sense, i.e., R_b's that minimize

$$\epsilon^2 = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left(\sum_{b=1}^{N} \frac{r_b(s)}{R_b} - 1 \right)^2.$$

The least-square error ϵ is then a measure of how close we are to a linear bottleneck: an error of 0 means that we have an exact linear bottleneck, and the larger the error, the further we are from a linear bottleneck. Figure 3 shows the throughput variance as a function of the linear bottleneck least-square error (the coloring of the points will be explained in the next section). We notice a fairly good correlation, and more so for the quad-core configuration than for the SMT configuration.



(a) SMT configuration.



(b) Quad core configuration.

Fig. 3. Optimal throughput versus linear bottleneck least square error for both configurations. Each point is a distinct workload. The color of the point represents the difference in average WIPC between the different job types in the workload.

We found that for the SMT configuration, mainly workloads with high-IPC jobs have a close to linear bottleneck, which could be explained by the fact that the fetch and dispatch width of the SMT core is limited to 4, so instantaneous IPC throughput can never exceed 4, and the IPC of a high-IPC benchmark is proportional to the fraction of the fetch bandwidth it gets. Memory-intensive jobs have more interference in the shared caches, which has a non-linear effect on performance. For the quad-core configuration, non-memory-intensive applications have less interference in the shared cache, making them insensitive and therefore a linear bottleneck.

2) Small solution space: The main restriction we impose on our optimal scheduler is that each job type should perform a fixed amount of work, which we assume equal for all job types (see Section III-D). This implies that if we execute (a) particular job type(s) for some amount of time, we are forced to execute the other job types later on, to let them keep up and execute their part of the work. Depending on the performance of the jobs in the coschedules, this can heavily restrict the

freedom of the scheduler to select coschedules. In extreme cases, it can occur that the workload forces the selection of the coschedules to be executed, not the scheduler.

Before giving a more formal analysis, we first explain this behavior intuitively. Let's assume that we have two job types A and B, with A a high-IPC job type and B a low-IPC job type. Coschedules with many A's will have a higher instantaneous throughput, so a scheduler might select these coschedules. However, because they have a high throughput, they will finish their work fast, and we will end up with only B's to execute, which have a low throughput. So, no matter how much effort we put in optimizing throughput, we will always have to execute low throughput coschedules to fulfill the fixed work requirement.

Analysis of the results showed that the difference between the execution rates of different job types is a good indicator of a limited solution space. The higher the difference between the performance of the individual jobs is, the more time is spent executing low-IPC jobs, and the less freedom a scheduler has in selecting coschedules to optimize throughput. We refer back to Figure 3, but now we also consider the color of points, which reflects the difference in average WIPC *between* the different job types in the workload. The figures clearly show that workloads that have a relatively large linear bottleneck least square error but a low average throughput variability, have a relatively high difference in per-job performance. The points with smaller IPC differences show good correlation between the linear bottleneck least square error and the throughput variability. Note that the differences between the WIPC of the benchmarks is much smaller for the quad-core configuration than for the SMT configuration. This is because the interference in the quad-core configuration is much smaller and more evenly divided over the different jobs (i.e., each job has a similar slowdown), whereas for the SMT configuration, there is more interference, and the interference is unequally divided, i.e., some jobs are slowed down much more than others.

3) Conclusion of the optimum versus worst average throughput analysis: In conclusion, it is clear that the average throughput variability (i.e., the difference between minimum and maximum throughput) is always smaller than the variability in per-job IPC and per-coschedule instantaneous throughput. Because of the fixed work constraint, we cannot always select the highest IPC jobs or highest throughput coschedules: jobs that do not belong to this category will be delayed and eventually the system is forced to execute them. Furthermore, average throughput is also less variable if

- Jobs are (close to) insensitive: if the performance of a job is (almost) independent of what jobs are co-running, symbiotic job scheduling has no impact.
- There is a linear bottleneck in the hardware: if the performance of every job is proportional to the fraction of a critical shared resource that the job gets (e.g., dispatch width), then average throughput is fixed, independent of the scheduling policy.
- There is a large difference in execution rate between different jobs in the same workload: slow jobs take longer to execute and will dominate the bulk of the execution profile, meaning that the scheduler has less freedom to select different coschedules.

TABLE II. INSTANTANEOUS THROUGHPUT AND FRACTION OF TIME COSCHEDULES ARE SELECTED AVERAGED PER NUMBER OF UNIQUE JOB TYPES PER COSCHEDULE.

(a) SMT configuration

Coschedule heterogeneity	Average inst. throughput	Frac. FCFS scheduler	Frac. optimal scheduler	Frac. worst scheduler
1	1.74	3%	1%	80%
2	1.83	38%	38%	20%
3	1.91	52%	50%	0%
4	1.97	7%	11%	0%

(b) Quad-core configuration

Coschedule heterogeneity	Average inst. throughput	Frac. FCFS scheduler	Frac. optimal scheduler	Frac. worst scheduler
1	3.36	2%	1%	65%
2	3.40	34%	10%	35%
3	3.46	55%	17%	0%
4	3.53	9%	72%	0%

If any of these conditions applies, then throughput variability is low, explaining the large fraction of workloads that have low throughput variability: for the SMT configuration, about 30% of the workloads have a variability of less than 10%. On the other hand, 15% of the workloads have throughput variability of more than 20%. Note that this is the variability between the worst and the best scheduler. The difference between the symbiosis-unaware FCFS scheduler and the optimal scheduler is discussed next.

D. Why FCFS throughput is close to the maximum throughput

The previous sections explained why the difference between minimum and maximum throughput is low for many workloads, but the difference between the FCFS scheduler and the optimal scheduler is even smaller. Figure 2 shows that the FCFS scheduler fills about 60% to 70% of the gap between the worst and best scheduler. In numbers, this means that of the average 12% to 14% throughput difference between the worst and best scheduler, 8% to 9% is already obtained by using a symbiosis-unaware FCFS scheduler instead of a deliberately bad scheduler, leaving only a small 3% to 6% potential throughput gain for a symbiotic job scheduler compared to the FCFS scheduler on average.

To explain this behavior, we analyzed the coschedules that are selected by the FCFS, optimal and worst scheduler. Table II lists these fractions, grouped by the coschedule heterogeneity, i.e., the number of different job types in the coschedules. The tables also list the average instantaneous throughput (WIPC) of these coschedules. Clearly, the higher the coschedule heterogeneity, the higher the instantaneous throughput. This is because similar programs stress the same component(s) of the processor, which results in a bottleneck. Different job types stress different components, resulting in better utilization and a higher throughput.

The coschedules selected by the FCFS scheduler result from a random process, where the next job is uniformly selected from the available job types. For example, the probability of a homogeneous coschedule (only one job type) equals the probability that 4 consecutively drawn jobs have the same type. These probabilities equal 2%, 33%, 56% and 9%, for coschedule heterogeneities of 1, 2, 3 and 4, respectively. The FCFS coschedule time fractions differ a little bit from these theoretical values, due to the fact that some jobs run longer than other jobs, which means that the slower jobs remain longer in the system than faster jobs.

The optimal scheduler clearly tries to execute the better performing heterogeneous schedules more often, but succeeds

in that far better for the quad-core configuration than for the SMT configuration. The cause for this is the fact that for the quad-core configuration, the interference is much less than for the SMT configuration. For the quad-core configuration, jobs run at (close) to full speed if there is not much common stress on components (i.e., the shared cache and memory bus), which is the case in the heterogeneous coschedules. This means that it is easy to fulfill the equal work constraint: all the jobs have (approximately) the same execution rate, and we only need to select other coschedules to let job types that are a little bit behind keep up with the other job types.

For the SMT configuration, there is much more interference, and the interference is not fairly distributed, i.e., some jobs progress faster than others. This means that the optimal scheduler can only select the heterogeneous coschedule for a short fraction of time, and then needs to select other coschedules to let the slower jobs keep up. This forces the optimal scheduler to be closer to the FCFS scheduler. The worst scheduler, on the other hand, mostly selects badly performing homogeneous coschedules. It has no problems to fulfill the equal-work constraint, because selecting the 4 homogeneous coschedules always enables satisfying this constraint.

To conclude, the average throughput of the FCFS scheduler is much higher than that of the worst scheduler, as FCFS 'naturally' selects homogeneous coschedules for a small fraction of time, while the worst scheduler selects them for a large fraction of time. For the quad-core configuration, the optimal scheduler can select the best performing heterogeneous coschedules most of the time, because coscheduled jobs progress roughly at the same rate, resulting in a throughput that is relatively higher than that of the FCFS scheduler. For the SMT configuration, there is less fairness between coscheduled jobs, forcing the optimal scheduler to select other, worse performing coschedules to satisfy the equal work constraint, which results in a throughput that is only slightly above that of the FCFS scheduler. To check this statement, we artificially changed the performance of jobs in the single 4-heterogeneous coschedule for the SMT configuration by making them more fairly distributed, without changing the instantaneous throughput of the coschedule (i.e., we gave slower jobs a higher IPC and faster jobs a lower IPC). Now, the optimal scheduler selects the heterogeneous coschedule for most of the time, which increases the average throughput substantially, while the average throughput of the FCFS and worst schedulers remains unchanged.

VI. Symbiotic Job Scheduling and Turnaround Time

The results in the previous section seem to contradict results from published symbiotic job scheduling papers. For example, Snavely and Tullsen [33] report a 17% reduction in turnaround time by using their symbiotic scheduler for a four-way SMT core, and Eyerman and Eeckhout [10] find that turnaround time is reduced by on average 21% for a similar setup. This is much larger than the 3% increase in average throughput that we obtain using a perfect optimal scheduler. The main difference is the reported metric: previous work reports *turnaround time* reductions in a *latency experiment*, whereas we report the *average throughput* using a *maximum throughput experiment*. The difference between both is explained and analyzed in this section.

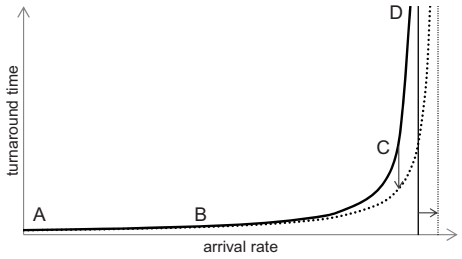


Fig. 4. Turnaround time as a function of arrival rate. The vertical line is the asymptote at the maximum throughput, the dotted lines correspond to a small increase in maximum throughput.

Turnaround time is the time that has elapsed between the entrance of a job in the system until it is fully executed and leaves the system. Turnaround time depends on the arrival rate of the jobs and the service rate of the servers: the higher the arrival rate and the lower the service rate, the larger the turnaround time, because a job will spend more time waiting in the queue. Figure 4 shows a (generic) curve of the turnaround time as a function of the job arrival rate (for the moment, we only consider the full line, we will discuss the dotted line later). If the arrival rate is very low (point A on the curve), there will be no jobs in the queue or in the processor when a job arrives, so its turnaround time is inversely proportional to the service rate of one server. When the arrival rate is slightly larger (point B), multiple jobs will be in the system concurrently, but the queue is usually empty. Turnaround time increases because of interference between jobs due to resource sharing. A further increase in the arrival rate (point C) causes some jobs to be queued for some amount of time. Turnaround time increases quickly due to the extra queuing delay. When the arrival rate is close to the maximum service rate (point D), the turnaround time increases very fast, and becomes infinite when the arrival rate exceeds the maximum service rate, because jobs arrive faster than they can be processed, leading to an infinitely large queue. In our setup, the maximum service rate corresponds to the maximum throughput of the processor as calculated in the maximum throughput experiment.

When the operating point of a processor is in points A and B, scheduling has no impact because there are no jobs in the queue to select specific coschedules. The coschedules are purely determined by the jobs that happen to be in the system at that time. Usually, point D is avoided, since this leads to a large amount of queued jobs, which would require a large queuing capacity or lost jobs, together with very long turnaround times. The most interesting point is point C, where the processor is fully loaded most of the time, and some jobs are queued. Here, an intelligent scheduler can try to reduce turnaround times by selecting coschedules from the jobs currently in the system (both in the servers and the queue). This is also the operating point of the experiments done by Snavely and Eyerman: they both strive at having approximately twice as many jobs in the system than the number of thread contexts.

Assume the full line in Figure 4 corresponds to the FCFS scheduler, and the arrival rate corresponds to point C. Using symbiosis-aware scheduling, we slightly increase maximum throughput, ending up with the dotted line. It is clear that if the arrival rate remains the same, the relative reduction in turnaround time is much larger than the relative increase in maximum throughput, explaining the discrepancy between our results and those in previous work. For example, for an M/M/4

queuing system (4 servers and exponentially distributed inter-arrival times and service times), with parameters $\lambda = 3.5$ and $\mu = 1$, there are on average 8.7 jobs in the system, and the turnaround time is 2.5. Increasing μ to 1.03 (3% increase in maximum throughput) results in 7.3 jobs in the system and a turnaround time of 2.1, a 16% reduction.

As a result, turnaround time reduction numbers are related to the arrival rate or the load presented to the server. If the load is high, significant decreases in turnaround time can be achieved with only small increases in throughput. Furthermore, turnaround time can also be improved without increasing throughput. For a single-processor server, always selecting the jobs with the shortest remaining time (the SRPT scheduling policy [15]) results in the lowest average turnaround time [30]. This is intuitive: putting a long job before a short one substantially increases the turnaround time of the short job, while doing it the other way around only slightly increases the turnaround time of the long job. To conclude, for a microarchitecture or scheduler study that targets increasing throughput (such as the studies by Snavely and Eyerman), (only) reporting average turnaround time reductions can be misleading, since these reductions can be magnified by increasing the load and/or by reordering the jobs in an SRPT-like manner (e.g., if jobs have approximately the same size, selecting the jobs with the highest execution rates amounts to selecting the jobs with the shortest execution times).

Maximum throughput is independent of the arrival rate and job size based ordering. However, most systems do not operate at their maximum throughput point, because that results in long queues and long turnaround times. The system is usually dimensioned such that the maximum throughput is larger than the (expected) load, to limit queue sizes and to avoid high turnaround times at peak moments. In that case, average throughput is always equal to the average load, and therefore, different schedulers cannot be compared by evaluating their throughput. The problem is that the throughput improvement only has an impact when the system is actually loaded, and if the load is less than the maximum throughput, the system becomes empty from time to time. When the system is loaded, throughput improvements will lead to a shorter time to execute the current load, thereby decreasing the processor utilization and enlarging the empty periods. So, instead of reporting throughput (which is constant) or turnaround time (which can be misleading), it is better to evaluate processor utilization (average number of cores that are executing jobs) and/or empty periods. These numbers are still impacted by the load, but they remain bounded when the load is increased.

To illustrate this and to show that it is possible to increase throughput using information from the theoretically optimal scheduler, we did the following experiment. We simulated a system where jobs arrive following a Poisson process with an arrival rate that is lower than the maximum throughput, and we measured turnaround time, processor utilization and the fraction of time the system is empty. We assume we know the performance of all jobs in all coschedules, and we know the distribution of job types and job sizes. We evaluated 4 schedulers:

- FCFS scheduler: jobs are executed in the order they arrive.
- MAXIT scheduler: from all jobs in the system, select the combination (coschedule) that has the highest instantaneous throughput; if multiple combinations have the same throughput, select the one containing the oldest jobs.
- SRPT scheduler: from all possible combinations of jobs, select the one with the smallest sum of remaining execution times (taking into account the remaining job length and its performance when coscheduled in that particular combination).
- MAXTP scheduler: get the optimal coschedules and their fraction of time from the linear programming model (offline phase); when one or more of these coschedules can be composed from the jobs in the system, select the one that is furthest away from its ideal time fraction (and thus also keep a counter per coschedule with the time it has been selected up to now); if none of these coschedules can be selected, resort to the MAXIT scheduler. This is a practical implementation of a scheduler that makes use of the methodology presented in this paper.

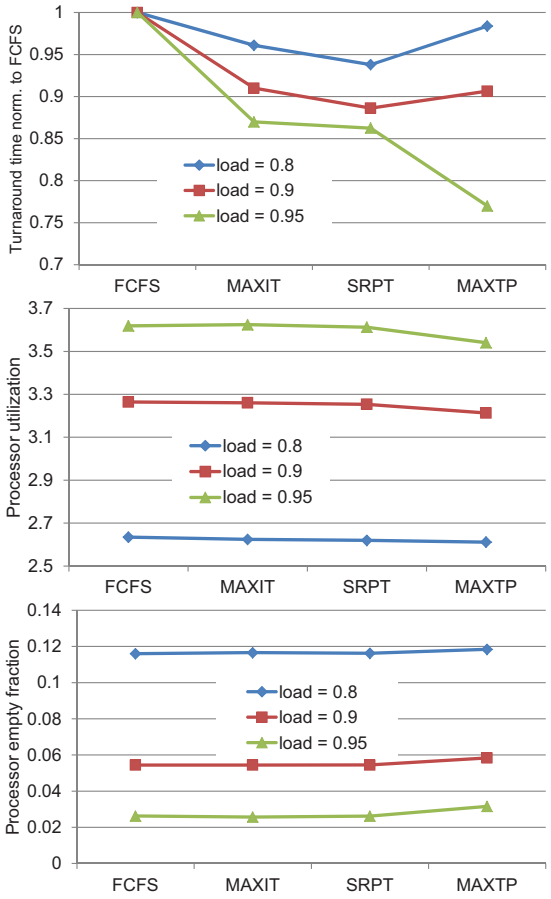


Fig. 5. Turnaround time, processor utilization and processor empty fraction for the FCFS, MAXIT, SRPT and MAXTP scheduler, and three different loads (relative to the FCFS maximum throughput).

Note that the FCFS scheduler needs no knowledge about the jobs. The MAXIT scheduler needs to know the instantaneous throughput of all coschedules (using offline profiling, a model [10], or information from the past). The SRPT scheduler needs this information too, together with the (estimated) size of the jobs in the system. The MAXTP scheduler has an offline phase, which requires the performance of each of the coschedules, and the distribution of job types and sizes. Once the optimal coschedules and their fractions are known,

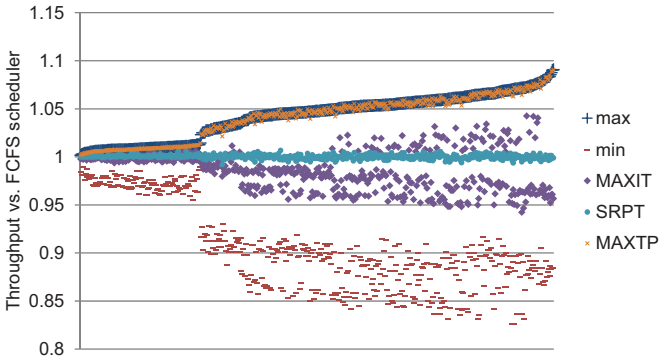


Fig. 6. Obtained throughput on a maximum throughput experiment (arrival rate > maximum throughput) for the MAXIT, SRPT and MAXTP scheduler, relative to the FCFS scheduler. The theoretical maximum and minimum throughput is also shown. Each point corresponds to a workload and points are ordered on increasing maximum throughput.

it just follows this schedule, without needing any knowledge about the size or the performance of the current jobs. Only the types of the current jobs need to be communicated to the scheduler. Note also that this is a theoretical study, with no intention of evaluating a real scheduler. It just shows the potential (the performance of a realistic scheduler will be impacted by context switch overhead effects, inaccurate modeling of the performance and/or inaccurate extrapolation of distributions seen in the past, effects that are not modeled in this experiment).

Figure 5 shows the results for the four schedulers. These results are averaged over all workloads consisting of 4 job types, out of the 12 considered benchmarks. For every workload, we did three experiments with three different arrival rates, relative to the FCFS maximum throughput (calculated by TPCalc [11]). For arrival rates that are 80% and 90% of the FCFS maximum throughput, SRPT results in the lowest turnaround time, but it has no considerable impact on processor utilization and on the empty fraction. Although not very visible, the lowest utilization and highest empty fraction is obtained by the MAXTP scheduler, despite the higher turnaround time. If the load increases to 95%, the MAXTP scheduler has more opportunities to select optimal coschedules, resulting in a 23% reduction of the turnaround time, which is much higher than the 3% increase in maximum throughput, as explained before. The MAXTP scheduler also has a lower processor utilization (-2.2% versus FCFS) and a higher empty fraction (from 2.6% for FCFS to 3.2% for MAXTP), which are in the same order as the maximum throughput increase.

We also performed a maximum throughput experiment by setting the arrival rate higher than the maximum throughput. Figure 6 shows the obtained throughput for all workloads, relative to the FCFS throughput. It also includes the theoretical maximum and minimum throughput using the linear programming model (Section IV). The SRPT scheduler has the same maximum throughput as the FCFS scheduler. The MAXIT scheduler has a slightly lower throughput on average, mainly because it delays badly performing jobs too much. The MAXTP scheduler has a throughput that almost exactly matches the maximum throughput obtained by the linear programming model.

VII. Optimal Throughput as a Throughput Metric

Although our results show that the impact of scheduling on maximum throughput is rather limited, the increase in throughput (3% to 10%) is similar to that of (small) microarchitectural improvements. Microarchitecture studies usually do not evaluate the impact of scheduling on the performance improvement of the proposed feature. However, the baseline processor with intelligent scheduling can possibly perform as good or even better than the improved processor. The linear programming technique as described in Section IV can be used to evaluate the impact of a theoretically optimal scheduler on average throughput. It needs only performance data from individual coschedules to calculate the optimal throughput, without requiring a real throughput experiment or designing new scheduling algorithms.

To illustrate the usage of optimal throughput as a metric in microarchitecture studies, we explore different fetch and resource sharing policies for an SMT core. We evaluated four different settings: a round-robin (RR) fetch policy and static ROB partitioning [27], RR fetch policy and dynamic ROB partitioning, ICOUNT fetch policy and static ROB partitioning, and ICOUNT with dynamic ROB partitioning [37] (which was the default setting in the previous results).

For all policies, we evaluate the average throughput of the FCFS scheduler [11] and the optimal scheduler for all workloads consisting of 4 different benchmarks. We find that the scheduler has no significant impact on the conclusion of the comparison, ICOUNT with dynamic ROB partitioning outperforms all other policies. On average, it has a 1.7% higher throughput than RR with static partitioning for the FCFS scheduler, and 1.5% for the optimal scheduler. For individual workloads, scheduling can have an impact on the conclusion: about 10% of the workloads select a different optimal policy when using an optimal scheduler compared to the standard FCFS scheduler. Also note that the throughput of the RR with static partitioning policy is improved by 1.7% by using a more complex policy, while intelligent scheduling improves the average throughput by 3.3% (assuming that we can reach this upper bound with a realistic scheduler).

VIII. Conclusions

In this paper, we explore the impact of symbiotic job scheduling on the maximum throughput of a fully symmetric SMT or multicore processor. We find that, despite the fact that job performance is very sensitive to which other jobs are coscheduled, symbiotic job scheduling has only a small impact on the maximum throughput of a system. The main reason is that in the end, we have to execute all jobs in a system, so selecting jobs that perform well and have a good symbiosis delays the other jobs and eventually forces the system to execute bad job combinations.

We also show that a small increase in maximum throughput can have a large impact on turnaround time, which explains previous results. We argue that only reporting turnaround time can lead to misleading conclusions because it depends on the arrival rate of jobs, while processor utilization or processor empty time are a better indicator for throughput improvement. Finally, we show how the impact of intelligent scheduling can be evaluated in a microarchitecture study, without having to implement a scheduler or simulating an arrival process.

Overall, our study shows that the impact of symbiotic scheduling on the average throughput of fully symmetric SMT or multicore processors is *likely* to be limited. However, our study does *not* say that trying to exploit application symbiosis is worthless in general. Application symbiosis might bring non-negligible throughput improvements on particular workloads (more than 10% for our study), even on fully symmetric multicores. More importantly, previous studies have shown that the instantaneous throughput of partially symmetric multicores (e.g., multicores with SMT cores) may be boosted substantially by exploiting application symbiosis for intelligent job-to-core mapping.

ACKNOWLEDGEMENTS

We thank the reviewers for their constructive and insightful feedback. Stijn Eyerman and Wouter Rogiest are postdoctoral fellows of the Research Foundation – Flanders (FWO). This work is supported in part by the European Research Council Advanced Grant DAL No. 267175.

REFERENCES

[1] M. Banikazemi, D. Poff, and B. Abali, "PAM: a novel performance/power aware meta-scheduler for multi-core systems," in *Supercomputing (SC)*, 2008.

[2] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. of the 3rd Conf. on Computing Frontiers (CF)*, 2006.

[3] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[5] J. R. Bulpin and I. A. Pratt, "Hyper-Threading aware process scheduling heuristics," in *Proc. of the USENIX Annual Technical Conference*, 2005.

[6] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, Aug. 2014.

[7] T. Creech, A. Kotha, and R. Barua, "Efficient multiprogramming for multicores with SCAF," in *MICRO*, 2013.

[8] M. DeVuyst, R. Kumar, and D. M. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors," in *IPDPS*, 2006.

[9] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *IPDPS*, 2006.

[10] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *ASPLOS*, 2010.

[11] S. Eyerman, P. Michaud, and W. Rogiest, "Multi-program throughput metrics: a systematic approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, Sep. 2014.

[12] S. Eyerman and L. Eeckhout, "The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism," in *ASPLOS*, 2014.

[13] A. Fedorova, M. Seltzer, and M. D. Smith, "A non-work-conserving operating system scheduler for SMT processors," in *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2006.

[14] D. G. Feitelson, "Job scheduling in multiprogrammed parallel systems," IBM T. J. Watson Research Center, IBM Research Report RC 19790 (87657), 1997.

[15] M. Harchol-Balter, *Performance modeling and design of computer systems - Queuing theory in action*. Cambridge University Press, 2013.

[16] R. Jain, C. J. Hughes, and S. V. Adve, "Soft real-time scheduling on simultaneous multithreaded processors," in *Proc. of the IEEE Int. Real-Time Systems Symp. (RTSS)*, 2002.

[17] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT*, 2008.

[18] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. of the 5th European Conf. on Computer Systems (EuroSys)*, 2010.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA*, 2004.

[20] E. L. Lawler and J. Labetoulle, "On preemptive scheduling of unrelated parallel processors by linear programming," *Journal of the ACM*, vol. 25, no. 4, pp. 612–619, Oct. 1978.

[21] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *IPDPS*, 2005.

[22] A. Merkel, J. Stoess, and F. Bellosa, "resource-conscious scheduling for energy efficiency on multicore processors," in *Proc. of the 5th European Conf. on Computer Systems (EuroSys)*, 2010.

[23] L. J. Miller, "A heterogeneous multiprocessor design and the distributed scheduling of its task group workload," in *ISCA*, 1982.

[24] H. Najaf-abadi and E. Rotenberg, "The importance of accurate task arrival characterization in the design of processing cores," in *Proc. of the IEEE Int. Symp. on Workload Characterization (IISWC)*, 2009.

[25] J. Nakajima and V. Pallipadi, "Enhancement for Hyper-Threading technology in the operating system - seeking the optimal scheduling," in *Proc. of the 2nd Workshop on Industrial Experiences with Systems Software*, 2002.

[26] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," University of Washington, Tech. Rep. UW-CSE-00-04-02, 2000.

[27] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors," in *PACT*, 2003.

[28] 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 *ASPLOS*, 2012.

[29] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proc. of the 5th European Conf. on Computer Systems (EuroSys)*, 2010.

[30] L. Schrage, "A proof of the optimality of the shortest processing remaining time discipline," *Operation Research*, vol. 16, no. 3, pp. 687–690, 1968.

[31] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale," in *NSDI*, 2006.

[32] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural support for enhanced SMT job scheduling," in *PACT*, 2004.

[33] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *ASPLOS*, 2000.

[34] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *SIGMETRICS*, 2002.

[35] K. Tian, Y. Jiang, X. Shen, and W. Mao, "Optimal co-scheduling to minimize makespan on chip multiprocessors," in *Proc. of the 16th Workshop on Job Scheduling Strategies for Parallel Processing*, 2012, LNCS 7698.

[36] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading Pentium 4 processor," in *PACT*, 2003.

[37] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ISCA*, 1996.

[38] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*, 2012.

[39] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *PACT*, 2010.

[40] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, vol. 45, no. 1, Nov. 2012.