Probabilistic Modeling for Job Symbiosis Scheduling on SMT Processors

STIJN EYERMAN and LIEVEN EECKHOUT, Ghent University, Belgium

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

This article proposes probabilistic job symbiosis modeling, which predicts whether jobs will create positive or negative symbiosis when coscheduled without requiring the coschedule 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 coschedule continuously, (iii) it evaluates a large number of possible coschedules 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 [**Processor Architectures**]: Parallel Architectures; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Simultaneous multithreading (SMT), symbiotic job scheduling, performance modeling

ACM Reference Format:
Eyerman, S. and Eeckhout, L. 2012. Probabilistic modeling for job symbiosis scheduling on SMT processors. ACM Trans. Architec. Code Optim. 9, 2, Article 7 (June 2012), 27 pages.
DOI = 10.1145/2207222.2207223 http://doi.acm.org/10.1145/2207222.2207223

This article extends "Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling," by Stijn Eyerman and Lieven Eeckhout, published at the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10), 91–102.
S. Eyerman is supported through a postdoctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0255.08, and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Communitys Seventh Framework Program (FP7/2007-2013)/ERC Grant agreement no. 259295.
Authors' addresses: S. Eyerman and L. Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: {seyerman, leeckhou}@elis.UGent.be.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1544-3566/2012/06-ART7 $10.00
DOI 10.1145/2207222.2207223 http://doi.acm.org/10.1145/2207222.2207223

1. INTRODUCTION

Simultaneous multithreading (SMT) processors [Tullsen et al. 1996, 1995], such as the Intel Core i7, IBM POWER7, and Sun Niagara, seek at improving microprocessor utilization by sharing hardware resources across multiple active threads. Shared hardware resources however may affect system performance in unpredictable ways. Coexecuting jobs may conflict with each other on various shared resources which may adversely affect overall performance. Or, reversely, system performance may greatly benefit from coexecuting jobs that put "compatible" demands on shared resources. In other words, the performance interactions and symbiosis between coexecuting jobs on multithreaded processors can be positive or negative [Tuck and Tullsen 2003].

Because of the symbiosis among coexecuting jobs, it is important that system software (the operating system or the virtual machine monitor) makes appropriate decisions about what jobs to coschedule 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 coscheduling 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 coscheduled. Previously proposed symbiotic job scheduling approaches [Snavely and Carter 2000; Snavely and Tullsen 2000; Snavely et al. 2002] sample the space of possible coschedules, that is, they select and execute a limited number of job coschedules, and then retain the most effective one according to some heuristic(s). This pragmatic approach however may lead to suboptimal coschedules and thus suboptimal system performance.

This article proposes probabilistic job symbiosis modeling. The probabilistic model uses probabilistic theory to estimate single-threaded progress rates for the individual jobs in a job coschedule without requiring its evaluation. The model uses as input per-thread cycle stacks (which are computed using our previously proposed cycle accounting architecture [Eyerman and Eeckhout 2009]), and is simple enough so that system software can evaluate all or at least a very large number of possible job coschedules at low overhead, and search for optimum job coschedules. 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 coschedule continuously.
(iii) It enables evaluating all (or at least a very large number of) possible job coschedules 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 coschedule, 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.
(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. [2002].

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 [Tullsen et al. 1996]. 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 [Raasch and Reinhardt 2003] (2.9%), flush [Tullsen and Brown 2001] (5.8%), MLP-aware flush [Eyerman and Eeckhout 2007] (4.2%) and DCRA [Cazorla et al. 2004a] (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) [Snavely et al. 2002], 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 article makes the following contributions.

—We propose probabilistic job symbiosis modeling for SMT processors. 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 [Eyerman and Eeckhout 2009]), and predicts progress for each job in a job coschedule. 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 on SMT processors and demonstrate that it significantly improves performance over prior proposals. We provide a comprehensive analysis as to where the performance improvement comes from.
—Using the model, we analyze which job types are best coscheduled to maximize throughput and job turnaround time. We conclude that if the goal is to maximize throughput, memory-intensive jobs should be coscheduled, however, if job turnaround time is to be optimized, then mixed coschedules with memory-intensive and compute-intensive jobs is the best choice.
—We find that global scheduling, that is, scheduling job mixes for a number of timeslices ahead, does not improve performance over local scheduling which schedules the next timeslice only.
—We study the interaction between job symbiosis scheduling on SMT processors and DVFS. We find that SMT job scheduling and DVFS can be optimized independently, that is, a combined scheme that optimizes job scheduling and DVFS independently yields comparable performance to a cooperative scheme that optimizes job scheduling and DVFS simultaneously.

This article 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

[Eyerman and Eeckhout 2008], we identified two primary performance metrics for multiprogram 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 multithreaded (SMT) 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 [2000], 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 multithreaded 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. [2001].

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, that is, 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 multithreaded 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 multithreaded execution.

3. PRIOR WORK IN SYMBIOTIC JOB SCHEDULING ON SMT PROCESSORS

Snavely and Tullsen [2000] pioneered the work on symbiotic job scheduling for simultaneous multithreading (SMT) processors, and proposed SOS, or Sample, Optimize and Symbios. For explaining SOS, we first need to define some terminology. A *job coschedule* refers to multiple independent jobs coexecuting on the multithreaded processor during a timeslice, that is, the jobs in a coschedule 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 coschedules such that each job in the schedule appears in an equal number of job coschedules; a schedule thus spans multiple timeslices. During the sample phase, SOS permutes the schedule periodically, changing the jobs that are coscheduled. While doing so, SOS collects various hardware performance counter values such as IPC, cache performance, issue queue occupancy, etc. to estimate the 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 coschedules. The goodness of symbiosis is a predictor (heuristic) for system throughput.

In their follow-on work, Snavely et al. [2002] 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 coscheduled, (ii) a more complex mechanism that strives at exploiting symbiosis when coscheduling 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; 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 multithreaded 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 [Eyerman and Eeckhout 2009] 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 corun 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 coschedule.

—*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 multithreaded execution.

The cycle accounting architecture also computes two sources of reduced performance under multithreaded execution compared to single-threaded execution. First, it quantifies the reduction in per-thread memory-level parallelism (MLP). Multithreaded 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 multithreaded 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 multithreaded 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 multithreading.

Given these 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 their respective ratios. For example, the branch misprediction cycle count is estimated as the multithreaded cycle count divided by the branch misprediction ratio, that is, $\tilde{C}^{st}_{br,i} = C^{smt}_{br,i}/R_{br,i}$. 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 multithreaded execution, that is, $\tilde{C}^{st}_{B,i} = C^{smt}_{B,i}$. 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 Eyerman and Eeckhout [2009] 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 coschedule. This is done by predicting the probability that a job would experience a base cycle, a miss event cycle and a waiting cycle when coscheduled 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 coschedule, that is, the higher single-threaded progress for each job in the coschedule, the better the symbiosis.

Probabilistic job symbiosis modeling proceeds in three steps.

Step 1. *Estimate multithreaded base and miss event cycle counts.* The multithreaded base cycle count is set to be the same as the single-threaded base cycle count, that is, we consider the same unit of work done under multithreaded and single-threaded execution. In other words, $\tilde{C}^{smt}_{B,i} = \tilde{C}^{st}_{B,i}$. The multithreaded 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, that is, $\tilde{C}^{smt}_{L3D,i} = \tilde{C}^{st}_{L3D,i} \cdot R_{L3D,i} \cdot R_{MLP,i}$.

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), that is, 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 multithreaded 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 multithreaded 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, that is,

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

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 multithreaded 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, $\bigvee^{2}_{j=1} P^{smt}_{backend\,stall,j} = P^{smt}_{backend\,stall,1} + P^{smt}_{backend\,stall,2} - P^{smt}_{backend\,stall,1} \cdot P^{smt}_{backend\,stall,2}$. 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 [2001] 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 coschedule equals the sum of these 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 multithreaded 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 coschedule. 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 coschedule during multithreaded execution without requiring its evaluation.

6. MODEL-DRIVEN SYMBIOTIC JOB SCHEDULING

The key problem to solve in symbiotic job scheduling on multithreaded processors is to determine which jobs to coschedule. Model-driven job scheduling leverages probabilistic job symbiosis modeling to estimate the performance of all (or a large number of) possible job coschedules for each timeslice. The scheduler then picks the coschedule that yields the best performance. Predicting the symbiosis of a coschedule 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 multithreaded and single-threaded performance measurements for each job since the job's arrival in the job mix. The multithreaded 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 multithreaded 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 coschedule 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 multithreaded execution time, and in turn STP and ANTT, for each job under each possible job coschedule. The end result is that model-driven scheduling can optimize the job coschedule 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 coschedule 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 coschedules can be evaluated for a two-thread SMT processor for a runtime overhead of around 1%. For a four-thread SMT processor, 500 coschedules can be evaluated at a 1% runtime overhead. In comparison, SOS [Snavely and Tullsen 2000] 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 multithreaded 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 coexecuting 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 coschedule 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 [Eyerman and Eeckhout 2008]:

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

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 coschedules 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 coschedules is predicted to achieve a fairness above 0.9, we pick the coschedule with the highest fairness if its fairness is larger than the accumulated fairness so far. If the highest fairness is smaller than 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. Note that the 0.9 threshold is chosen arbitrarily, and is a parameter that can be set by the user or operating system. Choosing a threshold close to one will result in job progress rates close to their relative shares. Picking a lower threshold value might result in better performance, however at the cost of job progress rates being off relative to their relative shares. For the 0.9 threshold, our experiments showed that a job coschedule was selected to improve performance (i.e., fairness is higher than 0.9) for approximately two thirds of the time; a job coschedule was selected to improve overall fairness (i.e., fairness is smaller than 0.9) for the remaining one third of the time.

8. EXPERIMENTAL SETUP

We use the SPEC CPU2000 benchmarks in this article 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 [Sherwood et al. 2002]. 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 multithreaded workloads and focuses on multiprogram workloads only. The reason is that symbiotic job scheduling is less effective and of less interest for multithreaded workloads. Threads in a multithreaded workload that communicate frequently are preferably coscheduled on a multithreaded processor in order to reduce synchronization and communication overhead. In other words, the real benefit of symbiotic job scheduling is in coscheduling 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 [Tullsen 1996] 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 another store. We simulate a 4-wide superscalar out-of-order SMT processor, as shown in Table I. We assume a shared reorder buffer, issue queue and rename register file unless mentioned otherwise; the functional units are always shared among the coexecuting threads. Unless mentioned otherwise, we assume the ICOUNT [Tullsen et al. 1996] fetch policy; in the evaluation, we will also consider alternative fetch policies such as flush [Tullsen and Brown 2001], MLP-aware flush [Eyerman and Eeckhout 2007] and DCRA [Cazorla et al. 2004a].

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

Table I. The Baseline SMT Processor Configuration

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)

design, and present a detailed analysis which characterizes the contributors to the reported performance improvement. We then evaluate model-driven proportional-share scheduling while exploiting job symbiosis. Finally, we provide analysis with respect to which job types are best coscheduled; we study whether global scheduling over multiple timeslices yields any performance benefit over local scheduling for the next timeslice; and we study the interaction between symbiotic job scheduling and DVFS for optimizing energy-efficiency.

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 coschedule under multithreaded execution. As explained in Section 5, probabilistic job symbiosis modeling basically boils down to estimating the waiting cycle count under multithreaded 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 multithreaded 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. The difference between the estimated waiting cycle count and the measured waiting cycle count, normalized to the total multithreaded 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 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.

To find out where the largest errors come from, we performed an experiment in which we first assume all caches and predictors to be perfect, that is, all cache accesses are hits and all branches are correctly predicted. In the next simulation, we assume a realistic instruction cache and TLB while assuming the other caches/TLBs and branch

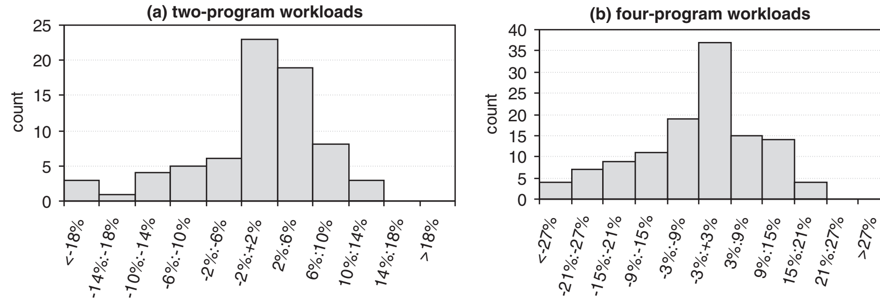

Fig. 1. Validating probabilistic job symbiosis modeling: error histogram for (a) two-program workloads and (b) four-program workloads.

Table II. Modeling Error and γ Parameter for Different Fetch Policies

Partitioning	Fetch policy	Error	γ
static	round robin [Raasch and Reinhardt 2003]	2.9%	0.11
dynamic	ICOUNT [Tullsen et al. 1996]	5.5%	0.36
dynamic	flush [Tullsen and Brown 2001]	5.8%	0.05
dynamic	MLP-aware flush [Eyerman and Eeckhout 2007]	4.2%	0.10
dynamic	DCRA [Cazorla et al. 2004a]	4.5%	0.12

predictor to be perfect. We subsequently add a realistic branch predictor, and finally realistic data caches and TLBs. This yields four simulations per workload, and for each of these simulations, we measure the number of waiting cycles. We also estimate the number of waiting cycles using the probabilistic model, and we then compare the estimated versus the measured number of waiting cycles. This experiment revealed that the model outliers come from data cache and TLB modeling. These outliers are caused by the implicit assumption made by the probabilistic model that the γ parameter is job mix independent, that is, we assume that the probability for a job to fill up the entire ROB in case of a long-latency data cache miss during SMT execution depends on the fetch policy only, and not the job mix. In addition, the probabilistic model assumes that the behavior of a job is uniform over time. This assumption turns out to be a good approximation on average, but for some specific jobs and mixes, the impact of data cache misses on waiting cycles is underestimated by the model. This is due to bursty data cache/TLB miss behavior: a burst of data cache/TLB misses is more likely to fill up the ROB than isolated misses.

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. Table II shows the error and the empirically derived γ parameter for different fetch policies. As expected, the γ parameter is higher for policies that do not explicitly prevent threads from occupying all ROB entries (e.g., ICOUNT), while a policy like flush, which flushes all instructions of a thread when a long-latency load miss occurs, has a small γ value. Note that this not necessarily means that that fetch policy performs better; in the case of the flush policy, performance can be low due to a significant loss in memory-level parallelism [Eyerman and Eeckhout 2007].

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

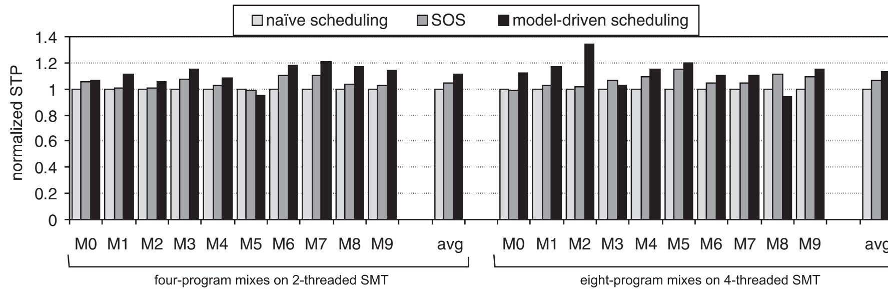

Fig. 2. Optimizing for system throughput (STP) for four-program mixes on a two-threaded SMT (left) and eight-program mixes on a four-threaded SMT (right).

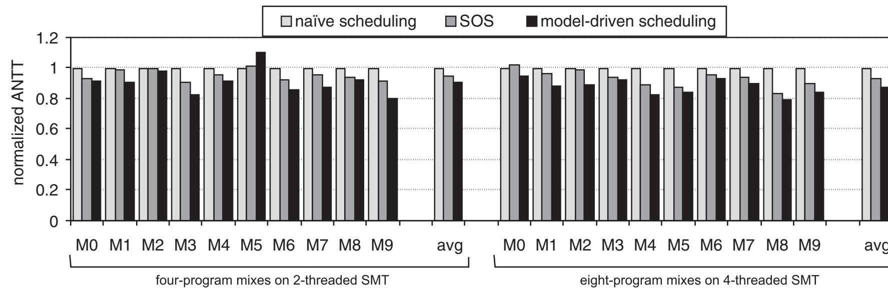

Fig. 3. Optimizing for job turnaround time (ANTT) for four-program mixes on a two-threaded SMT (left) and eight-program mixes on a four-threaded SMT (right).

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, for instance, 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 coschedules jobs in a round-robin manner, and (ii) Sample, Optimize, Symbios (SOS) proposed by Snavely and Tullsen [2000] (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, that is, $m = 2n$. 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.

For a couple job mixes, we observe a decrease in STP over naive scheduling (see M5 for four-program mixes and M8 for eight-program mixes), or an increase in ANTT (see M5 for four-program mixes). The reason is a combination of inaccurate cycle stack estimations and symbiosis modeling error. The per-thread cycle accounting architecture has an error of on average 7% for 2 SMT contexts and 11% for 4 SMT contexts, with the largest error for the last-level cache component [Eyerman and Eeckhout 2009]. The

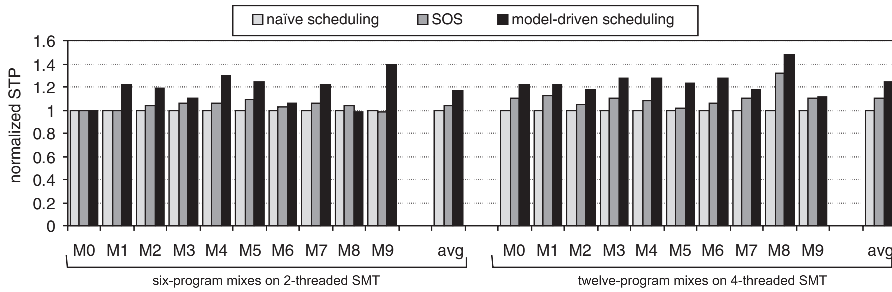

Fig. 4. Optimizing for system throughput (STP) for six-program mixes on a two-threaded SMT (left) and twelve-program mixes on a four-threaded SMT (right).

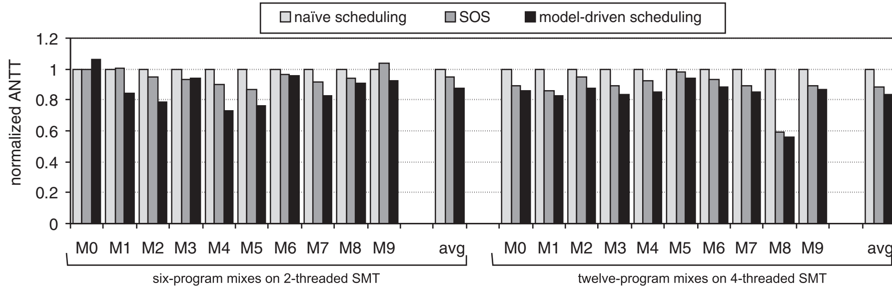

Fig. 5. Optimizing for job turnaround time (ANTT) for six-program mixes on a two-threaded SMT (left) and twelve-program mixes on a four-threaded SMT (right).

probabilistic job symbiosis model proposed in this article also shows the largest error for the data cache components, as described in the previous section. These errors occasionally enforce each other, leading to inaccurate performance estimates and suboptimal job coschedules. Improving cycle stack and/or symbiosis modeling accuracy is likely to improve scheduling performance for these job mixes. In addition, for the job mixes that show a decrease in STP over naive scheduling (see M8 for four-program mixes), we found that some jobs almost never get scheduled in the model-driven scheduling, although they exhibit good progress in SOS. Unlike naive and SOS scheduling, model-driven scheduling does not enforce each job to be scheduled an equal amount of time. This, in combination with optimizing STP without taking into account fairness, implies that jobs that do not show good symbiosis with other jobs at the beginning of their execution will (almost) never be scheduled again. As a consequence, these jobs will never proceed to a next phase in their execution which could show better symbiosis. Occasionally enforcing the scheduling of rarely scheduled threads will most likely solve this problem. Nevertheless, we found that model-driven scheduling leads to significantly better SMT performance on average.

It is also 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 8-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 coschedules to evaluate symbiosis but instead predicts symbiosis, which eliminates the sampling overhead of running suboptimal coschedules 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 coschedules but can evaluate a large number of possible coschedules. 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 [Snavely and Tullsen 2000] and gradually add features to arrive at model-driven scheduling; the deltas between the intermediate scheduling algorithms illustrate the importance of each of the given contributors. We consider the following job scheduling algorithms.

(a) *SOS with heuristic* is the SOS approach as proposed in Snavely and Tullsen [2000] 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 multithreaded 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 coschedules. It considers 10 possible coschedules and uses the probabilistic symbiosis model to gauge symbiosis but does not evaluate symbiosis by executing the coschedule. 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 article and evaluates all possible coschedules through probabilistic symbiosis modeling. The delta between (d) and (c) quantifies the impact of not being limited by the small number of job coschedules to choose from.

Figure 6 reports the achieved STP for each of these 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 coschedules 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 coschedules (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 suboptimal coschedules 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

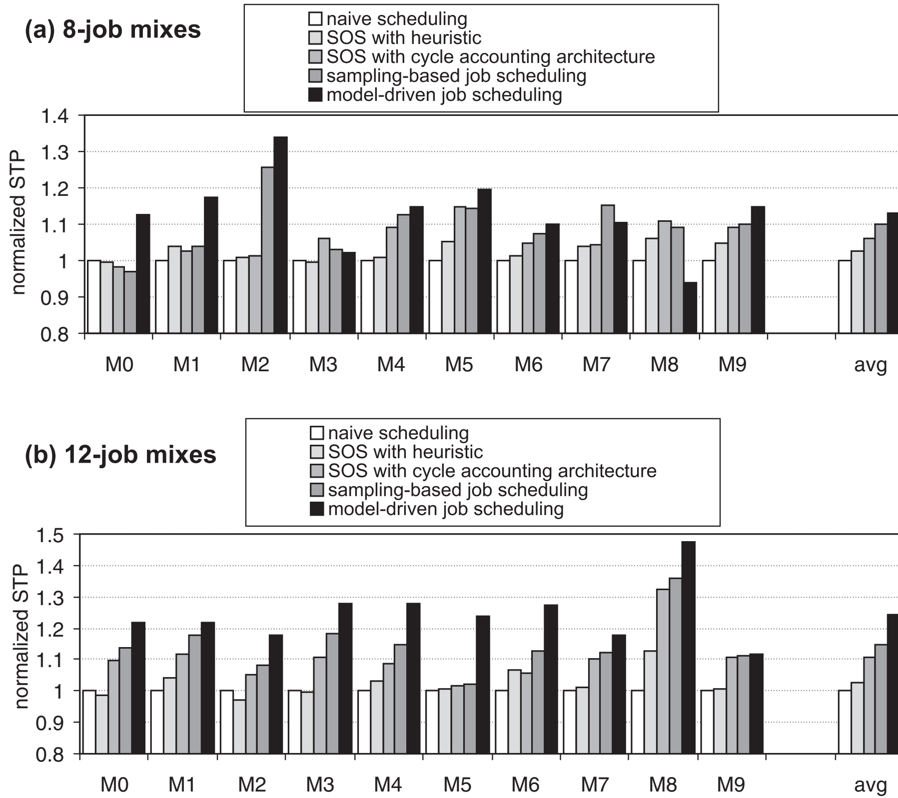

Fig. 6. Understanding where the performance improvement comes from: STP for a number of symbiotic job scheduling algorithms, assuming (a) an 8-job mix and (b) a 12-job mix on a four-threaded SMT processor.

the 12-job mixes being able to evaluate a large number of possible coschedules 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 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 interarrival 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

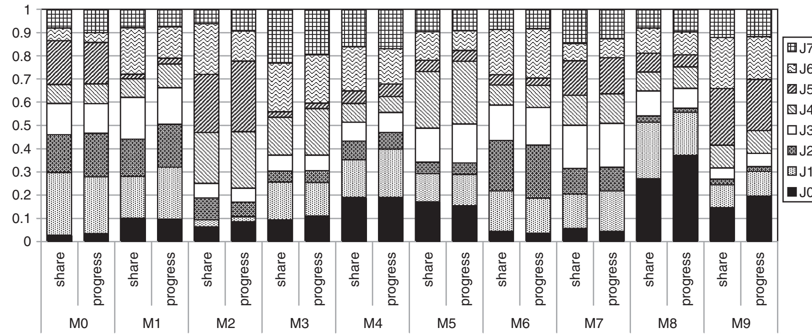

Fig. 7. Comparing normalized progress against relative shares for model-driven proportional-share job scheduling.

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. The scheduling method also differs from the fixed job mix case. In addition to rescheduling when a timeslice ends, we now also reschedule every time a job enters the system or finishes execution. We schedule each new job for one timeslice at the time it enters, to be able to measure its CPI stack, because that is needed as an input for the model. From then on, it is scheduled whenever the model predicts that it improves ANTT the most.

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 job turnaround time 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.

9.6. Job Coschedule Analysis

The scheduling algorithm presented in this article relies on a per-thread cycle accounting architecture for SMT processors. This infrastructure is not (yet) implemented in current SMT processors, and therefore the scheduling algorithm cannot be readily used in practice on real hardware. It is therefore interesting to understand what types of jobs are typically coscheduled by the algorithm: this can give us insight into what makes a good symbiotic coschedule, which could ultimately lead to symbiotic job scheduling heuristics that could be used on existing hardware.

In order to do so we use synthetic workloads rather than real benchmarks, the reason being that synthetic workloads allow for changing their behavioral characteristics at will, which facilitates the analysis. The synthetic workloads are characterized using

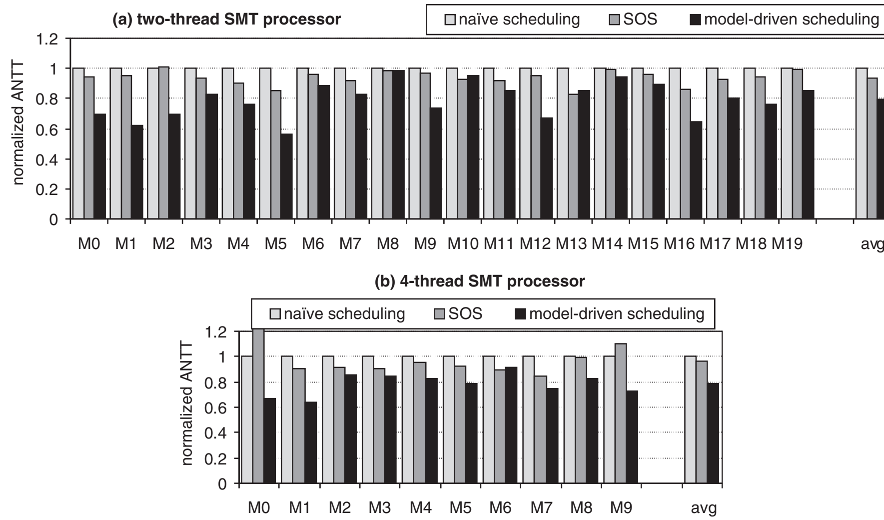

Fig. 8. Evaluating model-driven job scheduling for a dynamic job mix for (a) a two-thread SMT processor and (b) a four-thread SMT processor.

normalized cycle stacks, which represent where time is spent. A normalized cycle stack consists of a base component, which represents the fraction of time where useful work gets done, along with a number of “lost” cycle components due to miss events, such as cache misses, branch mispredictions, etc. The cycle component of interest in this study is the cache miss component. The reason is that cache misses represent opportunities for SMT to exploit symbiosis, that is, the processor can get useful work done for a thread while servicing the miss event for another thread. In particular, for this work, we focus on the LLC miss component. A similar analysis can be done for instruction cache misses and TLB misses, however, we did not consider this here because the LLC miss component is far larger than the instruction cache miss and TLB miss components for our set of benchmarks. Moreover, since instruction cache misses and TLB misses have a similar impact on symbiosis as LLC misses, what is referred to as the LLC miss component hereafter can also be interpreted as the sum of the LLC miss, instruction cache miss and TLB miss components. Hence, in this study, and without loss of generality, we characterize a workload’s behavior by a base cycle component along with an LLC miss component.

We now create four synthetic workloads with varying degree of memory-intensiveness by picking a (normalized) LLC miss component of 0.2, 0.4, 0.6, and 0.8. We will refer to these workloads as jobs 0, 1, 2, and 3, respectively. (Note that picking an LLC miss component also sets the base component to one minus the LLC miss component.) We then apply the job symbiosis model as described earlier in the article to each possible coschedule to estimate its performance. Figure 9 shows the resulting estimated STP for the six different two-job coschedules out of these four synthetic workloads. Perhaps unsurprisingly, the highest throughput is predicted when coscheduling the two most memory-intensive jobs (jobs 2 and 3). The intuition is that the more memory-intensive a job is, the more the other job can make progress while servicing memory requests for the first job. By consequence, in order to maximize system throughput, one should always coschedule the most memory-intensive jobs. (Determining whether a job is memory-intensive can be done through offline analysis, or could possibly be done online using performance counters.)

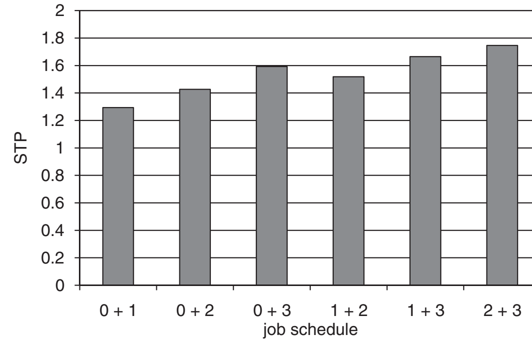

Fig. 9. STP for all 2-job schedules out of 4 synthetic jobs with normalized memory components 0.2, 0.4, 0.6 and 0.8 for job 0, 1, 2 and 3, respectively.

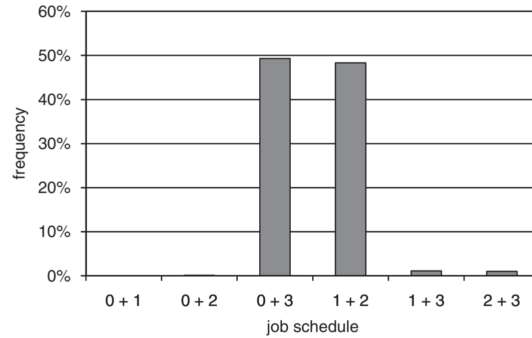

Fig. 10. Frequency of selected job schedules while minimizing ANTT (same configuration as in the previous graph).

Although prioritizing memory-intensive jobs maximizes system throughput, it may starve non-memory-intensive jobs, which is undesirable from a fairness perspective. In other words, it is important to balance system throughput and job turnaround time. A metric that captures the impact on job turnaround time is ANTT, which penalizes starving jobs. Therefore, we set up an experiment using the same 4 jobs, and we minimize ANTT by selecting a new coschedule every timeslice. Figure 10 shows the frequency of the selected coschedules. Surprisingly, the coschedule that maximizes system throughput (i.e., coschedule of jobs 2 and 3) is almost never chosen. Instead, the most memory-intensive job is combined with the least memory-intensive job (jobs 3 and 0, respectively); the two remaining jobs (jobs 1 and 2) are also coscheduled. The runtime schedule alternates between these two coschedules. The alteration can be explained as follows: if jobs 0 and 3 are coscheduled, then in order to maintain fairness, jobs 1 and 2 have to be coscheduled in the next timeslice. Interestingly, coscheduling the memory-intensive jobs (jobs 2 or 3) and the compute-intensive jobs (jobs 0 and 1) does not minimize job turnaround time. The reason is that although coscheduling the memory-intensive jobs (jobs 2 and 3) yields higher system throughput, coscheduling the compute-intensive jobs (jobs 0 and 1) yields less throughput, relatively speaking, compared to coscheduling mixed memory-intensive versus compute-intensive jobs (a coschedule of jobs 0 and 3, and jobs 1 and 2), see also Figure 9.

Figure 11 further validates this finding using a total of 100 random job workloads (with random LLC miss component) for 6 jobs and 2 SMT contexts, and 6 jobs and 4 SMT contexts; the graphs in Figure 11 show the relative frequency of the job

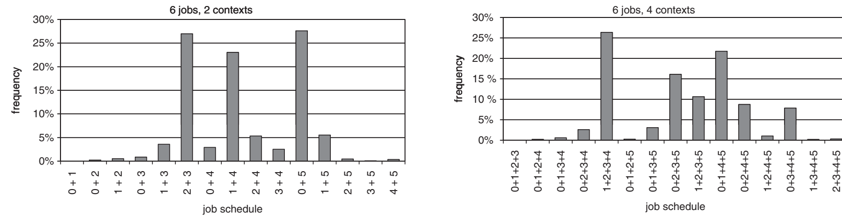

Fig. 11. Frequency of selected job schedules while minimizing ANTT (average across 100 random job workloads).

coschedules. All jobs are sorted by their LLC miss component (i.e., job 0 always has the lowest LLC miss component of all jobs, and job 5 has the highest). The conclusion from these results is that mixed schedules, consisting of both memory-intensive and compute-intensive jobs, yield shorter job turnaround times than alternately coscheduling memory-intensive and compute-intensive jobs. In conclusion, to optimize ANTT (i.e., minimize the individual job turnaround time degradation), coschedules should be made of memory-intensive and compute-intensive jobs.

9.7. Global Scheduling

So far, we considered local scheduling, that is, we determine what is the optimum coschedule for the next timeslice, and we do not optimize beyond the next timeslice. We now study whether global scheduling or optimizing for the next few timeslices yields any performance benefit over local scheduling. The rational is that global scheduling may select a nonlocal optimum coschedule for the next timeslice which enables achieving a global optimum in the longer term across multiple timeslices. Global scheduling might be of interest in batch-style systems in which job mixes do not change frequently, such as a supercomputer facility.

Say global scheduling optimizes over the next d timeslices, with d the depth of the global scheduling. This means that, with j jobs and c contexts, we now have $\binom{j}{c}^d$ possible coschedules for global scheduling, in contrast to $\binom{j}{c}$ coschedules for local scheduling. The search space is thus much larger for global scheduling, but could potentially lead to better performing schedules.

To evaluate the potential of global scheduling, we redo the experiments described in the previous section, and we vary the global scheduling depth d from 1 (local scheduling) to 10. Interestingly, these experiments showed that global scheduling does not provide any performance benefit: ANTT decreases by less than 0.1% for depth 10 compared to depth 1 (local scheduling). The explanation can be found by analyzing the selected coschedules, see Figure 12 for the 4 jobs and 2 contexts case, for depths $d = 1$, 2, 5 and 10. There seems to be no shift in the selected coschedules: the two most commonly selected coschedules for local scheduling are also selected for global scheduling, and their relative importance even slightly increases as scheduling depth is increased. The only benefit from global scheduling is that the optimal coschedules are selected sooner, whereas local scheduling converges somewhat slower to selecting the optimal coschedules.

We conclude there is no benefit in global scheduling over local scheduling, which means that there is no need to explore more schedules than the number of combinations for the next schedule. Because our experiment using synthetic workloads, which does not account for the extra overhead due to the enlarged search space, shows no benefit, we did not perform experiments with real benchmarks. Moreover, for real workloads, global scheduling may yield even worse performance compared to local

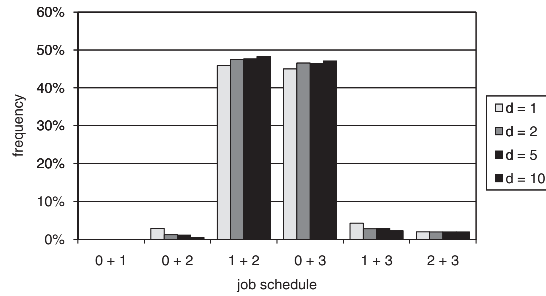

Fig. 12. Frequency of selected job coschedules for scheduling depth 1, 2, 5 and 10 in a global scheduling policy.

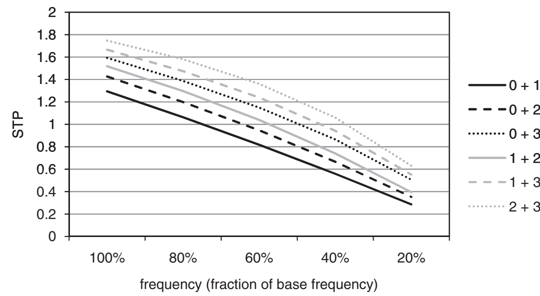

Fig. 13. STP as a function of frequency for different job coschedules.

scheduling. Global scheduling revisits its schedule every d timeslices in contrast to local scheduling which revisits its schedule every timeslice. This may lead to global scheduling reacting to time-varying execution behavior later, unless one would be able to predict and proactively react to upcoming phase changes.

9.8. SMT Scheduling and DVFS

Dynamic voltage and frequency scaling (DVFS) allows for trading off performance versus power consumption. Reducing the voltage of a processor reduces its dynamic and static power consumption, but also requires the clock frequency to decrease. This makes programs run slower, but consume less power. It is well known that memory-intensive programs are affected less by DVFS than compute-intensive programs. This is interesting for SMT scheduling: from a job symbiosis point of view it is beneficial to coschedule memory-intensive jobs, as discussed in the previous sections. One might thus expect some synergy between job symbiosis scheduling and DVFS for optimizing energy efficiency: by coscheduling memory-intensive jobs while lowering voltage/frequency, one could potentially improve both throughput and reduce power consumption; similarly, when coscheduling compute-intensive jobs, increasing voltage/frequency may compensate for the performance loss. This intuitive observation motivated us to study the synergy between job symbiosis scheduling and DVFS.

To do so, we again consider four jobs with a normalized LLC miss component of 0.2, 0.4, 0.6, and 0.8, for jobs 0, 1, 2, and 3, respectively. Figure 13 quantifies throughput for the six different 2-context combinations out of these 4 jobs, as a function of processor frequency. We consider five frequencies: the base frequency, and 80%, 60%, 40%, and 20% of the base frequency. STP is defined here as the total progress made relative to single-threaded progress at the base frequency, and reflects the impact of both SMT and DVFS.

Several interesting observations can be made from this figure. First, coscheduling memory-intensive workloads yields the highest SMT throughput across the entire frequency range. Coscheduling compute-intensive workloads yields the lowest throughput, with the mixed coschedules in between. Second, when going from the base frequency to 80% of the base frequency, the throughput decrease is less severe for a memory-intensive coschedule than for a compute-intensive schedule (-0.17 versus -0.23, respectively). This is because memory-intensive jobs are less susceptible to frequency scaling than non-memory-intensive jobs. As a result, the performance difference between the best and worst performing coschedule grows with decreasing clock frequency and voltage. The third observation is that at lower frequencies, for instance, when going from 40% to 20% of the base frequency, the decrease in throughput is now larger for the memory-intensive coschedule than for the compute-intensive coschedule (-0.43 versus -0.27). This can be explained by the fact that as frequency decreases, the memory component remains constant, while the computation component increases. This means that memory-intensive jobs become more and more compute-intensive, as the memory component decreases, relatively speaking. This causes memory- and compute-intensive jobs to "converge" at very low clock frequencies, that is, all the jobs become compute-intensive and most of their time is spent doing computation work.

Another interesting observation is that at 60% of the base frequency, the STP of the best performing coschedule (the one consisting of the two most memory-intensive jobs) is higher than that of the worst performing coschedule at the base frequency. This may open perspectives, as we can now take advantage of the fact that memory-intensive jobs are affected less by both SMT coscheduling and DVFS: as mentioned before, one could coschedule memory-intensive jobs and run them at a low frequency, and coschedule compute-intensive jobs and run them at a high frequency. Like that, we can possibly obtain large energy savings with a small impact on performance only. This would suggest that SMT job scheduling and DVFS should cooperate.

To better understand the synergestic benefits of SMT job scheduling and DVFS, we again use the same 4-job, 2-context example as before, and we consider two policies. In the first, so-called *combined* policy, SMT scheduling and DVFS are done independently, that is, SMT scheduling is done using the model as previously presented in this article, and frequency is set independently of the selected schedule. The DVFS policy used is to execute one fraction of the time at one frequency, and the other fraction at another neighboring frequency (i.e., one frequency step higher or lower). This policy yields optimal DVFS settings if the program behavior is homogeneous [Ishihara and Yasuura 1998]. The second policy is a *cooperative* policy that cooperatively explores the optimum SMT coschedule and the optimum clock frequency. More specifically, it selects the coschedule and frequency that optimizes ANTT for every timeslice. A cooperative policy needs to search a larger solution space which obviously incurs a higher overhead; this experiment is a limit study and does not take into account the overhead.

The results of this experiment are shown in Figure 14. Power consumption is plotted against ANTT. The gray curve connects results for the combined policy. The leftmost point reflects the case of executing all coschedules at the base frequency. For the second point to the left, the base frequency is assumed half of the time and 80% of the base frequency is assumed for the other half of the time, effectively running at 90% of the base frequency. The next point is generated by executing all coschedules at 80% of the base frequency, and at the fourth point the processor is clocked half of the time at 80% of the base frequency and half of the time at 60%, etc. The last gray point represents all coschedules executed at 40% of the base frequency. The black dots are generated using the cooperative policy. For the first point a maximum 10% performance decrease was allowed while minimizing power consumption, for the second point the maximum performance decrease is 20%, 30% for the third point, and 40% and 50%

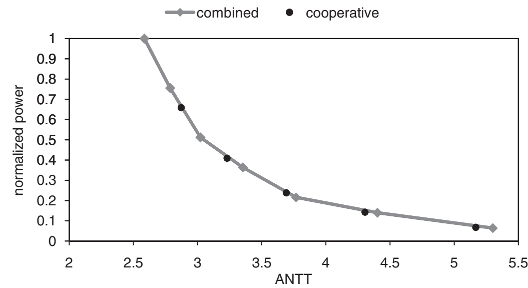

Fig. 14. ANTT versus power for the combined and cooperative policies.

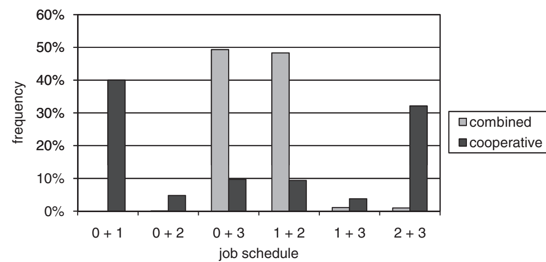

Fig. 15. Frequency of selected schedules for the combined policy versus the cooperative policy.

for the remaining points. Surprisingly, the cooperative policy does not outperform the combined policy. In other words, SMT scheduling and DVFS are independent problems that, when solved separately, yield a global optimum when combined. Interestingly, however, when looking at Figure 15, which shows the selected schedules for the two policies at a 30% performance degradation, we observe that the selected coschedules are very different for the two policies. The cooperative policy most frequenctly selects the extreme coschedules, whereas the combined policy barely selects these coschedules. In spite of the drastic shift in selected coschedules, there is no significant improvement in energy efficiency. We can conclude that SMT scheduling and DVFS can be performed completely independently. For completeness, we also performed experiments using global scheduling and the combined SMT/DVFS model, but again, no considerable gains were obtained.

10. RELATED WORK

10.1. Symbiotic Job Scheduling

Snavely and Carter [2000] 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 [2000] extended the SOS approach to SMT processors, and Snavely et al. [2002] 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 article.

Several other proposals have been made for symbiotic job scheduling. Settle et al. [2004] drive symbiotic job scheduling by monitoring activity in the memory subsystem. El-Moursy et al. [2006] monitor contention in the register file, functional units and L1 caches. Parekh et al. [2000] monitor cache miss rates and IPC. Bulpin and Pratt [2005] build an empirical model that predicts single-threaded progress based on hardware performance counter data. The key difference between these prior approaches and this

article 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 coschedules will result in positive symbiosis.

Other papers study job coscheduling in a different setting. Tam et al. [2007] coschedule threads from a multithreaded workload on the same chip in a multiprocessor environment based on shared memory access patterns. Jain et al. [2002] study symbiotic scheduling of soft real-time applications on SMT processors. Fedorova et al. [2006] find that non-work-conserving scheduling, that is, 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 [VMware 2004]. It assumes a simple accounting mechanism: it assumes that jobs coexecuting 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 coscheduled 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.

10.2. Multithreaded Processors

This article 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 coscheduled 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.

10.3. Improving Shared Resource Utilization

A large body of work has been done on improving shared resource utilization for both SMT and multicore processors. Tullsen et al. [1996] 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 [Tullsen and Brown 2001], MLP-aware flush [Eyerman and Eeckhout 2007], DCRA [Cazorla et al. 2004], hill-climbing [Choi and Yeung 2006], runahead threads [Ramirez et al. 2008], etc.

Chandra et al. [2005] 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 [2006] 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.

10.4. QoS Management in Multithreaded Processors

A number of studies have been done on improving quality-of-service (QoS) in multithreaded processors. Cazorla et al. [2004b, 2006] 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 [2003] describes an SMT processor architecture that combines OS priorities with thread efficiency heuristics (outstanding instruction counts, number of outstanding branches, number of data cache misses) to provide a dynamic priority for each thread scheduled on the SMT processor. The IBM POWER5 [Boneti et al. 2008; Gibbs et al. 2005] 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 spinlock, or for temporarily decreasing the priority of a process spinning for a lock, etc. Gabor et al. [2007] propose fairness enforcement on coarse-grained switch-on-event (SOE) multithreaded processors.

11. CONCLUSION

Job coscheduling by system software has a significant impact on overall SMT processor performance. Symbiotic job scheduling, which seeks to exploit the positive symbiosis between coexecuting jobs, can lead to substantially higher system throughput and lower job turnaround time. This article addressed the fundamental problem in symbiotic job scheduling and proposed probabilistic job symbiosis modeling for estimating the symbiosis between jobs in a coschedule without having to execute the coschedule. 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 coschedule execution to evaluate symbiosis, (ii) continuously readjusting the job coschedule, (iii) evaluating a large number of possible coschedules at very low overhead, (iv) tracking and predicting single-threaded progress during multithreaded 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. We also demonstrate that global scheduling over multiple timeslices does not yield better performance than local scheduling which schedules the next timeslice only. Finally, we found that SMT job scheduling and DVFS are independent of each other, that is, a combined scheme that optimizes job scheduling versus energy efficiency through DVFS independently performs equally well as a cooperative scheme that optimizes job scheduling and energy efficiency through DVFS simultaneously.

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

ACKNOWLEDGMENTS

We thank the reviewers for their constructive and insightful feedback.

REFERENCES

BONETI, C., CAZORLA, F. J., GIOIOSA, R., BUYUKTOSUNOGLU, A., CHER, C.-Y., AND VALERO, M. 2008. Software-controlled priority characterization of POWER5 processor. In *Proceedings of the International Symposium on Computer Architecture*. 415–426.

BULPIN, J. R. AND PRATT, I. 2005. Hyper-threading aware process scheduling heuristics. In *Proceedings of the USENIX Annual Technical Conference*. 103–106.

CAZORLA, F. J., KNIJNENBURG, P. M. W., SAKELLARIOU, R., FERNÁNDEZ, E., RAMIREZ, A., AND VALERO, M. 2006. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput. 55*, 7, 785–799.

CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNÁNDEZ, E. 2004a. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. 171–182.

CAZORLA, F. J., RAMIREZ, A., VALERO, M., KNIJNENBURG, P. M. W., SAKELLARIOU, R., AND FERNÁNDEZ, E. 2004b. QoS for high-performance SMT processors in embedded systems. *IEEE Micro 24*, 4, 24–31.

CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 2005. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*. 340–351.

CHOI, S. AND YEUNG, D. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. 239–250.

COTA-ROBLES, E. 2003. Priority based simultaneous multi-threading. United States Patent No. 6,658,447 B2.

EL-MOURSY, A., GARG, R., ALBONESI, D., AND DWARKADAS, S. 2006. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.

EYERMAN, S. AND EECKHOUT, L. 2007. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 240–249.

EYERMAN, S. AND EECKHOUT, L. 2008. System-level performance metrics for multi-program workloads. *IEEE Micro 28*, 3, 42–53.

EYERMAN, S. AND EECKHOUT, L. 2009. Per-thread cycle accounting in SMT processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–144.

FEDOROVA, A., SELTZER, M., AND SMITH, M. D. 2006. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*.

GABOR, R., WEISS, S., AND MENDELSON, A. 2007. Fairness enforcement in switch on event multithreading. *ACM Trans. Architect. Code Optim. 4*, 3, 34.

GIBBS, B., ATYAM, B., BERRES, F., BLANCHARD, B., CASTILLO, L., COELHO, P., GUERIN, N., LIU, L., MACIEL, C. D., SOSA, C., AND THIRUMALAI, C. 2005. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In P*roceedings of the International Symposium on Low Power Electronics and Design*. 197–202.

JAIN, R., HUGHES, C. J., AND ADVE, S. V. 2002. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*. 134–145.

LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 164–171.

PAREKH, S., EGGERS, S., LEVY, H., AND LO, J. 2000. Thread-sensitive scheduling for SMT processors. Tech. rep., University of Washington.

QURESHI, M. K. AND PATT, Y. N. 2006. 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*. 423–432.

RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques.* 15–26.

RAMIREZ, T., PAJUELO, A., SANTANA, O. J., AND VALERO, M. 2008. Runahead threads to improve SMT performance. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*. 149–158.

SETTLE, A., KIHM, J., JANISZEWSKI, A., AND CONNORS, D. 2004. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 63–73.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.

SNAVELY, A. AND CARTER, L. 2000. Symbiotic job scheduling on the MTA. In *Proceedings of theWorkshop on Multi-Threaded Execution, Architecture and Compilers*.

SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 234–244.

SNAVELY, A., TULLSEN, D. M., AND VOELKER, G. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 66–76.

TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the European Conference in Computer Systems*. 47–58.

TUCK, N. AND TULLSEN, D. M. 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 26–34.

TULLSEN, D. 1996. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference.*

TULLSEN, D. M. AND BROWN, J. A. 2001. Handling long-latency loads in a simultaneous multi-threading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*. 318–327.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture.* 191–202.

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 392–403.

VMWARE 2004. HyperThreading Support in VMware ESX Server 2.1. VMware.

Received June 2011; revised December 2011; accepted December 2011