Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling

Stijn Eyerman Lieven Eeckhout
ELIS Department, Ghent University, Belgium
Email: {seyerman,leeckhou}@elis.UGent.be

Abstract

Symbiotic job scheduling boosts simultaneous multithreading (SMT) processor performance by co-scheduling jobs that have 'compatible' demands on the processor's shared resources. Existing approaches however require a sampling phase, evaluate a limited number of possible co-schedules, use heuristics to gauge symbiosis, are rigid in their optimization target, and do not preserve system-level priorities/shares.

This paper proposes probabilistic job symbiosis modeling, which predicts whether jobs will create positive or negative symbiosis when co-scheduled without requiring the co-schedule to be evaluated. The model, which uses per-thread cycle stacks computed through a previously proposed cycle accounting architecture, is simple enough to be used in system software. Probabilistic job symbiosis modeling provides six key innovations over prior work in symbiotic job scheduling: (i) it does not require a sampling phase, (ii) it readjusts the job co-schedule continuously, (iii) it evaluates a large number of possible co-schedules at very low overhead, (iv) it is not driven by heuristics, (v) it can optimize a performance target of interest (e.g., system throughput or job turnaround time), and (vi) it preserves system-level priorities/shares. These innovations make symbiotic job scheduling both practical and effective.

Our experimental evaluation, which assumes a realistic scenario in which jobs come and go, reports an average 16% (and up to 35%) reduction in job turnaround time compared to the previously proposed SOS (sample, optimize, symbios) approach for a two-thread SMT processor, and an average 19% (and up to 45%) reduction in job turnaround time for a four-thread SMT processor.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General—Modeling of Computer Architecture; C.1.4 [*Computer Systems Organization*]: Processor Architectures—Parallel Architectures; C.4 [*Computer Systems Organization*]: Performance of Systems—Modeling Techniques

General Terms Design, Experimentation, Measurement, Performance

Keywords Simultaneous Multithreading (SMT), Symbiotic job scheduling, Performance modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03. . . $5.00

1. Introduction

Simultaneous multithreading (SMT) processors [31, 32], such as the Intel Core i7, IBM POWER7, Sun Niagara and Rock, seek at improving microprocessor utilization by sharing hardware resources across multiple active threads. Shared hardware resources however may affect system performance in unpredictable ways. Co-executing jobs may conflict with each other on various shared resources which may adversely affect overall performance. Or, reversely, system performance may greatly benefit from co-executing jobs that put 'compatible' demands on shared resources. In other words, the performance interactions and symbiosis between co-executing jobs on multithreaded processors can be positive or negative [28].

Because of the symbiosis among co-executing jobs, it is important that system software (the operating system or the virtual machine monitor) makes appropriate decisions about what jobs to co-schedule in each timeslice on a multithreaded processor. Naive scheduling, which does not exploit job symbiosis, may severely limit the performance enhancements that the multithreaded processor can offer. Symbiotic job scheduling on the other hand aims at exploiting positive symbiosis by co-scheduling independent jobs that 'get along', thereby increasing system throughput and decreasing job turnaround times. The key challenge in symbiotic job scheduling however is to predict whether jobs will create positive or negative symbiosis when co-scheduled. Previously proposed symbiotic job scheduling approaches [24, 25, 26] take a pragmatic approach. They sample the space of possible co-schedules, i.e., they select and execute a limited number of job co-schedules, and then retain the most effective one according to some heuristic(s). This pragmatic approach however may lead to suboptimal co-schedules and thus suboptimal system performance.

This paper proposes probabilistic job symbiosis modeling. The probabilistic model uses probabilistic theory to estimate single-threaded progress rates for the individual jobs in a job co-schedule without requiring its evaluation. The model uses as input per-thread cycle stacks (which are computed using our previously proposed cycle accounting architecture [12]), and is simple enough so that system software can evaluate all or at least a very large number of possible job co-schedules at low overhead, and search for optimum job co-schedules. Probabilistic modeling and model-driven symbiotic job scheduling provides six major innovations over prior work: (i) it does not require a sampling phase to evaluate symbiosis — instead, symbiosis is predicted by the probabilistic model; (ii) it readjusts the job co-schedule continuously; (iii) it enables evaluating all (or at least a very large number of) possible job co-schedules at very low overhead; (iv) it does not rely on heuristics to gauge symbiosis but instead tracks single-threaded progress rates for all jobs in the job mix; (v) because it estimates single-threaded

progress rates for each job co-schedule, system software can estimate and optimize an SMT performance target of interest such as system throughput, or job turnaround time, or a combination of both — prior approaches on the other hand are rigid in their optimization target; and (vi) when extended with the notion of system-level priorities/shares, it enables preserving shares as expected by end users while exploiting job symbiosis — a property that was not achieved in prior work by Snavely et al. [26]. In summary, probabilistic job symbiosis modeling makes symbiotic job scheduling both practical and more effective.

Our experimental results demonstrate the accuracy of probabilistic job symbiosis modeling on simultaneous multithreading (SMT) processors. The probabilistic model achieves an average absolute prediction error of 5.5% for a two-thread SMT processor and 8.8% for a four-thread SMT processor for the ICOUNT fetch policy [31]. In addition, we demonstrate the applicability of the probabilistic model across different SMT processor resource partitioning strategies and fetch policies, and report average absolute prediction errors similar to ICOUNT for static partitioning [20] (2.9%), flush [30] (5.8%), MLP-aware flush [10] (4.2%) and DCRA [4] (4.5%).

Model-driven job scheduling which leverages probabilistic symbiosis modeling achieves an average reduction in job turnaround time of 16% over the previously proposed SOS (Sample, Optimize, Symbios) proposal [26], and 21% over naive round-robin job scheduling for two-thread SMT processors; for particular job mixes, we report reductions in job turnaround time by up to 35% over SOS. For a four-program SMT processor, we report an average reduction in job turnaround time by 19% on average and up to 45% compared to SOS. Finally, we demonstrate that symbiotic job scheduling achieves proportional sharing: jobs achieve as much progress as they are entitled to, while at the same time exploiting job symbiosis.

This paper makes two major contributions.

- We propose probabilistic job symbiosis modeling for SMT processors (Section 5). The model takes as input a cycle stack for each of the individual jobs when run in isolation (which are measured by our recently proposed per-thread cycle accounting architecture [12]), and predicts progress for each job in a job co-schedule. To the best of our knowledge, this is the first analytical performance model to target SMT processors.
- We apply probabilistic modeling to symbiotic job scheduling (Sections 6 and 7) and demonstrate that it significantly improves over prior proposals. We provide a comprehensive analysis as to where the performance improvement comes from.

This paper is organized as follows. Before describing probabilistic job symbiosis modeling in great detail in Section 5, we first need to provide some background on SMT performance metrics (Section 2), prior work in symbiotic job scheduling (Section 3) and per-thread cycle accounting (Section 4). Sections 6 and 7 present model-driven job scheduling without and with system-level priorities/shares, respectively. After explaining our experimental setup in Section 8, we then evaluate probabilistic symbiosis modeling and model-driven job scheduling (Section 9). Finally, we discuss related work (Section 10) and conclude (Section 11).

2. SMT performance metrics

In order to understand the optimization target of symbiotic job scheduling, we need to have appropriate metrics that characterize SMT performance. In our prior work [11], we identified two primary performance metrics for multi-program workloads: system throughput (STP), a system-oriented performance metric, and average normalized turnaround time (ANTT), a user-oriented performance metric.

System throughput (STP) quantifies the number of jobs completed per unit of time by the system, and is defined as

$$STP = \sum_{i=1}^{n} \frac{C_i^{st}}{C_i^{smt}},$$

with n jobs in the job mix, and C_i^{st} and C_i^{smt} the execution time for job i under single-threaded (ST) execution and multi-threaded (MT) execution, respectively. Intuitively speaking, STP quantifies the accumulated single-threaded progress of all jobs in the job mix under multithreaded execution. STP equals weighted speedup as defined by Snavely and Tullsen [25], and is a higher-is-better metric.

Average normalized turnaround time (ANTT) quantifies the time between submitting a job to the system and its completion. ANTT is defined as

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{C_i^{smt}}{C_i^{st}},$$

and quantifies the average user-perceived slowdown during multi-threaded execution compared to single-threaded execution. ANTT is a smaller-is-better performance metric, and equals the reciprocal of the hmean metric proposed by Luo et al. [17].

Optimizing STP has a positive impact on ANTT in general, and vice versa. For example, improving system throughput implies that new jobs can be executed faster, leading to shorter job turnaround times. However, STP and ANTT may also represent conflicting performance criteria, i.e., optimizing one of the two performance metrics may have a negative impact on the other. For example, giving priority to jobs that experience little slowdown during multi-threaded execution compared to single-threaded execution, may optimize system throughput but may have a detrimental impact on job turnaround time, and may even lead to the starvation of jobs that experience substantial slowdown during multi-threaded execution.

3. Prior work in symbiotic job scheduling on SMT processors

Snavely and Tullsen [25] pioneered the work on symbiotic job scheduling for simultaneous multithreading (SMT) processors, and proposed the SOS approach; SOS stands for Sample, Optimize and Symbios. For explaining SOS, we first need to define some terminology. A *job co-schedule* refers to multiple independent jobs co-executing on the multithreaded processor during a timeslice, i.e., the jobs in a co-schedule compete with each other for processor resources on a cycle-by-cycle basis during their scheduled timeslice. A *schedule* refers to a set of job co-schedules such that each job in the schedule appears in an equal number of job co-schedules; a schedule thus spans multiple timeslices. During the sample phase, SOS permutes the schedule periodically, changing the jobs that are co-scheduled. While doing so, SOS collects various hardware performance counter values such as IPC, cache performance, issue queue occupancy, etc. to estimate the goodness or level of symbiosis of each schedule. After the sampling phase, SOS selects the schedule with the highest symbiosis — the optimize phase — and then runs the selected schedule for a number of timeslices — the symbios phase. SOS goes into sampling mode again when a new job comes in or when a job leaves the system or when a symbiosis timer expires. SOS guarantees a level of fairness by making sure each job in the schedule appears in an equal number of job co-schedules. The goodness of symbiosis is a predictor (heuristic) for system throughput.

In their follow-on work, Snavely et al. [26] studied symbiotic job scheduling while taking into account priorities. They propose four mechanisms for supporting priorities: (i) a naive mechanism that assumes that all jobs make equal progress when co-scheduled, (ii) a more complex mechanism that strives at exploiting symbiosis when co-scheduling jobs, (iii) a hardware mechanism that gives more resources to high priority jobs, and (iv) a hybrid hardware/software mechanism. Although these mechanisms improve SMT performance at varying degrees, they do not preserve the notion of system-level shares as expected by end users (see Figures 4 through 6 in [26]) — end users expect single-threaded progress to be proportional to their relative shares. The fundamental reason why these prior mechanisms do not preserve the user-expected notion of proportional sharing is that they are unable to track single-threaded progress during multi-threaded execution.

4. Per-thread cycle accounting

Probabilistic job symbiosis modeling, as we will explain in the next section, uses as input a cycle stack for each job. We rely on our previously proposed per-thread cycle accounting architecture [12] for computing per-thread cycle stacks on SMT processors. A per-thread cycle stack is an estimate for the single-threaded cycle stack had the thread been executed in isolation. The cycle accounting architecture computes per-thread cycle stacks while jobs co-run on the processor.

The cycle accounting architecture accounts each cycle to one of the following three cycle counts, for each thread i in the job co-schedule:

- **base cycle** count $C^{smt}_{B,i}$: the processor dispatches instructions (i.e., is making progress) for the given thread;
- **miss event cycle** counts: the processor consumes cycles handling miss events (cache misses, TLB misses and branch mispredictions) for the given thread — we make a distinction between miss event cycles due to L1 I-cache misses ($C^{smt}_{L1I,i}$), L1 D-cache misses ($C^{smt}_{L1D,i}$), I-TLB misses ($C^{smt}_{ITLB,i}$), D-TLB misses ($C^{smt}_{DTLB,i}$), L2 and L3 I- and D-misses ($C^{smt}_{L2D,i}$, $C^{smt}_{L2I,i}$, $C^{smt}_{L3D,i}$, $C^{smt}_{L3I,i}$), branch mispredictions ($C^{smt}_{br,i}$), and other resource stalls due to long-latency instructions ($C^{smt}_{other,i}$);
- **waiting cycle** count $C^{smt}_{W,i}$: the processor is dispatching instructions for another thread and can therefore not make progress for the given thread. The waiting cycle count thus quantifies the number of cycles during which the processor does not make progress for thread i because of multi-threaded execution.

The cycle accounting architecture also computes two sources of reduced performance under multi-threaded execution compared to single-threaded execution. First, it quantifies the reduction in per-thread memory-level parallelism (MLP), or the number of outstanding long-latency memory requests (load misses and D-TLB misses). Multi-threaded execution exposes less per-thread memory-level parallelism than single-threaded execution because there are fewer reorder and issue buffer resources available per thread, and thus the hardware can expose fewer outstanding memory requests per thread. The cycle accounting architecture measures the amount of per-thread MLP under multi-threaded execution (MLP^{smt}_i) and also estimates the amount of MLP under single-threaded execution (MLP^{st}_i). The MLP ratio $R_{MLP,i}$ is defined as $R_{MLP,i} = MLP^{st}_i/MLP^{smt}_i$ and quantifies the reduction in per-thread MLP due to multithreading. Second, the cycle accounting architecture estimates the number of additional conflict misses due to sharing in the branch predictor, caches and TLBs. The ratio of the number of per-thread misses under multi-threaded execution divided by the (estimated) number of misses under single-threaded execution is denoted as $R_{br,i}$ for the branch mispredictions, $R_{L1I,i}$ for the L1 I-cache misses, $R_{L1D,i}$ for the L1 D-cache misses, etc. These ratios estimate the increase in the number of misses due to multi-threading.

Given the above cycle counts and ratios, the cycle accounting architecture can estimate single-threaded cycle stacks and execution times. This is done by dividing the above cycle counts with the respective ratios. For example, the branch misprediction cycle count is estimated as the multi-threaded cycle count divided by the branch misprediction ratio, i.e., $\tilde{C}^{st}_{br,i} = C^{smt}_{br,i}/R_{br,i}$. The single-threaded L3 D-cache miss cycle count takes into account the reduction in per-thread MLP as well, and is computed as $\tilde{C}^{st}_{L3D,i} = C^{smt}_{L3D,i}/(R_{L3D,i} \cdot R_{MLP,i})$. Further, the base cycle count under single-threaded execution equals the base cycle count under multi-threaded execution, i.e., $\tilde{C}^{st}_{B,i} = C^{smt}_{B,i}$. The sum of the estimated single-threaded cycle counts is an estimate for the single-threaded execution time $\tilde{C}^{st}_i$.

The cycle accounting architecture incurs a reasonable hardware cost (around 1KB of storage) and estimates single-threaded execution times accurately with average prediction errors around 7.2% for two-program workloads and 11.7% for four-program workloads. We refer the interested reader to [12] for more details.

5. Probabilistic job symbiosis modeling

Starting from a cycle stack for each job, the goal of probabilistic job symbiosis modeling is to predict single-threaded progress for each job in a co-schedule. This is done by predicting the probability that a job would experience a base cycle, a miss event cycle and a waiting cycle when co-scheduled with another job. Single-threaded progress then is the sum of the base cycle count plus the miss event cycle counts, and is an indication of the goodness of the co-schedule, i.e., the higher single-threaded progress for each job in the co-schedule, the better the symbiosis.

Probabilistic job symbiosis modeling proceeds in three steps.

Step 1: Estimate multi-threaded base and miss event cycle counts. The multi-threaded base cycle count is set to be the same as the single-threaded base cycle count, i.e., we consider the same unit of work done under multi-threaded and single-threaded execution. In other words, $\tilde{C}^{smt}_{B,i} = \tilde{C}^{st}_{B,i}$. The multi-threaded miss event cycle counts are estimated by multiplying the single-threaded miss event cycle counts with their respective ratios. For example, the branch misprediction miss event cycle count is estimated as $\tilde{C}^{smt}_{br,i} = \tilde{C}^{st}_{br,i} \cdot R_{br,i}$. The L3 D-cache cycle count also needs to account for the reduction in per-thread MLP, i.e., $\tilde{C}^{smt}_{L3D,i} = \tilde{C}^{st}_{L3D,i} \cdot R_{L3D,i} \cdot R_{MLP,i}$.

Step 2: Probability calculation under perfect multithreading. We transform these cycle counts into probabilities by normalizing the individual multi-threaded cycle counts to their overall sum

$$C^{smt}_{perfect,i} = \tilde{C}^{smt}_{B,i} + \sum_e \tilde{C}^{smt}_{e,i},$$

which quantifies the total execution time under perfect multithreading (in the absence of any waiting cycles), i.e., this is the sum of the base cycle count and all the miss event cycle counts. We define the probability for a base cycle for thread i under perfect multi-threaded execution as

$$P^{smt}_{B,i} = \tilde{C}^{smt}_{B,i}/C^{smt}_{perfect,i}.$$

Likewise, we define the probability for a miss event cycle for thread i under perfect multi-threaded execution as

$$P^{smt}_{e,i} = \tilde{C}^{smt}_{e,i}/C^{smt}_{perfect,i}.$$

We also rescale the single-threaded cycle counts using the same denominator, i.e.,

$$P^{st}_{B,i} = \tilde{C}^{st}_{B,i} / C^{smt}_{perfect,i},$$

and

$$P^{st}_{e,i} = \tilde{C}^{st}_{e,i} / C^{smt}_{perfect,i}.$$

Step 3: Waiting cycle probability estimation. Having computed the probabilities for a base cycle $P^{smt}_{B,i}$ and a miss event $P^{smt}_{e,i}$ under perfect multithreading for all m jobs, we can now estimate the probability for a waiting cycle $P^{smt}_{W,i}$. There are three reasons for a waiting cycle, which results in three terms in the calculation of the waiting cycle probability.

(1) Waiting cycle due to dispatching useful instructions from another thread. Thread i experiences a waiting cycle if the processor could dispatch an instruction for thread i but instead dispatches an instruction for another thread. The probability for thread i to dispatch an instruction equals $P^{smt}_{B,i}$; the probability for another thread j to dispatch an instruction equals $P^{smt}_{B,j}$ with $j \neq i$. The product of both probabilities $P^{smt}_{B,i} \cdot P^{smt}_{B,j}$ then quantifies the probability that the processor could dispatch an instruction for thread i but the processor dispatches an instruction for thread j instead. This can be generalized to more than two threads: for each thread $j \neq i$, we thus have the same product from above, and these products can be added. In summary, the first term in the waiting cycle probability equals $P^{smt}_{B,i} \cdot \sum_{j \neq i} P^{smt}_{B,j}$.

(2) Waiting cycle due to dispatching wrong-path instructions from another thread. Thread i experiences a waiting cycle if the processor could dispatch an instruction for thread i but instead dispatches a wrong-path instruction for another thread. This second term is similar to the first term because the thread experiencing a branch misprediction continues fetching (wrong-path) instructions until the branch misprediction is resolved. The likelihood for this case is computed as $P^{smt}_{B,i} \cdot \sum_{j \neq i} P^{smt}_{M,br,j}$.

(3) Waiting cycle due to a back-end resource stall caused by another thread. Thread i experiences a waiting cycle if the processor could dispatch an instruction for thread i but is prevented from doing so because of a back-end resource stall caused by another thread. The back-end resource stall causes dispatch to stall because of a full reorder buffer, full issue queue, no more rename registers, etc. A back-end resource stall primarily occurs upon a long-latency load (cache or TLB) miss or a long chain of dependent long-latency instructions, which causes the reorder buffer to fill up.

The likelihood for this case can be computed as the product of two probabilities: the probability that thread i does not stall on a back-end miss, times the probability that another thread causes a back-end resource stall. The former probability (i.e., thread i does not stall) is computed as $1 - P^{smt}_{backend\ stall,i} = 1 - (P^{smt}_{L3D,i} + P^{smt}_{L2D,i} + P^{smt}_{L1D,i} + P^{smt}_{DTLB,i} + P^{smt}_{other,i})$. The latter probability (i.e., another thread causes a back-end resource stall during multi-threaded execution) is more complicated to compute because it is a function of the other threads' characteristics as well as the SMT fetch policy. We conjecture that this probability can be computed as $\gamma \cdot \bigvee_{j \neq i} P^{smt}_{backend\ stall,j}$, or γ times the probability that at least one other thread causes a back-end resource stall. The big 'or' ($\bigvee$) operator is defined following the sum rule or the addition law of probability. For example, for two threads, $\bigvee_{j=1}^{2} P^{smt}_{backend\ stall,j} = P^{smt}_{backend\ stall,1} + P^{smt}_{backend\ stall,2} - P^{smt}_{backend\ stall,1} \cdot P^{smt}_{backend\ stall,2}$. The γ metric is an empirically derived constant that is specific to the SMT fetch policy and resource partitioning strategy. Intuitively speaking, γ characterizes the likelihood for a long-latency load to result into a resource stall for a given SMT processor configuration. For example, a fetch policy such as round-robin along with a shared reorder buffer and issue queue, will most likely lead to a back-end resource stall because of a full reorder buffer upon a long-latency load miss. In other words, if a thread experiences a long-latency load miss, this will most likely lead to a full reorder buffer and thus a resource stall. Hence, γ will be close to one for the round-robin policy. The flush policy proposed by Tullsen and Brown [30] on the other hand, flushes instructions past a long-latency load miss in order to prevent the long-latency thread from clogging resources. As a result, the likelihood for a back-end resource stall due to a long-latency load miss under the flush policy is small, hence γ will be close to zero for the flush policy. In summary, the likelihood for a waiting cycle for thread i because of a back-end resource stall due to another thread, is computed as $(1 - P^{smt}_{backend\ stall,i}) \cdot \gamma \cdot \bigvee_{j \neq i} P^{smt}_{backend\ stall,j}$.

The overall probability that job i experiences a waiting cycle in a job co-schedule equals the sum of the above probabilities, hence:

$$P^{smt}_{W,i} = P^{smt}_{B,i} \left(\sum_{j \neq i} P^{smt}_{B,j} + \sum_{j \neq i} P^{smt}_{br,j} \right) + \left(1 - P^{smt}_{backend\ stall,i}\right) \cdot \gamma \cdot \bigvee_{j \neq i} P^{smt}_{backend\ stall,j}.$$

The ratio of the sum of the single-threaded probabilities ($P^{st}_{B,i} + \sum P^{st}_{e,i}$) and the estimated multi-threaded probabilities ($P^{smt}_{B,i} + \sum P^{smt}_{e,i} + P^{smt}_{W,i}$) is a measure for the relative progress for job i in a co-schedule. By consequence, for a timeslice of T cycles, single-threaded progress for each job i is estimated as

$$\tilde{C}^{st}_{i} = T \cdot \frac{P^{st}_{B,i} + \sum P^{st}_{e,i}}{P^{smt}_{B,i} + \sum P^{smt}_{e,i} + P^{smt}_{W,i}}.$$

The end result of probabilistic job symbiosis modeling is that it estimates single-threaded progress for each job in a job co-schedule during multi-threaded execution without requiring its evaluation.

6. Model-driven symbiotic job scheduling

The key problem to solve in symbiotic job scheduling on multi-threaded processors is to determine which jobs to co-schedule. Model-driven job scheduling leverages probabilistic job symbiosis modeling to estimate the performance of all (or a large number of) possible job co-schedules for each timeslice. The scheduler then picks the co-schedule that yields the best performance. Predicting the symbiosis of a co-schedule not only eliminates the sampling phase required in prior symbiotic job scheduling proposals, it also enables continuously optimizing the job schedules for optimum performance on a per-timeslice basis; SOS on the other hand, involves multiple timeslices before a new schedule can be established, as described in Section 3.

Model-driven symbiotic job scheduling uses two sources of information. First, it uses multi-threaded and single-threaded performance measurements for each job since the job's arrival in the job mix. The multi-threaded execution time for a job simply is the accumulated number of timeslices since the job's arrival. The single-threaded execution time is the job's accumulated single-threaded progress. System software needs to keep track of the single-threaded and multi-threaded accumulated execution times for each job. The single-threaded execution time for a job in a timeslice is provided by the per-thread cycle accounting architecture, as explained in Section 4; this does not involve any time overhead. Second, probabilistic job symbiosis modeling estimates single-threaded progress for each job in each possible job co-schedule in the next timeslice, as explained in the previous section. (A job's single-threaded cycle stack that serves as input to the probabilistic model is the one computed during the last timeslice that the job was scheduled.) By combining the accumulated performance measures since the job's arrival time with predictions

for the next timeslice, we can estimate the single-threaded progress and multi-threaded execution time, and in turn STP and ANTT, for each job under each possible job co-schedule. The end result is that model-driven scheduling can optimize the job co-schedule for either system throughput, or job turnaround time, or a combination of both; in fact, it is flexible in its optimization target. Prior work on the other hand, uses heuristics to gauge symbiosis and is rigid in its optimization target.

The overhead involved by model-driven symbiotic job scheduling is very limited. Model-driven symbiotic job scheduling requires computing the model formulas for every possible co-schedule each timeslice. For n jobs and m hardware threads, this means that the formulas need to be computed m-combinations out of n. From our experiments we found that computing the formulas takes 22 cycles on average for a two-thread SMT processor and 90 cycles on average for a four-thread SMT processor (using the experimental setup which is described later). Given the simplicity of the formulas, this is done at very limited overhead: for example, around 2000 co-schedules can be evaluated for a two-thread SMT processor for a runtime overhead of around 1%. For a four-thread SMT processor, 500 co-schedules can be evaluated at a 1% runtime overhead. In comparison, SOS [25] considers only 10 possible schedules.

7. Symbiotic proportional-share job scheduling

Modern system software allows users to specify the relative importance of jobs by giving priorities in priority-based scheduling, or by giving shares in proportional-share scheduling. The intuitive understanding in proportional-share scheduling is that a job should make progress proportional to its share. This means that a job's normalized progress under multi-threaded execution C_i^{st}/C_i^{smt} should be proportional to its relative share $p_i/\sum_j p_j$ with p_i the share for job i (the higher p_i, the higher the job's share). For example, a job that has a share that is twice as high compared to another job, should make twice as much progress. System software typically uses time multiplexing to enforce proportional-share scheduling on single-threaded processors by assigning more time slices to jobs with a higher share. Preserving proportional shares on multithreaded processors on the other hand is much harder because of symbiosis between co-executing jobs.

Probabilistic job symbiosis modeling provides a unique opportunity compared to prior work because it tracks and predicts single-threaded progress, which enables preserving proportional shares while exploiting job symbiosis. We pick the job co-schedule that optimizes the SMT performance target of interest if it preserves the proportional shares within a certain range. We therefore define proportional progress PP_i for job i as

$$PP_i = \frac{C_i^{st}/C_i^{smt}}{p_i/\sum_j p_j}.$$

Proportional progress quantifies how proportional a job's normalized progress is compared to its relative share. To quantify proportional progress across jobs, we use the following fairness metric [11]:

$$fairness = \min_{i,j} \frac{PP_i}{PP_j}.$$

Fairness is the minimum ratio of proportional progress for any two jobs in the system, and equals zero if at least one program starves and equals one if all jobs make progress proportional to their relative shares.

Symbiotic proportional-share job scheduling now works as follows. From all job co-schedules that are predicted to achieve a fairness close to one (above 0.9), we choose the one that optimizes SMT performance (recall, this could be either STP, or ANTT, or a combination of both). If none of the job co-schedules is predicted to achieve a fairness above 0.9, we pick the co-schedule with the highest fairness if this fairness is larger than the accumulated fairness so far. If the highest fairness is below the accumulated fairness, we run the job that has made the smallest proportional progress so far in isolation, if this is predicted to improve the overall fairness of the schedule. This will enable the job to catch up with its relative share.

parameter	value
fetch policy	ICOUNT
number of threads	2 and 4 threads
pipeline depth	14 stages
(shared) reorder buffer size	256 entries
instruction queues	96 entries in both IQ and FQ
rename registers	200 integer and 200 floating-point
processor width	4 instructions per cycle
functional units	4 int ALUs, 2 ld/st units and 2 FP units
branch misprediction penalty	11 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
write buffer	24 entries
L1 instruction cache	64KB, 2-way, 64-byte lines
L1 data cache	64KB, 2-way, 64-byte lines
unified L2 cache	512KB, 8-way, 64-byte lines
unified L3 cache	4MB, 16-way, 64-byte lines
instruction/data TLB	128/512 entries, fully-assoc, 8KB pages
cache hierarchy latencies	L2 (11), L3 (35), MEM (350)

Table 1. The baseline SMT processor configuration.

8. Experimental setup

We use the SPEC CPU2000 benchmarks in this paper with their reference inputs. These benchmarks are compiled for the Alpha ISA using the Compaq C compiler (`cc`) version V6.3-025 with the `-O4` optimization flag. For all of these benchmarks we select 500M instruction simulation points using the SimPoint tool [23]. We compose job mixes using these simulation points. In our evaluation, we will be considering two experimental designs. The first design considers a fixed job mix. The second design considers a dynamic job mix in which jobs arrive and depart upon completion.

This study does not include multi-threaded workloads and focuses on multi-program workloads only. The reason is that symbiotic job scheduling is less effective and of less interest for multi-threaded workloads. Threads in a multi-threaded workload that communicate frequently are preferably co-scheduled on a multi-threaded processor in order to reduce synchronization and communication overhead. In other words, the real benefit of symbiotic job scheduling is in co-scheduling unrelated jobs. In addition, sequential programs will continue to be an important class of workloads which motivates further research towards more effective symbiotic job scheduling.

We use the SMTSIM simulator [29] in all of our experiments. We added a write buffer to the simulator's processor model: store operations leave the reorder buffer upon commit and wait in the write buffer for writing to the memory subsystem; commit blocks in case the write buffer is full and we want to commit a store. We simulate a 4-wide superscalar out-of-order SMT processor as shown in Table 1. We assume a shared reorder buffer, issue queue and rename register file unless mentioned otherwise; the functional units are always shared among the co-executing threads. Unless mentioned otherwise, we assume the ICOUNT [31] fetch policy; in the evaluation, we will also consider alternative fetch policies such as flush [30], MLP-aware flush [10] and DCRA [4].

9. Evaluation

We first evaluate the accuracy of probabilistic job symbiosis modeling. Subsequently, we evaluate model-driven scheduling using

a fixed and a dynamic job mix experimental design, and present a detailed analysis which characterizes the contributors to the reported performance improvement. Finally, we evaluate model-driven proportional-share scheduling while exploiting job symbiosis.

9.1 Probabilistic job symbiosis modeling

Recall that the goal for probabilistic job symbiosis modeling is to estimate single-threaded progress for individual jobs in a job co-schedule under multi-threaded execution. As explained in Section 5, probabilistic job symbiosis modeling basically boils down to estimating the waiting cycle count under multi-threaded execution. We now evaluate the accuracy in estimating this waiting cycle count. We therefore consider 36 randomly chosen two-program job mixes and 30 randomly chosen four-program job mixes. We run a multi-threaded execution for each job mix, and compute the single-threaded cycle stacks as described in Section 4; starting from these single-threaded cycle stacks, we then estimate the waiting cycle count for each job in the job mix using the probabilistic job symbiosis model, as described in Section 5, and compare the estimated waiting cycle count against the one measured using the cycle accounting architecture described in Section 4. The difference between the estimated waiting cycle count and the measured waiting cycle count, normalized by the total multi-threaded execution time is our error metric for probabilistic job symbiosis modeling. This evaluation setup considers the full path accuracy of the probabilistic model.

Figure 1 quantifies the error for probabilistic job symbiosis modeling as a histogram; the two graphs in Figure 1 consider two-program workloads and four-program workloads, respectively. These histograms show the number of jobs on the vertical axis for which the error metric is within a given bucket shown on the horizontal axis. The average absolute error equals 5.5% and 8.8% for two-program and four-program workloads, respectively. The largest errors are observed for only a couple outlier job mixes. These outliers are caused by the implicit assumption made by the probabilistic model that the γ parameter is job mix independent.

These results assume an SMT processor with a dynamically partitioned or shared reorder buffer and issue queue along with the ICOUNT fetch policy. We obtain similarly accurate results for other resource partitioning strategies and fetch policies. For example, the average absolute error is around 2.9% for a statically partitioned reorder buffer and issue queue (and shared functional units) and the round-robin fetch policy [20], assuming two-program workloads. Similarly, for the flush policy [30] and a dynamically partitioned reorder buffer and issue queue, we achieve an average absolute error of 5.8%; for MLP-aware flush [10], we achieve an average absolute error of 4.2%; and for DCRA [4], we achieve an average absolute error of 4.5%. Of course, each of these fetch policies comes with a different empirically derived γ parameter; γ varies between 0.05 (for the flush policy) to 0.36 (for ICOUNT).

9.2 Fixed job mix

Having evaluated the accuracy of probabilistic symbiosis modeling, we now evaluate the effectiveness of symbiotic job scheduling that leverages the probabilistic model. We first consider a fixed job mix; we will consider a dynamic job mix later. In each of the following experiments, we assume the following setup. We consider a fixed job mix consisting of m randomly chosen jobs on an n-threaded SMT processor, with $m > n$. Each timeslice is assumed to be 5M cycles, which corresponds to a few milliseconds given contemporary processor clock frequencies in the GHz range; this is a realistic assumption given today's operating systems, e.g., the Linux 2.6 kernel allows for a timeslice as small as 1ms with a default timeslice of 4ms. (The short timeslice also somewhat compensates for the lack of blocking behavior in the compute-intensive SPEC CPU benchmarks in our setup.) Symbiotic job scheduling schedules jobs in each timeslice following the algorithm described in Section 6. We compare against (i) a naive scheduling approach that co-schedules jobs in a round-robin manner, and (ii) Sample, Optimize, Symbios (SOS) proposed by Snavely and Tullsen [25] (SOS uses a set of heuristics to assess symbiosis, and we report performance results for the IPC heuristic.)

In our first experiment, we optimize for system throughput (STP); in our second experiment, we optimize for job turnaround time (ANTT). The results are shown in Figures 2 and 3 for STP and ANTT, respectively, for a two-thread SMT and four-thread SMT with twice as many jobs in the job mix as there are hardware threads, i.e., $m = 2n$. Model-driven scheduling improves STP by on average 7% and 13% over SOS and naive scheduling, respectively; system throughput improves by up to 34% for some job mixes. We obtain similarly good results when optimizing for job turnaround time. Model-driven job scheduling reduces ANTT by on average 5.3% and 9.1% over SOS and naive scheduling, respectively, and up to 20% for some job mixes. The job mixes for which we observe a decrease in STP or increase in ANTT compared to naive scheduling and/or SOS, are due to inaccurate single-threaded cycle stack estimates by the cycle accounting architecture. Improving the accuracy of the cycle accounting architecture is likely to improve the efficacy of model-driven scheduling further. On average, model-driven scheduling leads to (significant) SMT performance improvements.

It is worth noting that the efficacy of model-driven job scheduling increases with an increasing number of jobs. Figures 4 and 5 show STP and ANTT results, respectively, for a 6-job mix on a two-threaded SMT processor and a 12-job mix on a four-threaded SMT processor. In comparison to Figures 2 and 3, we observe a higher STP improvement over naive scheduling for 6-job mixes (17%) than for 4-job mixes (11%) for the two-threaded SMT processor; on the four-threaded SMT processor, we achieve an average 25% STP increase for the 12-job mixes compared to 13% for the 6-job mixes. Similarly, we observe a higher ANTT reduction for 6-job mixes (13%) than for 4-job mixes (9%) for the two-threaded SMT processor; for the four-threaded SMT processor, the average ANTT goes down by 16% for the 12-job mixes compared to 12% for the 8-job mixes. In other words, the more jobs in the job mix, the better the probabilistic symbiosis model exploits the potential performance improvement through symbiotic job scheduling.

9.3 Detailed performance breakdown

Having reported these substantial improvements over prior work in symbiotic job scheduling, the interesting question is where these overall improvements come from, and what the relative importance is for each of these contributors. We identify three potential contributors: (i) model-driven job scheduling does not rely on heuristics; (ii) it does not execute job co-schedules to evaluate symbiosis but instead predicts symbiosis, which eliminates the sampling overhead of running suboptimal co-schedules to evaluate symbiosis and which enables continuous optimization upon each timeslice in contrast to optimization on schedule boundaries that span multiple timeslices; and (iii) it does not rely on sampling of a limited number of co-schedules but can evaluate a large number of possible co-schedules. To understand the relative importance of these three contributors, we set up the following experiment in which we evaluate a range of job scheduling algorithms starting with SOS [25] and gradually add features to arrive at model-driven scheduling; the deltas between the intermediate scheduling algorithms illustrate the importance of each of the above contributors. We consider the following job scheduling algorithms:

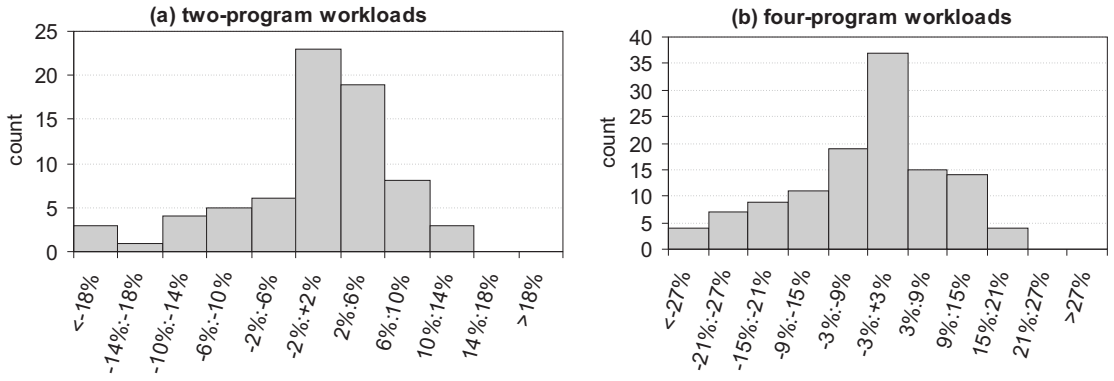

Figure 1. Validating probabilistic job symbiosis modeling: error histogram for (a) two-program workloads and (b) four-program workloads.

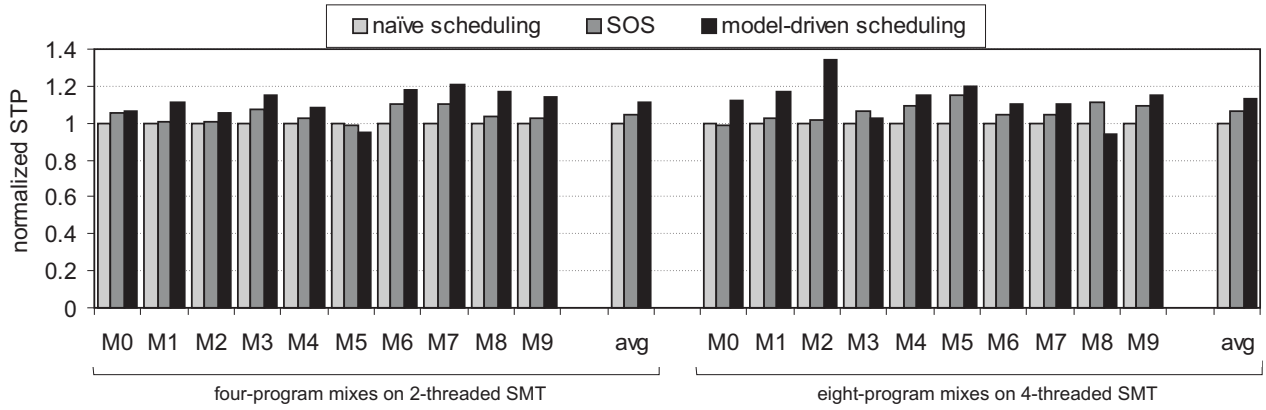

Figure 2. Optimizing for system throughput (STP) for four-program mixes on a two-threaded SMT (left) and eight-program mixes on a four-threaded SMT (right).

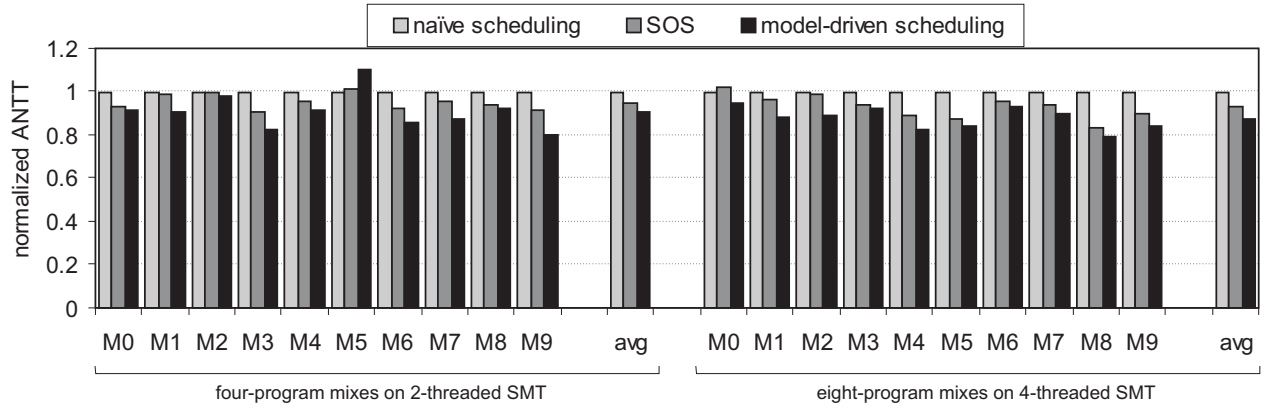

Figure 3. Optimizing for job turnaround time (ANTT) for four-program mixes on a two-threaded SMT (left) and eight-program mixes on a four-threaded SMT (right).

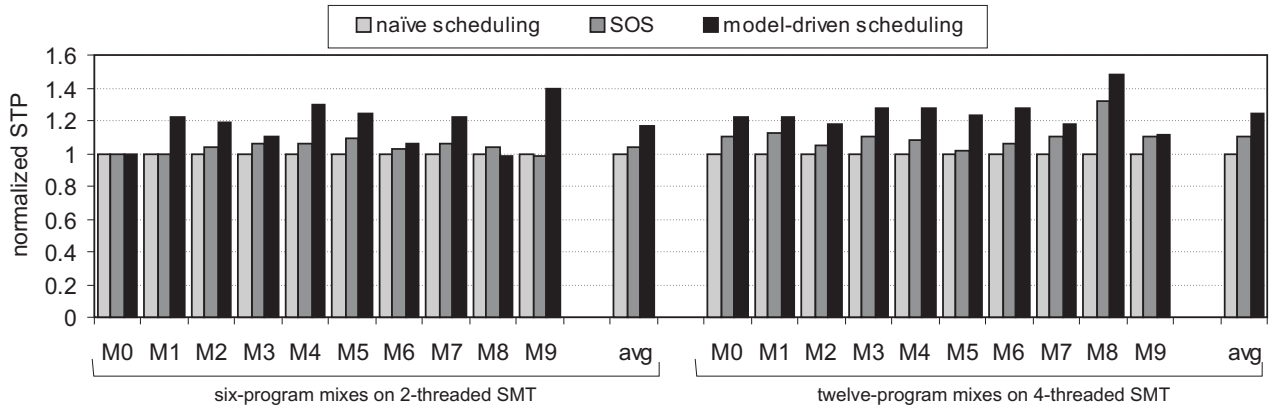

Figure 4. Optimizing for system throughput (STP) for six-program mixes on a two-threaded SMT (left) and twelve-program mixes on a four-threaded SMT (right).

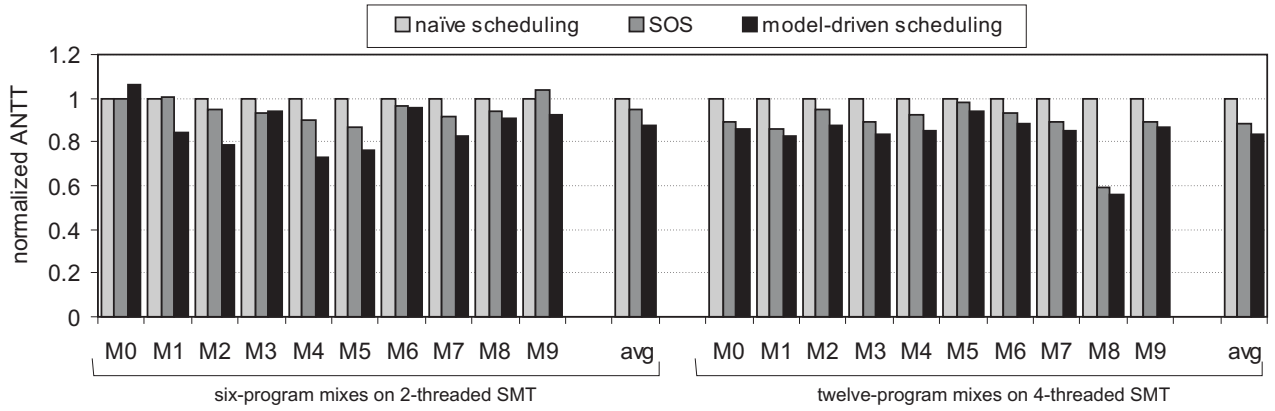

Figure 5. Optimizing for job turnaround time (ANTT) for six-program mixes on a two-threaded SMT (left) and twelve-program mixes on a four-threaded SMT (right).

(a) *SOS with heuristic* is the SOS approach as proposed in [25] using IPC as the heuristic.

(b) *SOS with cycle accounting architecture* is a variant of the SOS approach that uses the cycle accounting architecture to estimate single-threaded progress during multi-threaded execution. These single-threaded progress rates are then used to evaluate whether the schedule optimizes system throughput. The delta between (b) and (a) quantifies the importance of not having to rely on heuristics for gauging job symbiosis.

(c) *Sampling-based job scheduling* uses probabilistic job symbiosis modeling to estimate job symbiosis for a limited number of possible co-schedule. It considers 10 possibles co-schedules and uses the probabilistic symbiosis model to gauge symbiosis but does not evaluate symbiosis by executing the co-schedule. The delta between (c) and (b) quantifies the importance of eliminating the sampling overhead and not having to evaluate symbiosis through execution.

(d) *Model-driven job scheduling* is the approach as proposed in this paper and evaluates all possible co-schedules through probabilistic symbiosis modeling. The delta between (d) and (c) quantifies the impact of not being limited by the small number of job co-schedules to choose from.

Figure 6 reports the achieved STP for each of the above job scheduling algorithms for an 8-job mix and a 12-job mix on a four-threaded SMT processor — this results in 70 and 495 possible co-schedules to choose from, respectively. We observe that the overall performance improvement compared to SOS comes from multiple sources. For some job mixes, the performance improvement comes from estimating symbiosis for a large number of possible co-schedules (see 8-job mixes #0 and #1). For other mixes, such as 8-job mix #2, the performance improvement comes from eliminating the overhead of running sub-optimal co-schedules during the SOS sampling phase and from continuous schedule optimization on per-timeslice basis. For yet other mixes, such as 8-job mix #5, the biggest improvement comes from not having to rely on heuristics. Interestingly, for the 8-job mixes, the various sources of improvement contribute equally, however, for the 12-job mixes being able to evaluate a large number of possible co-schedules has the largest contribution.

9.4 System-level priorities and shares

Symbiotic job scheduling should be able to preserve system-level shares for it to be useful in modern system software. Recall from Section 7 that the intuitive meaning of proportional-share scheduling is that a job should make progress proportional to its relative share. For evaluating whether relative shares are met by model-driven symbiotic job scheduling, we set up an experiment in which

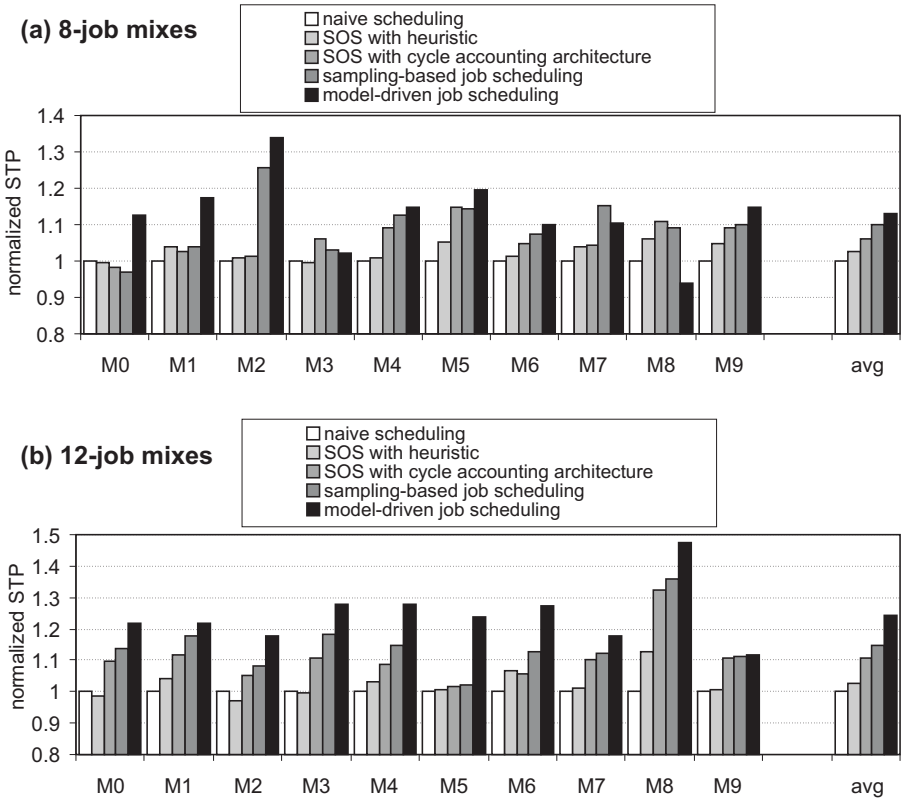

Figure 6. Understanding where the performance improvement comes from: STP for a number of symbiotic job scheduling algorithms, assuming (a) an 8-job mix and (b) a 12-job mix on a four-threaded SMT processor.

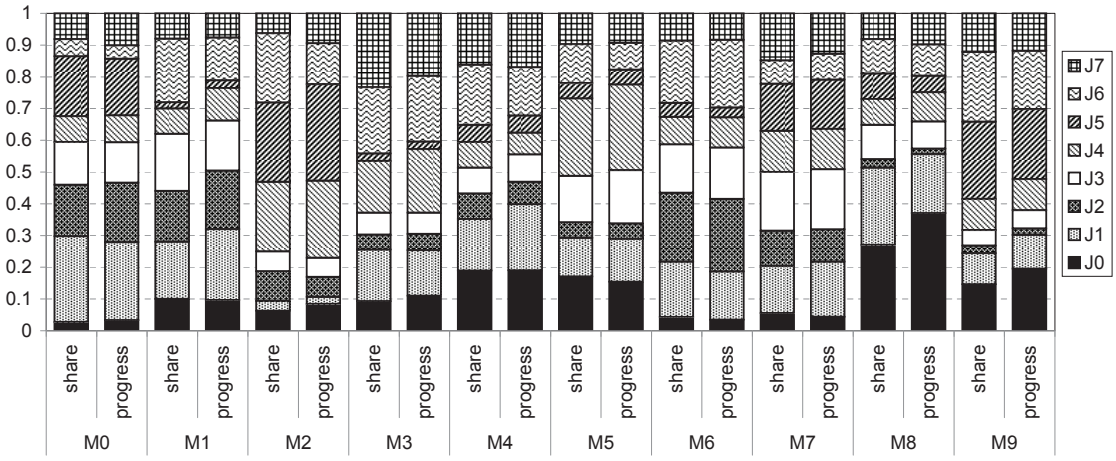

Figure 7. Comparing normalized progress against relative shares for model-driven proportional-share job scheduling.

we randomly assign shares (between 1 and 10) to the 8 jobs in the job mix; we assume a four-thread SMT processor. Figure 7 compares the normalized progress for each of the jobs in each job mix against its relative share; both are shown as a normalized stack. A good match between both stacks demonstrates that relative shares are preserved by the job scheduling algorithm, which we find to be the case for model-driven job scheduling.

9.5 Dynamic job mix

So far, we assumed a fixed job mix. However, in a practical situation, jobs come and go as they enter the system and complete. To mimick this more realistic situation, we now consider a dynamic job mix. We assume an average job length of 200 million cycles, and the average job inter-arrival time is determined such that the average number of available jobs at any time is more than two times the number of hardware threads in the SMT processor — this is done using M/M/1 queueing theory. Further, to quickly warm up the system, we assume 4 and 8 initial threads for the two-thread and four-thread SMT processor, respectively. For the two-thread and four-thread SMT processor, this yields on average 3.6 and 7.2 ready jobs at each point in time, respectively. We record the arrival times for the jobs in the dynamic job mix once and subsequently replay the job mix when comparing different job scheduling algorithms.

In a situation where jobs come and go, it makes sense to focus on turnaround time rather than system throughput. Although system throughput can be measured over a period of time where the job mix is constant, system throughput measured over the entire dynamic job mix makes little sense because system throughput cannot possibly exceed the job arrival rate. We therefore optimize for job turnaround time in this experiment. Figure 8 reports the improvement in job turnaround time for model-driven job scheduling compared to naive scheduling and SOS for both two-thread and four-thread SMT processors. Model-driven scheduling improves by 21% on average compared to naive scheduling, and by 16% on average compared to SOS. For some job mixes we observe a reduction in job turnaround time by 44% (mix 5 for the 2-threaded SMT processor) and by 36% (mix 1 for the 4-threaded SMT processor) compared to naive scheduling, and by 35% (mix 1 for the 2-threaded SMT processor) and 45% (mix 0 for the 4-threaded SMT processor) compared to SOS.

10. Related work

Symbiotic job scheduling. Snavely and Carter [24] were the first to coin the term 'symbiotic job scheduling' and developed the SOS symbiotic job scheduling algorithm for the Tera MTA (Multi-Threaded Architecture). Snavely and Tullsen [25] extended the SOS approach to SMT processors, and Snavely et al. [26] studied the interplay between symbiotic job scheduling and system-level priorities. We extensively argued the improvements of probabilistic job symbiosis modeling and model-driven job scheduling over SOS throughout the paper.

Several other proposals have been made to symbiotic job scheduling. Settle et al. [22] drive symbiotic job scheduling by monitoring activity in the memory subsystem. El-Moursy et al. [9] monitor contention in the register file, functional units and L1 caches. Parekh et al. [18] monitor cache miss rates and IPC. Bulpin and Pratt [2] build an empirical model that predicts single-threaded progress based on hardware performance counter data. The key difference between these prior approaches and this paper is that these prior approaches use heuristics, focus on a single source of resource contention, and/or require a sampling phase to gauge symbiosis. Probabilistic symbiosis modeling on the other hand does not rely on heuristics and enables predicting a priori which co-schedules will result in positive symbiosis.

Other papers study job co-scheduling in a different setting. Tam et al. [27] co-schedule threads from a multi-threaded workload on the same chip in a multiprocessor environment based on shared memory access patterns. Jain et al. [16] study symbiotic scheduling of soft real-time applications on SMT processors. Fedorova et al. [13] find that non-work-conserving scheduling, i.e., running fewer threads than there are hardware threads, can improve system performance; they use an analytical model to find cases where a non-work-conserving policy is beneficial. Probabilistic job symbiosis modeling could be helpful in predicting the impact of a non-work-conserving schedule on overall SMT performance; this would be a fairly straightforward extension to model-driven job scheduling.

VMware's ESX Server 2.1 hypervisor offers SMT support [33]. It assumes a simple accounting mechanism: it assumes that jobs co-executing on a 2-thread SMT processor make half as much progress as when run in isolation. VMware's ESX Server leverages this accounting mechanism to give CPU time to virtual machines proportional to their share allocation, capped by minimum and maximum values. To achieve proportional progress, ESX Server dynamically decides whether or not to run virtual machines in isolation or co-scheduled with other virtual machines. To the best of our knowledge, ESX Server does not exploit job (i.e., virtual machine) symbiosis. In addition, our cycle accounting scheme (as described in Section 4) makes a more accurate estimate of single-threaded progress during SMT execution.

Multithreaded processors. This paper focused on probabilistic modeling for symbiotic job scheduling in the context of a simultaneous multithreading (SMT) processor. Our choice for SMT processors is motivated by its wider commercial adoption and the larger performance entanglement between co-scheduled jobs compared to other multithreading paradigms such as fine-grained multithreading (e.g., Tera MTA, HEP) and coarse-grained multithreading (e.g., IBM RS64 IV, Intel Montecito). By consequence, the modeling challenge for job symbiosis is the largest for SMT processors. We strongly believe that the general idea of probabilistic job symbiosis modeling is (easily) extendable to other flavors of multithreading.

Improving shared resource utilization. A large body of work has been done on improving shared resource utilization for both SMT and multi-core processors. Tullsen et al. [31] realized the importance of resource partitioning and fetch policies on SMT performance, and proposed the ICOUNT mechanism as an effective solution. Follow-on research has proposed further refinements and improvements, such as flush [30], MLP-aware flush [10], DCRA [4], hill-climbing [7], runahead threads [21], etc.

Chandra et al. [6] propose an analytical model that predicts the number of additional misses for each thread due to cache sharing. The input to the model is the per-thread L2 stack distance distribution. This analytical model can be used for example by system software to improve cache symbiosis in a chip multiprocessor with shared caches. Qureshi and Patt [19] aim at creating better cache symbiosis through a hardware mechanism that provides more cache resources to threads that benefit more performance-wise from the increased cache resources.

QoS management in multi-threaded processors. A number of studies have been done on improving quality-of-service (QoS) in multi-threaded processors. Cazorla et al. [3, 5] target QoS in SMT processors through resource allocation. They propose a system that samples single-threaded IPC, and dynamically adjusts the resources to achieve a pre-set percentage of single-threaded IPC. Cota-Robles [8] describes an SMT processor architecture that combines OS priorities with thread efficiency heuristics (outstanding instruction counts, number of outstanding branches, number of data

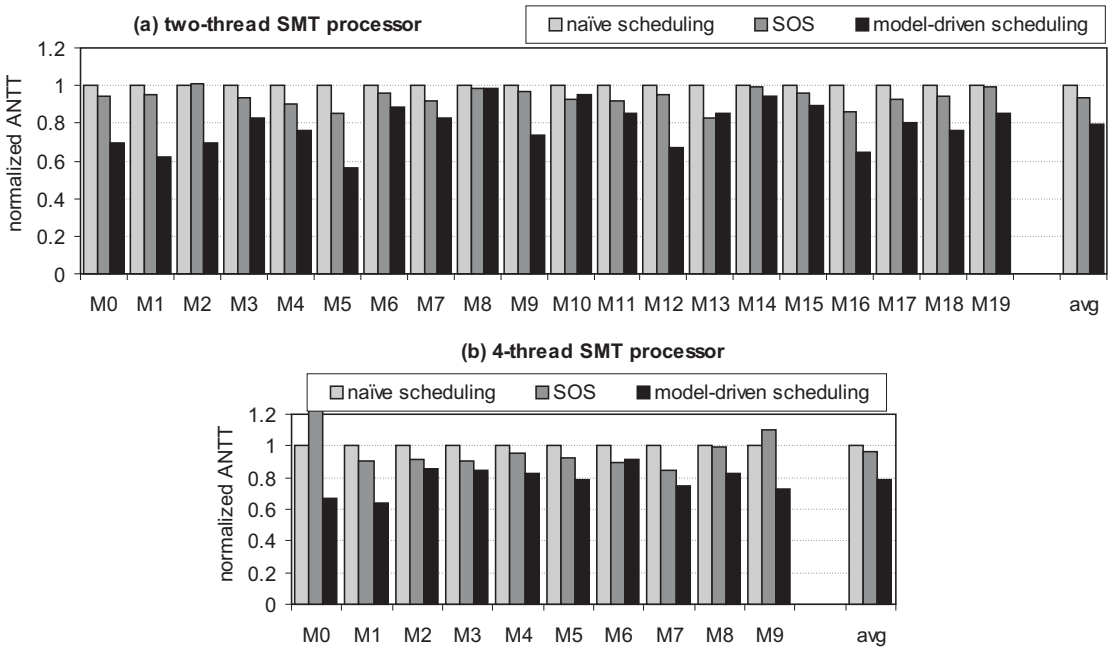

Figure 8. Evaluating model-driven job scheduling for a dynamic job mix for (a) a two-thread SMT processor and (b) a four-thread SMT processor.

cache misses) to provide a dynamic priority for each thread scheduled on the SMT processor. The IBM POWER5 [1, 15] implements a software-controlled priority scheme that controls the per-thread dispatch rate. Software-controlled priorities are independent of the operating system's concept of thread priority and are used for temporarily increasing the priority of a process holding a critical spin-lock, or for temporarily decreasing the priority of a process spinning for a lock, etc. Gabor et al. [14] propose fairness enforcement on coarse-grained switch-on-event (SOE) multi-threaded processors.

11. Conclusion

Job co-scheduling by system software has a significant impact on overall SMT processor performance. Symbiotic job scheduling, which seeks to exploit the positive symbiosis between co-executing jobs, can lead to substantially higher system throughput and lower job turnaround time. This paper addressed the fundamental problem in symbiotic job scheduling and proposed probabilistic job symbiosis modeling for estimating the symbiosis between jobs in a co-schedule without having to execute the co-schedule. The model itself is simple enough to be implemented in system software. Probabilistic job symbiosis enhances previously proposed symbiotic job scheduling algorithms by: (i) eliminating the sampling phase which requires co-schedule execution to evaluate symbiosis, (ii) continuously readjusting the job co-schedule, (iii) evaluating a large number of possible co-schedules at very low overhead, (iv) tracking and predicting single-threaded progress during multi-threaded execution instead of having to rely on heuristics, (v) optimizing SMT performance targets of interest (e.g., STP, or ANTT), (vi) preserving system software level priorities/shares. These innovations over prior work make symbiotic job scheduling both practical and more effective. Our experimental results report substantial improvements over prior work. In a realistic experiment where jobs come and go, we report an average 16% (and up to 35%) and 19% (and up to 45%) reduction in job turnaround time for a two-thread and four-thread SMT processor, respectively, compared to the previously proposed SOS algorithm.

As part of our future work, we plan to extend probabilistic symbiosis modeling and model-driven job scheduling to other forms of multi-threading (fine-grained and coarse-grained multi-threading), as well as multi-core and many-core processors. In addition, we plan to study symbiotic job scheduling for multi/many-core processors in which each core is a multi-threaded processor: the key question then is to decide which jobs to co-schedule on a core and which jobs to schedule on different cores for optimum performance. Also, we plan to study job symbiosis job scheduling issues when co-scheduling multi-program and parallel workloads.

Acknowledgements

We thank the reviewers for their constructive and insightful feedback. Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109.

References

[1] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of POWER5 processor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 415–426, June 2008.

[2] J. R. Bulpin and I. Pratt. Hyper-threading aware process scheduling heuristics. In *Proceedings of the USENIX Annual Technical Conference*, pages 103–106, Apr. 2005.

[3] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006.

[4] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the*

37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 171–182, Dec. 2004.

[5] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, July 2004.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In *Proceedings of the Eleventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, Feb. 2005.

[7] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 239–250, June 2006.

[8] E. Cota-Robles. *Priority Based Simultaneous Multi-Threading*, Dec. 2003. United States Patent No. 6,658,447 B2.

[9] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006.

[10] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 240–249, Feb. 2007.

[11] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[12] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–144, Mar. 2009.

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

[14] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):34, Sept. 2007.

[15] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, C. Sosa, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM, Nov. 2005.

[16] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*, pages 134–145, Dec. 2002.

[17] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, Nov. 2001.

[18] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, 2000.

[19] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, Dec. 2006.

[20] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–26, Sept. 2003.

[21] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve SMT performance. In *Proceedings of the Fourteenth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 149–158, Feb. 2008.

[22] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, Sept. 2004.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[24] A. Snavely and L. Carter. Symbiotic jobscheduling on the MTA. In *Proceedings of the Workshop on Multi-Threaded Execution, Architecture and Compilers*, Jan. 2000.

[25] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Nov. 2000.

[26] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, June 2002.

[27] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the European Conference in Computer Systems (EuroSys)*, pages 47–58, Mar. 2007.

[28] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 26–34, Sept. 2003.

[29] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[30] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 318–327, Dec. 2001.

[31] 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 *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 191–202, May 1996.

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

[33] VMware. *HyperThreading Support in VMware ESX Server 2.1*, Apr. 2004.