UC San Diego

UC San Diego Previously Published Works

Title

Making the Most of SMT in HPC

Permalink

https://escholarship.org/uc/item/19z4c9d5

Journal

ACM Transactions on Architecture and Code Optimization, 11(4)

ISSN

1544-3566

Authors

Porter, Leo
Laurenzano, Michael A
Tiwari, Ananta
et al.

Publication Date

2015-01-09

DOI

10.1145/2687651

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Making the Most of SMT in HPC: System- and Application-Level Perspectives

LEO PORTER, EP Analytics and the University of California, San Diego
MICHAEL A. LAURENZANO, EP Analytics and the University of Michigan, Ann Arbor
ANANTA TIWARI, EP Analytics and the San Diego Supercomputer Center
ADAM JUNDT, EP Analytics
WILLIAM A. WARD, JR. and ROY CAMPBELL, Department of Defense HPC Modernization Program
LAURA CARRINGTON, EP Analytics and the San Diego Supercomputer Center

This work presents an end-to-end methodology for quantifying the performance and power benefits of simultaneous multithreading (SMT) for HPC centers and applies this methodology to a production system and workload. Ultimately, SMT's value system-wide depends on whether users effectively employ SMT at the application level. However, predicting SMT's benefit for HPC applications is challenging; by doubling the number of threads, the application's characteristics may change. This work proposes statistical modeling techniques to predict the speedup SMT confers to HPC applications. This approach, accurate to within 8%, uses only lightweight, transparent performance monitors collected during a single run of the application.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Modeling Techniques

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Simultaneous multithreading, high-performance computing, performance evaluation, energy evaluation

ACM Reference Format:
Leo Porter, Michael A. Laurenzano, Ananta Tiwari, Adam Jundt, William A. Ward, Jr., Roy Campbell, and Laura Carrington. 2014. Making the most of SMT in HPC: System- and application-level perspectives. ACM Trans. Architec. Code Optim. 11, 4, Article 59 (December 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2687651

1. INTRODUCTION

Simultaneous multithreading (SMT) [Tullsen et al. 1995] is a processor feature that allows a single physical core to simultaneously fetch and execute instructions from multiple threads. By interweaving instructions from those threads into the same pipeline, otherwise idle execution resources are employed to improve overall throughput.

This work was supported in part by a grant of computer time from the DoD High Performance Computing Modernization Program at the AFRL, ARL and ERDC DoD Supercomputing Resource Centers. This work was also supported in part by the HPCMP's PETTT program (Contract No: GS04T09DBC0017 though DRC) and by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476.
Authors' addresses: L. Porter, Computer Science and Engineering Department, University of California, San Diego; email: leporter@ucsd.edu; M. A. Laurenzano, Electrical Engineering and Computer Science Department, University of Michigan; email: michael.laurenzano@epanalytics.com; A. Tiwari, A. Jundt, and L. Carrington, EP Analytics; emails: {ananta.tiwari, adam.jundt, laura.carrington}@epanalytics.com; W. A. Ward, Jr. and R. Campbell, Department of Defense HPC Modernization Program; email: {william.ward, roy.campbell}@hpc.mil.
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.
© 2014 ACM 1544-3566/2014/12-ART59 $15.00
DOI: http://dx.doi.org/10.1145/2687651

However, SMT is not guaranteed to benefit any particular application, as threads competing for resources shared between SMT contexts may interfere with one another [Snavely and Tullsen 2000].

SMT is available on many high-performance computing (HPC) systems today, with at least 58 of the fastest 100 systems in the world (at the time of this writing) containing hardware that support it [Meuer et al. 2014]. The performance sensitivity of an application to SMT is highly variable and depends on many factors, an issue that has been well documented in the literature [Celebioglu et al. 2004; Curtis-Maury et al. 2005; Gepner et al. 2011; Grant and Afsahi 2005; Gray et al. 2006; Milfeld et al. 2003; Saini et al. 2011; Vega et al. 2013]. Whether employing SMT improves performance or not, using SMT contexts often increases processor activity and, therefore, increases power consumption. In some cases, the improved performance compensates for the increased power consumption for a net energy efficiency benefit. But in cases where SMT does not benefit performance, the increased time to solution combined with increased power consumption results in significantly diminished energy efficiency.

Because SMT can result in diminished performance and energy efficiency if used incorrectly at the application level, system designers and operators are put in the difficult position of deciding whether SMT should even be an option given to users (SMT can be disabled or enabled system-wide). Choosing whether to *allow users the option to use SMT* is more complicated than it may first appear. Proponents point to potential performance benefits, while opponents observe that these benefits are rarely obtained in practice. As a result, SMT is enabled in some HPC systems and disabled in others, often based on ad-hoc estimates of its efficacy or biases of the system operators.

At the same time, HPC users generally wish to improve their applications' performance, but they may ignore SMT because (1) they do not fully understand it, (2) its benefits are difficult to predict without that understanding, (3) experimenting with SMT for their application can be expensive in computing hours, and (4) those benefits may change with different application input sets, core counts, and system configurations. What HPC users need to better employ SMT is a robust, deployable methodology for estimating if, and to what extent, their application would benefit from using SMT.

This work examines the impact of SMT in HPC at both the system and application levels. First, we provide system operators with a system-level methodology for quantifying the performance and power benefits of SMT using workload-specific characteristics. One of the key insights of this methodology is that an important factor in understanding the impact of SMT is the degree to which application-level decisions about using SMT are made correctly. To address this, we provide users with a methodology for predicting the speedup of running an application on SMT, showing that simple performance monitor-driven models can be used to accurately assess the benefits of SMT for a wide spectrum of applications within both single- and multi-node execution scenarios. These models are designed so that they yield insights about the particular processor bottlenecks that are instrumental in predicting the performance benefit of SMT, and require only a single lightweight performance monitoring run of the application to obtain a prediction. Our specific contributions are as follows:

(1) **System-level Impact of SMT**—We introduce a methodology for quantifying the benefits of enabling or disabling SMT in an HPC system (Section 3). The methodology only requires the performance measurements of a representative set of applications and job sizes, much of which are already collected during system procurement or system testing, and summary statistics from standard job logging facilities.
(2) **Energy Implications of Enabling SMT**—We perform an investigation of the effects of simply enabling and disabling SMT (Section 4.3). We show that, *without utilizing SMT*, keeping it enabled has a small power overhead that persists across

multiple microarchitectures and process technologies in Intel hardware. This work is the first to document this overhead and its implications.

(3) **SMT Bottleneck Analysis—**We present a model-based analysis and quantification of the architectural features instrumental in determining whether SMT is beneficial for an HPC application (Section 5). We find commonality between Nehalem and Sandy Bridge, as SMT's effectiveness on both depends on some combination of memory bandwidth and instruction throughput. However, we find memory bandwidth is the more significant factor on Nehalem, while instruction throughput is the key factor on Sandy Bridge.

(4) **Predicting SMT's Performance Benefit—**We demonstrate that a model based on performance monitors can accurately predict the benefit of SMT (Section 5). We describe simple multiple regression models and more sophisticated rule-based machine learning models. The rule-based machine learning models achieve an average error of 7% and 8%, respectively, for Nehalem and Sandy Bridge systems.

(5) **Production Workload Study—**We employ our methodology on the 2011 production workload from the Department of Defense HPC Modernization Program (HPCMP) on an Intel Sandy Bridge-based system (Sections 4.4 and 5.6). Using our prediction model to guide application-level SMT deployment, we find that system-level performance and energy efficiency are within 97% of the performance and energy efficiency of an oracle predictor.

2. BACKGROUND AND RELATED WORK

2.1. Simultaneous Multithreading (SMT)

SMT [Tullsen et al. 1995] enables the execution of multiple threads on a single processor core by modifying and augmenting a small amount of hardware in the core to allow multiple threads to compete for otherwise idle resources. This approach is attractive to designers because it is relatively unintrusive and low in cost while potentially providing large increases in overall throughput. An additional benefit of SMT in the multicore era is that it enables simpler, homogeneous core designs to provide similar performance as more complex, heterogeneous core designs [Eyerman and Eeckhout 2014].

However, the competition for shared resources can be destructive to the performance of all threads and to overall throughput when threads contend for resources, such as out-of-order execution logic, functional units, caches, branch predictors, memory, and TLB. A variety of approaches have been proposed to mitigate this destructive behavior. Early work [Tullsen et al. 1996] proposed using instruction fetch logic to prioritize threads not receiving their share of shared resources. Subsequently, a number of architectural modificiations have been proposed that partition the shared resources to reduce conflicts [Cazorla et al. 2004; Choi and Yeung 2006; Raasch and Reinhardt 2003; Wang et al. 2008].

The first implementation of SMT in a general-purpose processor was Intel's Hyper-threading (HT), which appeared in 2002 within Xeon server processors [Koufaty and Marr 2003] and within the Pentium 4 line of desktop processors [Tuck and Tullsen 2003]. Subsequent implementations of SMT have appeared in other Intel processor lines: Itanium [Tian et al. 2003], Nehalem [Singhal and Engineer 2008], Sandy Bridge [Schöne et al. 2011] and Haswell. IBM has also included SMT in its Power processors, beginning with 2-way SMT in the Power5 [Mathis et al. 2005], up to 4-way SMT in the Power7 [Kalla et al. 2010] and 8-way SMT in the Power8.

2.2. Predicting Parallel and SMT Performance

There are a number of works that model the performance of parallel applications [Carrington et al. 2005; Kerbyson and Jones 2005; Marin and Mellor-Crummey 2004;

Snavely et al. 2002a], parallel application scalability [Barnes et al. 2008; Carrington et al. 2013; Kerbyson et al. 2001], and the impact of changing the input set of a parallel application [Lee et al. 2007]. However, surprisingly little work has focused on predicting whether a single parallel application stands to benefit from employing SMT. Much of the work in predicting SMT performance derives from efforts to create cooperative co-schedules among multiple applications.

Avoiding destructive interference by predicting symbiotic schedules has been studied by using a combination of sampling co-schedules and heuristics to select the best co-schedule [Snavely and Tullsen 2000; Snavely et al. 2002b]. To avoid sampling phases, Eyerman and Eeckhout [2012] create snapshots of performance (using per-thread cycle stacks from the architecture in Eyerman and Eeckhout [2009]) for use in probabilistically modeling co-schedules to select the top performer.

Examining CMPs of SMTs, DeVuyst et al. [2006] used runtime measurements to predict better co-schedules for multiprogram workloads; moving applications between cores to improve energy and/or performance. Rakvic et al. [2010] aim to improve performance and energy consumption of parallel workloads by identifying critical threads in a parallel region to use SMT to give higher priority to the critical thread for thread balancing. These techniques are complementary to ours.

Moseley et al. [2005] predict performance when co-scheduling two single-threaded applications in a multiprogram environment by using linear regression modeling and recursive partitioning of performance counters. We adopt a similar approach in this work, however, we apply statistical modeling to predict, for a fixed number of nodes, whether a parallel application would run better using n physical cores or 2n virtual cores. We are the first, to our knowledge, to address this question.

Curtis-Maury et al. [2006] modify multithreading libraries to dynamically profile applications using performance monitors on a small number of cores, predicting and enacting an effective OpenMP configuration. This work is complementary to ours in that our work aims to inform users to make better SMT choices rather than modifying multithreading libraries to improve runtime OpenMP decisions. Our work also applies to MPI and OpenMP alike and addresses internode communication present in large full-size HPC applications.

2.3. SMT in HPC

SMT is an important feature in HPC systems. At the time of writing, the majority of the most recent Top500 list [Meuer et al. 2014] contains hardware featuring an SMT implementation. There is a wealth of related work that documents the effects of SMT on the performance of various HPC applications [Celebioglu et al. 2004; Curtis-Maury et al. 2005; Gepner et al. 2011; Grant and Afsahi 2005; Gray et al. 2006; Milfeld et al. 2003; Saini et al. 2011; Vega et al. 2013]. We draw two conclusions from this literature. First, the variability in the impact of employing SMT is large. Second, the benefit of employing SMT is difficult to predict, varying significantly across computational areas and applications, and even within datasets and core counts for the same application. Supported by the literature, we draw similar conclusions when examining the benefit of SMT across a spectrum of HPC application and benchmark programs using shared memory (OpenMP) and message passing (MPI) parallelization approaches (see Section 4.2).

These results may lead to two conflicting conclusions. One might interpret the significant variation in runtime improvements as support for the idea that there is a substantial opportunity for users to selectively employ SMT to their advantage. Alternatively, the difficulty in determining whether a specific job will benefit from SMT may lead operators to conclude that employing SMT in just the right way is too difficult a

task for users to undertake, causing more harm than good. Aiding user decisions by predicting SMT's value is hence addressed in Section 5.

Although prior work has addressed energy and performance implications of SMT for certain groups of applications, this is the first work to report the increase in power consumption from merely *enabling* (not necessarily employing) SMT for the system. Additionally, this work is the first to address the fundamental question of whether to allow users the option of using SMT by enabling SMT throughout the HPC center.

3. MODELS FOR SMT: SYSTEM AND APPLICATION-LEVEL APPROACHES

This section provides a methodology for evaluating the trade-offs involved in deciding whether system operators should enable SMT for users. The methodology determines the impact SMT has on system-wide performance and energy by accounting for three primary factors:

(1) **Energy Impact of SMT:** Recent work has shown that using SMT increases power consumption by roughly 5% on Sandy Bridge systems [Schöne et al. 2011]. In Section 4, we revisit this claim for Sandy Bridge and Haswell systems and find that it is consistent with our measurements. As we document in Section 4.3.2, a further confounding factor is that merely *enabling* SMT on the system increases power consumption for both Sandy Bridge and Haswell systems. That is, even when running jobs that leave all SMT contexts idle, there is an increase in power draw over an identical job running on identical hardware that has SMT disabled. Therefore, enabling SMT in an HPC system causes the power consumed during *all* jobs to increase, including a slight increase in system power while the system is vacant. Please see Section 4.3.2 for our analysis of this effect. As such, each of these overheads (the cost of enabling and the cost of using SMT) are accounted for in our methodology.

(2) **Workload and System Sensitivity to SMT:** SMT's benefit, for both energy and performance, may be sensitive to a number of factors including the application, input set, number of cores chosen for a particular run, and the underlying hardware features that change from generation to generation. These factors are hence dependent on system workload and system hardware. We provide a method for summarizing the workload based on a representative subset of jobs and for measuring SMT's benefit for that workload.

(3) **User Accuracy:** A key factor in determining the value of SMT is how well users employ SMT. The accuracy of their decisions to employ SMT impacts not only the performance of their application, but system-wide energy consumption. As previously mentioned, SMT's benefit for an application can be hard to predict for users because it depends on a number of factors. To aid user decisions, we provide a system- and workload-aware modeling technique, which can provide users with a predicted benefit of using SMT with their application (at the size of a recent run).

To aid our analysis, Table I defines the three SMT states possible in an HPC system. For hardware that supports SMT, the operator may choose to disable the SMT thread contexts in BIOS (SMT^{off}), making SMT unavailable to users. The operator may also make the decision to enable SMT in BIOS, in which case users have the choice of leaving those SMT contexts idle (SMT^{idle}) or employing them (SMT^{used}).

Our methodology consists of operators supplying a set of representative applications and, for these applications, performance measurements when running with and without SMT (SMT^{used} and SMT^{idle}) on a range of core counts (this provides the speedup from SMT at these core counts). These representative applications are carefully

Table I. Summary of SMT-related System/Application States

State	Summary	Detailed Description
SMT^{off}	SMT disabled	SMT is supported by the processor, but is disabled by the operator in BIOS and is, hence, unavailable to users
SMT^{idle}	SMT enabled but idle	SMT is enabled by the operator, but thread/process management is employed to ensure that only one thread/process per physical core is used
SMT^{used}	SMT enabled and used	SMT is enabled by the operator, and jobs use all SMT contexts (virtual cores)

constructed into a workload, in order to match the characteristics of the actual system workload observed within the system's job logs. The speedup measurements for the applications are combined with power characteristics of the system running in the possible SMT states (SMT^{off}, SMT^{idle} and SMT^{used}) to produce an estimate of the energy impact of enabling SMT on the system as a function of how aptly users choose correctly between SMT^{idle} and SMT^{used}. Finally, as applications are run for speedup measurement, a prediction model is created so that users can submit jobs and receive feedback on the predicted performance had the job been run using SMT (for the same number of nodes).

The results of this methodology are twofold: (1) it provides the operator an understanding of whether SMT is worth the energy cost for the system workload, which is dependent on users accurately employing SMT, and (2) it provides a prediction model to aid user decisions. If the model's ability to predict SMT's value (or existing user accuracy) is less than the threshold where the system would benefit from SMT, an operator may elect to disable SMT system-wide. Conversely, if the model's prediction accuracy (or existing user accuracy) is higher than the threshold for the system to benefit from SMT, the operator should elect to enable SMT system-wide.

3.1. Quantifying the Impact of SMT at the System Level

To understand the impact to the overall system of enabling or using SMT, it is necessary to understand how SMT impacts particular applications as well as how SMT affects power draw. These factors are then combined to quantify the benefits of enabling SMT on an HPC system.

3.1.1. Power Costs. We begin by defining the power draw for a job j on the same number of physical cores when using (SMT^{used}) and when not using SMT (SMT^{idle}), each divided by the power draw of the same system with SMT disabled (SMT^{off}). We define these as C_j^{used} and C_j^{idle}, respectively. Similarly, we define C_{vacant}^{idle} as the power draw of idling all cores (including SMT contexts) on a vacant/jobless system divided by the power draw of the same vacant system with SMT disabled (SMT^{off}). We note that these values are normalized to SMT^{off} and reflect a notion of the power "cost" of SMT in each case. For example, a value for any of the costs (C) of 1.0 would indicate no increased power for SMT, and 1.20 would indicate a 20% cost increase. The inputs to these calculations are power measurements, which can be gathered directly by measuring power while applying different SMT states to the system/jobs.

3.1.2. Job-level Metrics. We use the standard definition of speedup to formulate the speedup of a job j for SMT^{idle} and SMT^{used} relative to SMT^{off} as S_j^{used} and S_j^{idle}, respectively. These definitions appear in Equation (1), where T_j is execution time of j. We note that for many of the applications we study in this work, the performance difference between SMT^{idle} and SMT^{off} is negligible (i.e., S_j^{idle} is very close to 1), as shown

in Section 4.3.2. We use S_j^{idle} rather than 1 throughout to generalize our methodology should the performance difference increase with future processor generations.

$$S_j^{used} = \frac{T_j^{off}}{T_j^{used}} \qquad S_j^{idle} = \frac{T_j^{off}}{T_j^{idle}} \tag{1}$$

Using the power and speedup, we define the normalized energy for a job j for SMT^{used} and SMT^{idle} in Equation (2), where a value for normalized energy below 1 reflects an energy savings.

$$E_j^{used} = \frac{C_j^{used}}{S_j^{used}} \qquad E_j^{idle} = \frac{C_j^{idle}}{S_j^{idle}} \tag{2}$$

We then define in Equation (3), of the possible choices a user can make with respect to SMT (SMT^{used} or SMT^{idle}), the *ideal*- and *worse*-case speedups for a job j.

$$S_j^{ideal} = max\left(S_j^{used}, S_j^{idle}\right) \qquad S_j^{worse} = min\left(S_j^{used}, S_j^{idle}\right) \tag{3}$$

Similarly, we define the ideal- and worse-case energy improvements for a job j in Equation (4). Note that idealized energy is formulated as a function of which choice produces the best performance outcome for the user.[1]

$$E_j^{ideal} = \begin{cases} E_j^{idle} & \text{if } S_j^{idle} \geq S_j^{used} \\ E_j^{used} & \text{otherwise} \end{cases} \qquad E_j^{worse} = \begin{cases} E_j^{used} & \text{if } S_j^{idle} \geq S_j^{used} \\ E_j^{idle} & \text{otherwise} \end{cases} \tag{4}$$

We incorporate next the probability P^{ideal} that users make the ideal choice (i.e., the choice that offers the best performance) between using SMT contexts (SMT^{used}) and not using SMT contexts (SMT^{idle}) when running jobs. Since P^{ideal} is a probability, $P^{ideal} \in [0, 1]$. Using P^{ideal}, we define the *expected* speedup and energy for a job j as a function of how likely the user is to select the better performing SMT state for the job j in Equations (5) and (6), respectively.

$$S_j^{expect} = P^{ideal} * S_j^{ideal} + (1 - P^{ideal}) * S_j^{worse} \tag{5}$$

$$E_j^{expect} = P^{ideal} * E_j^{ideal} + (1 - P^{ideal}) * E_j^{worse} \tag{6}$$

Next, we apply these definitions for all of the workload's jobs to ascertain the performance and energy consequences of enabling and disabling SMT for that workload.

3.1.3. Workload Formulation. We define a workload W as a set of jobs. Within a workload, each job is weighted to reflect its importance to the workload. We assign the weight R_j to job j based on the fraction of the CPU-hours consumed by jobs within the production system that resemble j. Ensuring that W is representative of the jobs run in production and describing the mechanics of how those weights are assigned is the subject of Section 3.1.4.

As HPC systems are not always running jobs and can often be underutilized, we incorporate the utilization level U of the system, where $U \in [0, 1]$. Bringing these components together, we can now define the expected overall performance benefit and

[1]As only one configuration (SMT_j^{used} or SMT_j^{idle}) can be selected for a given application, the decision to select the "ideal" based on performance inherently prioritizes performance over energy. One could prioritize energy over performance, modifying the "ideal" to be the better result based on energy instead.

the energy benefit of enabling SMT for the workload W in Equations (7) and (8), respectively, as a function of the probability that users make the ideal choice about using or idling the available SMT contexts on the system (factored into S_w^{expect} and E_w^{expect}).

$$S_W^{expect} = \sum_{w \in W} R_w * S_w^{expect} \tag{7}$$

$$E_W^{expect} = U \left(\sum_{w \in W} R_w * E_w^{expect} \right) + (1 - U) C_{vacant}^{idle} \tag{8}$$

These equations, along with the workload model described in the following, are used to evaluate the implications of enabling SMT for particular HPC systems and workloads.

3.1.4. Workload Modeling. The particular weighting scheme used for the jobs in a workload depends on the purpose for which the workload is being constructed. For this work, weights are assigned to jobs based on the CPU-hours the job consumes out of the total CPU-hours consumed by all jobs in the workload.

The workload for a production HPC system W^{actual} consists of all jobs run on that system over some period of time. Were we to construct a workload from the actual set of all jobs on the system, we would weight each job according to its CPU-hour usage. As production workloads may contain tens or hundreds of thousands of jobs over the course of a year and because our methodology requires running each job under both the SMT^{idle} and SMT^{used} scenarios, deploying it on a raw production workload in total is impractical. Therefore, we construct a workload W^{model} that possesses the same characteristics as W^{actual}, allowing us to study the behavior of the W^{actual} without resorting to rerunning every job in the workload.

The construction of W^{model} is guided by summary statistics of W^{actual} extracted from the job logs on the system to obtain information about the job size and runtime, as well as user surveys to determine computational domains for each job. Our goal in constructing W^{model} is to achieve similarity in the distribution of the physical core counts and the computational domains of the jobs. Our strategy for achieving this similarity focuses on selecting a set of jobs that are diverse in both of these dimensions. However, because applications generally cannot be run at arbitrary core counts, the selection of a reasonably small number of applications that mirrors the diversity of W^{actual} is difficult to perform by hand and thus is treated as an optimization problem. We then employ a hill-climbing approach [Aarts and Lenstra 1997] to solve the optimization problem, the result of which is an assignment of weights to jobs. In Section 4.4, we use this approach to develop a workload model for the 2011 production workload from the Department of Defense's HPC Modernization Program (HPCMP).

3.2. Application-level Performance Models to Inform User Decisions

On HPC systems, running applications with both SMT used and SMT idle can be expensive for users because it requires them to repeat jobs, possibly in the thousands of compute hours, to see if SMT is beneficial. As such, this section describes performance monitor-driven models for predicting the performance benefit of employing SMT at the application level. The input to these models are collected on a single non-SMT execution of the application. One such run allows the model to produce a prediction of the performance outcome of running that same application on the *same number of physical cores* using SMT (hence using 2× the number of threads).

An overview of our technique for building the performance model is presented in Figure 1. Models are built by leveraging the performance monitoring data from a

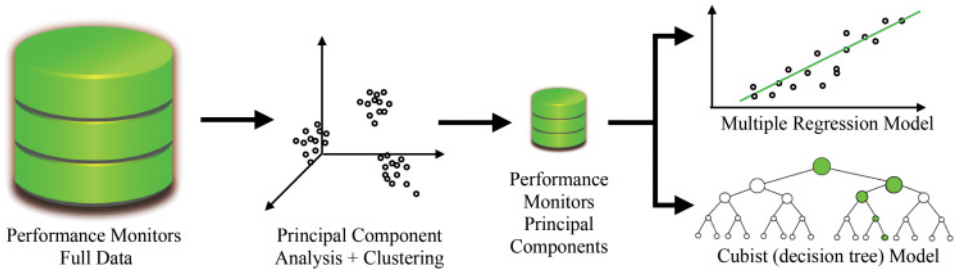


Fig. 1. Overview of model building methodology.

pool of application runs, described in Section 3.2.1. The resulting performance monitoring data from those runs is reduced in size by a principal component analysis and clustering methodology described in Section 3.2.2. Finally, the resulting principal components from the performance data are used to build statistical models described in Section 3.2.3.

3.2.1. Performance Monitors. All of the prediction models described in this work leverage hardware performance monitors, simple counts of microarchitectural events that are exposed to the ISA via model-specific registers. Our baseline models use a rich set of at least 40 hardware performance monitors (the exact number varies by system). These monitors provide information on the behavior of a number of architectural features as an application executes, including behavior within the memory hierarchy, floating point unit, branch predictor, instruction pipeline, and so forth. Later we present a streamlined approach that uses principal component analysis to narrow the number of hardware performance monitors to just nine.

In addition to hardware performance monitors, the multinode prediction models presented in this work also use simple summary statistics of the network communication events generated by the application. These summary statistics are collected in software using a monitoring library that wraps calls to the message passing interface (MPI) library [Tikir et al. 2009]. Collection of these communications statistics generates negligible extra overhead ($<$1%).

3.2.2. Principal Component Analysis. We extend each of the models by adopting an optimization known as *principal component analysis (PCA)* [Jolliffe 2005]. PCA is a statistical technique that converts a set of explanatory variables that may be correlated into a smaller number of noncorrelated explanatory variables, the *principal components*. When used effectively, PCA can be used to dramatically reduce the number of explanatory variables without losing (much of) the predictive quality of the model.

Our approach to extracting principal components combines PCA and K-means clustering [Duda and Hart 1973; Ding and He 2004]. First, we extract N principal components from the explanatory variables. The result of this first step is an N-element vector for each of the original explanatory variables, with each such vector describing the linear combination of principal components that makes up the explanatory variable. Second, we use K-means clustering on those N-element vectors, yielding clusters of explanatory variables that have similar principal components.

From each cluster found by K-means, we choose a single explanatory variable as the representative of that group. We then discard all other nonrepresentative explanatory variables and build models using only the remaining representatives. As we later show, downselecting the explanatory variables in this fashion has little negative impact on the quality of the models, yet makes it far easier to perform analysis such as the variable importance calculation described in Section 3.2.5 and implemented in Section 5.3.

3.2.3. Multiple Regression Models. Multiple linear regression is a statistical technique for modeling the relationship between a response variable and multiple explanatory variables [Draper and Smith 1981]. In this work, the explanatory variables are performance monitor measurements of the application running without SMT, as described earlier. The response variable is the predicted speedup of doubling the number of threads and executing the application using SMT context (i.e., using twice the number of threads/processes while still running on the same number of physical cores). Multiple linear regression models assume the form:

$$R = c_1E_1 + c_2E_2 + \cdots + c_nE_n \tag{9}$$

Building a linear regression model involves selecting values for the coefficients $c_1, c_2, \ldots, c_n$, typically choosing them in such a way that some measure of error is minimized. In this work, we select the coefficients that minimize root mean square error (RMSE).

3.2.4. Rule-based Machine Learning Models. Cubist [RuleQuest Research 2012] models are rule-based machine learning models, built using a tree of linear regression models. The intuition behind a cubist model is as follows: predictions are made using the linear model found at the leaf node of the tree, while the choice of leaf is determined by rules at the nonleaf nodes that are also based on linear models. Cubist models have the capability of explaining complex and nonlinear relationships, which are pervasive in processor architecture. For example, memory access time is a complex nonlinear relationship of working set size (among other factors) and has several "cliffs" at cache size boundaries.

In this work, we use cubist models in a manner similar to the regression models described earlier, where the model takes explanatory variables $E_1, E_2, \ldots, E_n$ and yields a response R, which is the speedup of running an application using SMT. Like the regression models, when building cubist models, they are built to minimize RMSE.

3.2.5. Variable Importance. One of the important insights we obtain from the models developed in this work is to assess how important each of the explanatory variables is to predicting the response variable. Variable importance analysis results in a score for how "influential" each explanatory variable is in calculating the result of the model. For our models, the variable importance tells us which hardware/software interactions are most important for determining how beneficial doubling the number of-threads and running an application using SMT is to performance.

Our approach to assigning variable importance in the cubist model is to use a linear combination of each explanatory variable based on its use in the rule conditions among nonleaf nodes and in the final model/leaf node. For the linear models, we assign variable importance by examining the relationship between each explanatory variable and the response by fitting a linear model to that relationship and finding the R^2 statistic of that model to an intercept-only model (an intercept-only model is akin to a regression built using only a single constant).

3.2.6. Preventing Model Overfitting. To avoid overfitting, we employ two techniques. First, we use "out-of-sample" model validation, wherein we divide the empirical samples into nonoverlapping training and test sets. The model is trained on the training subset and validated on the test set. Second, we use 10-fold cross-validation to produce the models. In k-fold cross-validation (in our case, $k = 10$), the training dataset is randomly partitioned into k subsets of approximately equal size. k different models are then constructed, each using $(k - 1)$ of the k partitions as training input so that one of the k sets can be set aside for model validation. Each of the k models are then validated against the validation set and the model that yields the minimum error is selected.

4. EVALUATION

We begin our evaluation by describing the experimental setup used throughout our evaluation, including the details of all test platforms, software and hardware apparatus, and application codes.

4.1. Experimental Setup

4.1.1. Platforms. We employ an array of test platforms in our experiments. The full-scale production HPC systems are:

—**HPC-NH** is an ERDC/HPCMP system composed of 1,920 Intel Nehalem-based compute nodes. Compute nodes are dual-socket systems with 24GB of memory, and are connected with a 4X DDR Infiniband interconnect. Each socket houses a quad-core Intel Xeon X5560 running at 2.8GHz that supports two-way Hyperthreading[2] (HT) per core for a total of 8 virtual cores per processor, or 16 virtual cores per node.
—**HPC-NH2** is an ARL/HPCMP system with 1,344 Intel Nehalem-based compute nodes. This system is nearly identical to HPC-NH with the same processors and interconnect; however, each node has 48GB of memory.
—**HPC-SB** is an AFRL/HPCMP system composed of 4,590 Intel Sandy Bridge-based compute nodes. Each of the compute nodes are dual-socket systems with 32GB of memory and are connected with a 14X FDR Infiniband interconnect. Each socket has an 8-core Xeon E5-2670 running at 2.6GHz that supports two-way HT per core for a total of 16 virtual cores per processor, or 32 virtual cores per node.

We also employ a series of single-node systems:

—**RCK-SB** is a single-node rack-mounted server that is configured as a dual-socket system with 32GB of memory. Each socket supports an 8-core Intel Xeon E5-2450 (Sandy Bridge) processor running at 2.1GHz with two-way HT on each core for a total of 32 virtual cores.
—**SRV-SB** is a single-node desktop server with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors running at 2.6GHz, with two-way HT on each core. The whole system has 32GB of memory and 32 virtual cores.
—**SRV-HW** is a single-node desktop server that has a single 4-core Intel i7-4770 (Haswell) processor. The i7-4770 runs at 3.4GHz, has 16GB of memory, and two-way HT on each core for a total of 8 virtual cores.

4.1.2. Application Codes. We use a number of benchmarks and applications in our experiments, including the NAS Parallel Benchmarks (NPBs) [Bailey et al. 1991], NERSC benchmarks [Antypas et al. 2008; Cordery et al. 2013], PARSEC [Bienia et al. 2008], OpenCV [Bradski 2000], and full-scale production HPC applications from the DoD TI-13 benchmark suite. Production applications include HYCOM [Chassignet et al. 2007], AVUS [Hoke et al. 2004], LAMMPS [Plimpton et al. 2007], ICEPIC [Mardahl et al. 2003], and CTH [Davis et al. 2007]. The microbenchmarks include a number of kernels from pcubed [Laurenzano et al. 2011] and polybench [Pouchet 2012]. Table II summarizes these applications, along with the parallelization model each application uses and the core count ranges used in this work.

4.1.3. Additional Experimental Setup Details. All applications on the large-scale systems **HPC-NH**, **HPC-NH2** and **HPC-SB** were compiled with the Intel compiler, while all applications on the single-node systems **RCK-SB**, **SRV-SB** and **SRV-HW** were compiled with the GNU compiler collection. For all experiments, Turbo mode is disabled. On all systems and for all experiments, the results presented are the average of multiple runs:

[2]Hyperthreading is Intel's proprietary implementation of SMT [Koufaty and Marr 2003].

Table II. Applications and Benchmarks

Group	Applications Included	Parallel Model	Phys. Core Min/Max
HPCMP Production	HYCOM, AVUS, LAMMPS, ICEPIC, CTH	MPI	256/1k
NPB	BT, CG, DC, EP, FT, IS, LU, MG, SP	MPI/OMP	8/1k
NERSC	amg, CoMD, gtc, miniFE, miniGhost	MPI	8/64
PARSEC	blacksholes, bodytrack, freqmine	OMP	8/32
OpenCV	2d_convolution, dwt53, histogram_equalization	OMP	8/32
microbenchmarks	pcubed, polybench	MPI/OMP	8/32

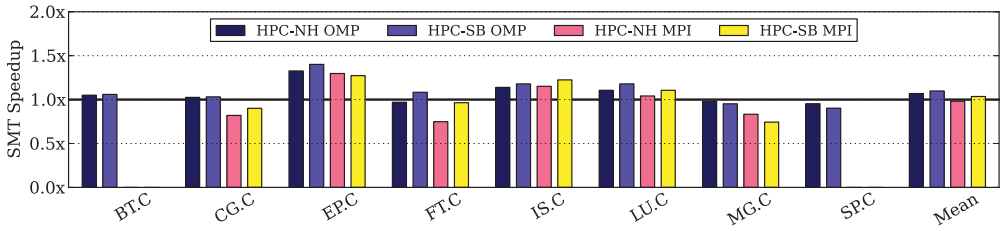


Fig. 2. Speedup from SMT for OpenMP and MPI NPBs on single dual-socket Nehalem (HPC-NH) and Sandy Bridge (HPC-SB) systems.

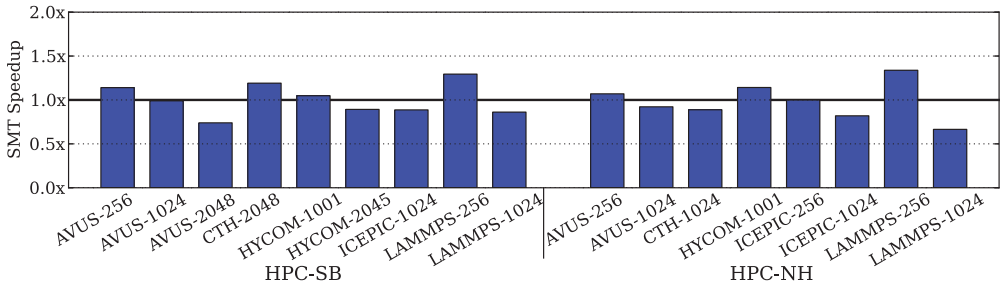


Fig. 3. Speedup using SMT for MPI applications at varying physical core counts on Nehalem (HPC-NH) and Sandy Bridge (HPC-SB) systems.

at least 3 for performance, at least 5 for power, and at least 6 performance counters (for performance counters, the closest three results were averaged to eliminate outliers). Power measurements are AC power measured at the supply for the entire system using a WattsUp meter [ThinkTank Energy Products Inc. 2014].

4.2. SMT Performance Characterization

We begin by examining the performance impact of using SMT for the selected benchmarks and applications. We first examine the NPBs on a single-node and then on larger runs of the NPBs and applications.

4.2.1. Single-node Performance Characterization. Figure 2 shows results for the NPBs run on a single-node within **HPC-SB** and **HPC-NH**. Within a single-node, a benchmark's benefit from SMT may change depending on which processors are in the system and whether the parallelization is done using OpenMP or MPI. A number of benchmarks consistently either benefit or do not benefit from SMT, though the degree by which they benefit changes depending on the particular configuration being run.

4.2.2. Multinode Performance Characterization. Figure 3 provides the speedup from SMT on large applications on our two HPC systems. In these scaling runs, strong scaling

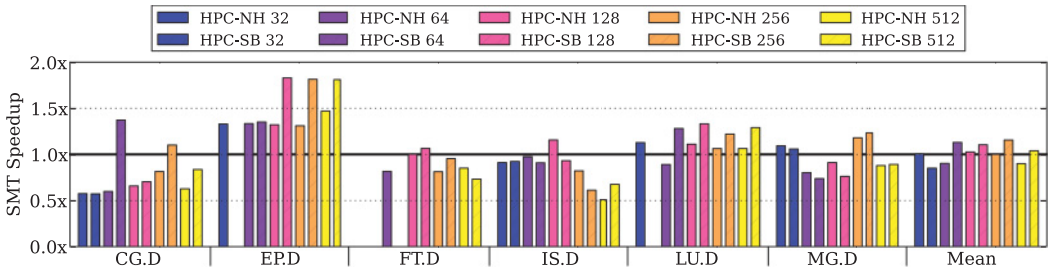


Fig. 4. Speedup from SMT for the MPI NPBs at varying physical core counts on Sandy Bridge (HPC-SB) and Nehalem (HPC-NH) systems.

Table III. Physical Core versus Virtual Core Scaling for Application MG (CLASS D)

	Change in Number of Threads				
Scaling Mode	32 → 64	64 → 128	128 → 256	256 → 512	512 → 1024
Physical Cores (NoSMT)	2.60	1.74	1.91	2.06	0.99
Virtual Cores (SMT)	1.06	0.74	0.76	1.23	0.89

was used, which results in the per core memory footprint decreasing as the core count increases. For the large production applications in Figure 3, we find a consistent trend: SMT is most valuable with smaller core counts and performance degrades with larger core counts. In addition, whether SMT is beneficial for each application remains generally consistent on the two systems.

Figure 4 presents the performance analysis of the MPI NPBs at large core counts. Unlike the large applications where consistent trends were generally recognizable, the MPI NPBs often exhibit inconsistent behavior. Although some benchmarks, such as EP and LU, are largely consistent in benefiting from SMT, the remaining benchmarks—CG, FT, IS, and MG—are inconsistent in this respect. Since only certain core counts benefit and others do not, the choice between SMT^{idle} and SMT^{used} cannot be thought of simply in terms of the application as whole. CG, in particular, changes significantly depending on core count, almost certainly because grid-layouts are more favorable at certain core counts than others. In addition, CG benefits from using SMT for 40% of the core counts used on Sandy Bridge systems, yet never benefits on Nehalem systems. These inconsistencies between different core counts and different systems for a single application highlight how difficult it may be for users to effectively utilize SMT.

Moreover, the performance benefit from SMT is not closely tied to application scalability in general. Among the NPBs, EP is the only application where EP's scaling (using only physical cores, no SMT) and the benefit from using SMT (doubling threads, same number of cores) are closely connected. For the other applications, the connection is much weaker. For example, Table III shows the weak connection between scaling using physical cores (no SMT) and scaling using virtual cores (SMT) for MG. As such, even if users are well versed in the scalability of their application, it may have little relevance for the speedup from using SMT to increase their thread count.

4.2.3. Summary. The performance benefit of using SMT varies by application, core count, and system configuration. To illustrate the difficulty of deciding whether to use SMT, Figure 5 presents a breakdown of the speedup when using SMT by application group. These results include 100 different tests of whether SMT is valuable for a given application, system, and size. Exactly 50 of those benchmarks benefited, whereas 50 did not. Thus, a blanket strategy such as "always use SMT" or "never use SMT" would result in a performance degradation exactly half the time.

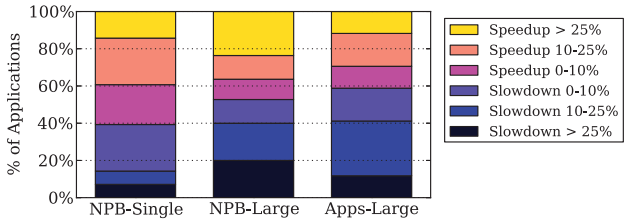
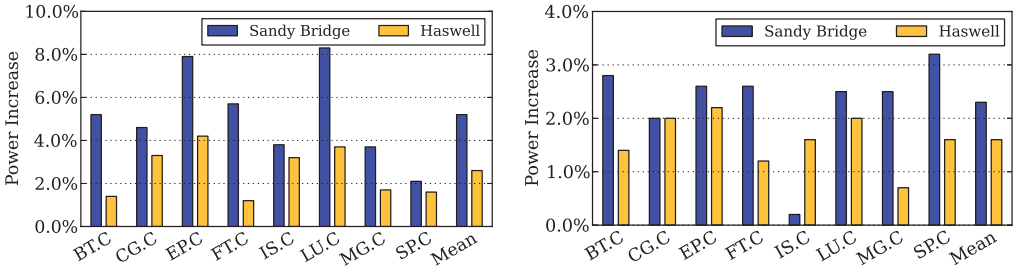


Fig. 5. For each application group, the number of applications that experienced various levels of SMT speedup (>25% improvement, 10%–25% improvement, etc.).



(a) Cost of Employing SMT

(b) Cost of Enabling SMT

Fig. 6. Average power increase associated with SMT when running NPBs (OpenMP and MPI) on Sandy Bridge (RCK-SB and SRV-SB) and Haswell (SRV-HW) systems. (a) provides the power increase when employing SMT contexts (32 threads vs. 16 threads). (b) Provides the power increase of enabling SMT contexts in the BIOS without using the additional SMT contexts (16 threads vs. 16 threads).

4.3. The Power Cost of SMT

This section evaluates the cost of *using* SMT and then the cost of merely *enabling* SMT.

4.3.1. Impact of Using SMT. Figure 6(a) provides the power increase when using SMT (SMT^{used} in Table I) for the NPBs. This increase in power is experienced regardless of whether the application realized a performance benefit, and it occurs on both Sandy Bridge and Haswell systems when using SMT. On average, we observe a 5.8% power increase on the Sandy Bridge systems and a 3.2% increase on the Haswell system.

4.3.2. Impact of Enabling SMT. Figure 6(b) provides the power consumption increase from running the NPBs with SMT^{idle} versus SMT^{off} for both Sandy Bridge and Haswell-based systems. On average, the Sandy Bridge system **SRV-SB** consumes 2.3% more power with SMT^{idle}. The Haswell system **SRV-HW** consumes slightly less additional power, 1.7% on average, but this may be a byproduct of the fact that **SRV-SB** has 4× more cores than **SRV-HW**.

Although there is a notable power difference when running the same job with SMT^{idle} and SMT^{off}, we notice much smaller performance impacts. In the tests presented in Figure 6(b), most benchmarks showed runtime differences of less than 0.2%. Those benchmarks with larger differences were the OpenMP version of MG where the runtime increased by 2.4% on **SRV-SB** and the OpenMP version of SP where the runtime improved by 0.9% on **SRV-SB**.

Last, we observe that on a vacant system (i.e., with nothing but a minimal set of OS services running), SMT^{idle} may slightly increase the power consumption above that of SMT^{off}. For **SRV-SB,** the average increase is 0.5% (0.6W), and for **RCK-SB,** the increase is 0.6% (0.5W). For **SRV-HW**, we found no measurable change.

Table IV. Average Increase in Hardware Monitor Events on RCK-SB When Enabling SMT in the BIOS and Running OpenMP NPBs with the Same Number of Threads and Same Core Pinning

Performance Monitor	SMT^{idle} Increase
L1 I-Cache Misses	11.8%
L2 I-Cache Hits	14.6%
Cycles w/o Insn. Issue	24.2%
Cycles w/o Insn. Issue (excl. IS)	5.2%
Data TLB Misses	6.6%
Insn. TLB Misses	14.2%

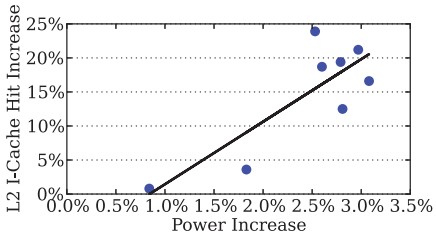


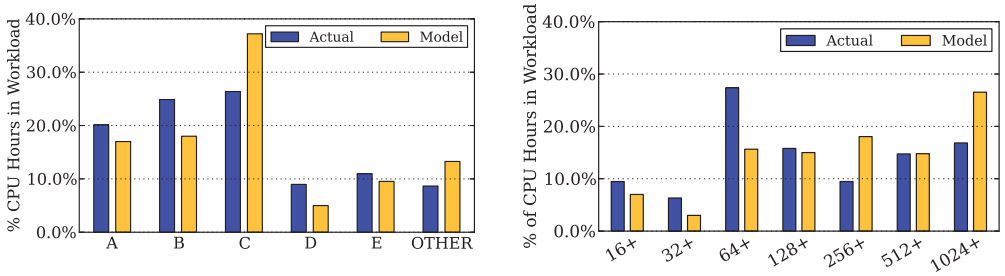
Fig. 7. The relationship between power consumption and additional L2 I-Cache Hits while SMT is enabled but unused. Correlation is 0.82.

Power Increase Examination: Discovering that merely enabling SMT impacts power, and to a lesser extent performance, caused us to examine the phenomenon more closely. On **RCK-SB**, we collected statistics from 43 hardware performance counters during the runs of each of the NPBs run with both SMT^{idle} and SMT^{off}. As we anticipated, most performance counters (38/43) saw little change between the two states. However, five of them showed significant amounts of variation. The counters that were found to differ by more than 5% are shown in Table IV. In this table, it is worth noting that the high increase in the number of Cycles without Instruction Issue is dominated by IS. With SMT^{off}, IS had a small number of these cycles in absolute terms, and a small absolute increase resulted in a relative increase of 157%. Excluding IS from this average (also shown in Table IV) lowers this average to 5%.

Examining another counter, L2 instruction cache (I-Cache) hits, in more depth indicates that the relationship between L2 I-Cache hits and power increase are highly correlated, exhibiting a Spearman correlation of 0.82 (see Figure 7). This strong correlation may suggest that some form of resource partitioning is being performed by the processor when SMT is enabled, perhaps I-Cache and TLB partitioning between SMT contexts (virtual cores) is occurring in some cases regardless of whether both SMT contexts are in use by the application. This could be a performance optimization, as partitioning of out-of-order resources between SMT contexts can be effective at ensuring one thread does not starve another [Raasch and Reinhardt 2003]; indeed, avoiding such starvation can have a major impact on performance [Tullsen and Brown 2001].

Another possible explanation for this effect is the ability for OS threads to run on idle SMT contexts. To examine this issue, we first used the OS (not BIOS) to disable idle thread contexts for SMT^{idle} and found a small decrease in power relative to SMT^{idle} when the OS could use idle cores. The OS occupying the idle cores may explain this fraction of power increase, as might any interference between the application and OS threads. However, we also found that using the OS to disable idle thread contexts from SMT^{idle} still consumed more power than SMT^{off} (approximately half of the power overhead of SMT remained for single-node NPBs). As these two configurations are effectively identical from an OS perspective, this suggests that part of the increased power consumption may stem from system differences rather than OS thread scheduling.

These results only provide limited insight into the underlying cause of the power increase. As such, more investigation is required to provide precise explanations for the behavior. Finally, it is important to underscore that these results do not represent a fundamental limitation of SMT, nor is this necessarily a criticism of the Hyperthreading implementation. Although SMT has become nearly ubiquitous, it may be the case that the operating system, BIOS, and motherboard interfaces require a static number of cores to be deployed at startup. As a result, changes to the hardware and software



(a) Workload Domain (b) Workload Size

Fig. 8. Comparison of computational domain and physical CPU counts in actual versus modeled workload.

stack that allow dynamic enabling of SMT may ultimately be necessary to improve deployed SMT implementations.

4.4. Workload Evaluation

In this section, we deploy our methodology on the Department of Defense HPCMP 2011 workload in order to quantify the benefit of enabling SMT on a large-scale HPC system. First, we devise a representative workload model (Section 3.1.4).

4.4.1. HPCMP 2011 Workload. We use statistics collected from a number of Department of Defense HPCMP systems to obtain summary statistics on scientific computational area profiles and physical core counts of that workload. These profiles are shown in Figures 8(a) and 8(b), respectively (dark blue bars).

Using the hill-climbing approach described in Section 3.1.4 along with the set of job types for our applications in Table II, we assign weights to each of the jobs to create a workload model. The workload model covers five different computational domains[3] that account for over 90% of the CPU-hours in HPCMP's 2011 production workload. Our modeled workload (light yellow bars in Figures 8(a) and 8(b)) shows a great deal of similarity to the actual workload. For computational area, the modeled workload differs by an average of 5.1% of the CPU-hours across the six areas (including OTHER) in Figure 8(a), and differing by an average of 5.2% across the seven CPU bins in Figure 8(b).

4.4.2. Putting it Together: Full-System SMT Impact. We can now augment the workload model for the HPCMP 2011 production workload developed in Section 4.4 with measurements of the performance factors related to SMT for the jobs in our workload model (Section 4.2) and the power overheads for enabling and using SMT on the system (Section 4.3). We bring all these factors together to quantify the benefits of enabling SMT for this workload running on a Sandy Bridge system.

As a simplification, we use exactly three power costs within our methodology. Because the power increase on Sandy Bridge when using SMT was relatively stable around 5% (Figure 6(a)), we use a fixed power cost $C_j^{used} = 105\%$ for all jobs j. Similarly, enabling SMT but not using SMT contexts was found to increase power consumption by at least 1.5% and enabling SMT increases idle power consumption by very close to 0.5%. Therefore, we use $C_j^{idle} = 101.5\%$ and $C_{vacant}^{idle} = 100.5\%$ for all jobs j.

Taking these factors together and computing Equations (7) and (8) results in Figure 9. In Figure 9, we present the overall performance and energy impacts of enabling SMT for this workload as a function of how likely users are to make the ideal decision

[3]For confidentiality, computational areas are anonymized and application weighting cannot be provided.

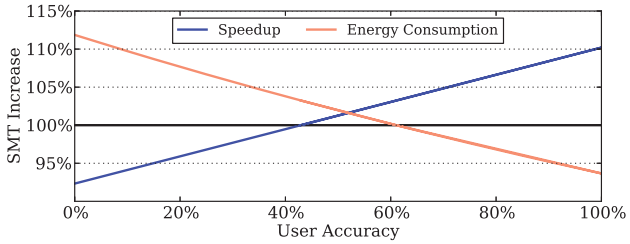


Fig. 9. Impact of enabling SMT system-wide on runtime performance (higher is better) and energy (lower is better) for the HPCMP based on user accuracy. This assumes the workload is at 100% utilization, a 5% overhead from using SMT, a 1.5% overhead for enabling SMT but not using SMT contexts, and a 0.5% overhead when idle.

Table V. Average Absolute Error of SMT Speedup Predictions **SRV-SB** by Model

Linear	Cubist	Linear w/PCA	Cubist w/PCA
13.1%	6.1%	12.0%	6.8%

in terms of using or leaving idle the SMT contexts available to them when SMT is enabled. The figure demonstrates that as user accuracy increases, SMT's benefit to energy and performance also increases. In this figure, user accuracy for static policies of "always use SMT" or "never use SMT" would produce a user accuracy of 49% and 51%, respectively. To improve performance, on average, when employing SMT, users must correctly decide to enable or disable it at least 43% of the time. However, because of the power costs involved with SMT, users must make the correct decision at least 61% of the time to break even for energy consumption. As user correctness is clearly at the heart of determining SMT's system-wide impact, the following section addresses means to improve users' ability to predict SMT's value.

5. PREDICTING APPLICATION-LEVEL BENEFIT

In this section, we evaluate statistical models for predicting the speedup of employing SMT for an application. In each case, the prediction model aims to provide the user, when running a job without using SMT, with a predicted speedup for running that job with SMT *on the same number of physical cores*.

5.1. Single-Node Sandy Bridge Predictions

Both multiple linear regression and cubist models were built using the methodology described in Section 3.2. For all models, 40% of applications are randomly selected as the training set for building the model, and the remaining 60% are reserved for the test set. For Sandy Bridge, a total of 585 applications and microkernels using MPI and OpenMP were included. The average actual speedup from SMT for these 586 applications was 0.88 with a standard deviation of 0.16. Of the applications, 115 (20%) benefited from using SMT on the single-node.

Four models were created for **SRV-SB**. The first two ("Linear" and "Cubist") used all performance monitors available on the system to aid in the predictions, and the latter two ("Linear w/PCA" and "Cubist w/PCA") only used nine monitors based on performing explanatory variable reduction (PCA and Clustering) described in Section 3.2.2. For these models, Table V provides the average absolute error and Figure 10 provides a histogram of the errors, where the error is the predicted speedup subtracted from the actual speedup. Note that for Figure 10, one datapoint was excluded for each of the linear models as the prediction's error was abnormally large. The prediction would

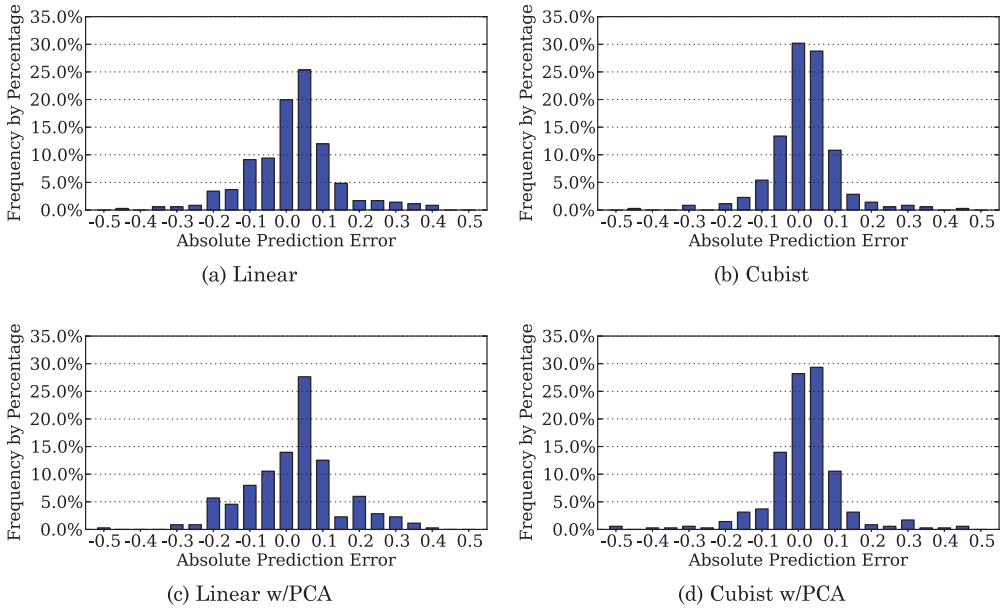


Fig. 10. Error histograms comparing predicted versus actual SMT speedup for various modeling techniques on **SRV-SB**.

clearly be mistaken due to its magnitude of speedup and the system could omit such predictions from reaching the user. The Cubist model had no such artifacts.

Examining the four models for **SRV-SB**, there are two conclusions. The first is that performing variable reduction using PCA did not have a large negative impact on the models and actually improved the prediction accuracy for the Linear model. This is a positive result as only a small number of monitors are required to make a prediction, eliminating the need to run an application multiple times to gain the metrics needed for a prediction. The second is that the Cubist model predicts better than the Linear model for this system. For the Cubist PCA model, 72% of the test set has an error in predicted speedup of less than 5%.

5.2. Single-Node Nehalem Predictions

Performing the same analysis on Nehalem as in the previous section, we find results that are broadly compatible. On the Nehalem system (**HPC-NH2**), 100 of the 588 total applications benefited from SMT. The average SMT speedup for the group is 0.86 with a standard deviation of 0.16.

Building four models using the same technique as in the previous section, we find that variable reduction (PCA) performed as well as models built using all performance monitors, and hence we limit the remaining discussion to the PCA-based models. Cubist PCA is again more effective with a smaller absolute error than Linear PCA (7.8% and 9.8%, respectively). Figure 11 provides histograms grouping predictions by their error (again, predicted speedup subtracted from actual speedup). The Cubist PCA model is capable of predicting the majority of the speedups (69%) within 5% of actual.

5.3. Model-based Insights: Which Architectural Features Matter Most?

We next explore the variable importance of various monitors in the Linear and Cubist w/PCA models produced in the previous sections. The variable importance for these models can be found in Figure 12. Intuitively, the importance of an explanatory variable

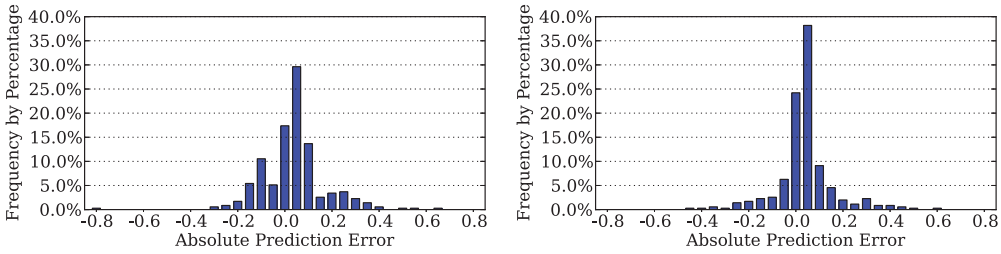


Fig. 11. Error histograms comparing predicted versus actual SMT speedup for our modeling techniques on **HPC-NH**.

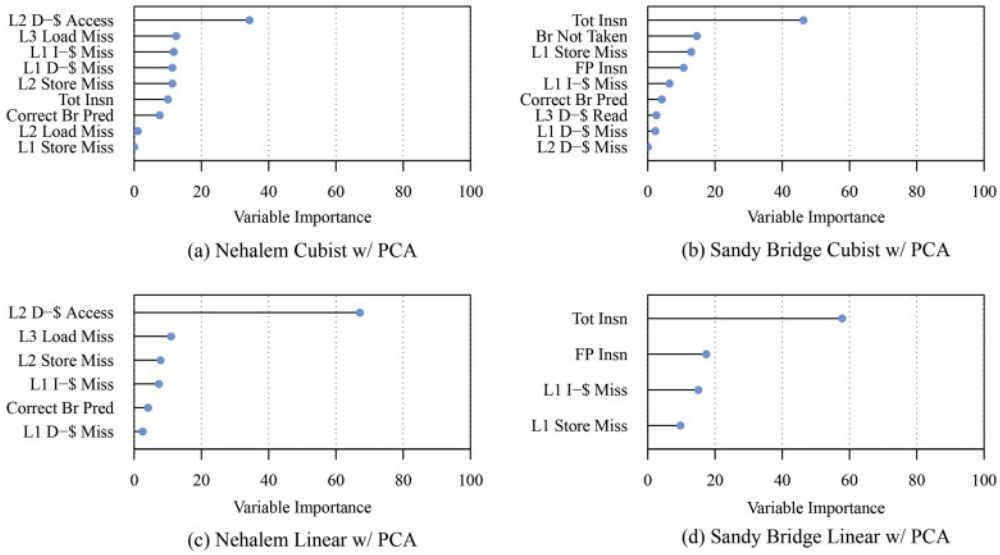


Fig. 12. Most influential predictors of Linear and Cubist PCA models for **HPC-NH** and **SRV-SB**. All monitors are normalized by total execution time, thus Total Instructions reflects total instructions per second.

is its impact in terms of whether it makes a large difference to the response variable. Values in this figure represent what fraction of the model's accuracy, out of 100, can be explained by that particular monitor.

We begin by inspecting the Cubist models, as they provided the better prediction accuracy, and then look to the Linear model to see if similar trends are present. The monitors that appear in the variable importance analysis for the Cubist model (Figures 12(a) and 12(b)) correspond well with intuition about when SMT would be beneficial. We would expect SMT to be beneficial when there are idle resources available on each core. These exploitable idle resources appear in the monitors in the following ways:

(1) **Instruction Throughput**: Both systems value total instructions committed (per second). As instruction commit bandwidth is limited and if instruction throughput is high, there would be no room in the core for an additional thread. Conversely, if few instructions are being committed per second, this might imply that commit bandwidth (and other resources) are available. The Sandy Bridge system relies

heavily on this variable while it is valued less in Nehalem. Similarly, floating point units are limited and, particularly in high-performance computing applications, these units can be heavily stressed. Few floating point instructions being committed per second implies availability of floating point units for additional threads. Branches not taken (per second) and data cache accesses per second may simply reflect another form of instruction throughput but may also reflect utilization of other microarchitectural resources (branch predictor, caches). Branches predicted correctly (per second) could not only also just reflect overall instruction throughput, but may also reflect that the branch predictor is performing well and hence may be able to support increased utilization.

(2) **Memory Bandwidth**: Both systems prioritize a number of metrics related to memory operations per second. L1 data cache accesses may indicate pressure on the L1 cache, and it is a critical variable for Nehalem. L1 instruction cache misses, L1 data cache misses, and L1 store misses all reflect pressure on the L2 cache, as well as how heavily the bandwidth is used between L1 and L2. L2 pressure is also reflected by L2 reads and L2 misses. L3 pressure is represented by L2 misses, L3 data reads, and L3 load misses. L3 load misses are a top variable for Nehalem and also reflect utilization of bandwidth to memory. In all of these cases, the hardware monitors selected by PCA correspond to likely predictors of memory pressure. In the absence of that pressure, SMT may be effective.

Although both Nehalem and Sandy Bridge have many of the same input variables among their top variables for prediction, it appears that predicting the success of SMT on Sandy Bridge is more heavily based on instruction throughput first and cache pressure second (although both are certainly interconnected). These are reversed for Nehalem, which places L1 cache pressure and off-chip bandwidth as factors more critical than instruction throughput. These trends are present in the Linear model as well (Figures 12(c) and 12(d)), providing support that these trends are likely indicative of behavior of the architectures themselves, rather than being artifacts of the Cubist models.

These results correspond to intuition based on the improvements made to Sandy Bridge over Nehalem. Sandy Bridge adds additional cache support for instructions (a μops cache), nearly doubles read bandwidth, improves memory bandwidth by 20%, and switches to a ring-bus interconnect between L2 and L3 rather than the direct L3 connections in Nehalem [Saini et al. 2013]. These improvements result in better intranode communications, internode communications, and most critically, better memory performance. Hence, we find memory performance to be a more significant predictor of SMT's value in Nehalem than Sandy Bridge due to Sandy Bridge's improvements to memory likely moving memory performance off of the critical path.

Another potentially key component, not reflected in these monitors, to the possible success of using SMT (splitting into twice as many threads/processes) is available cache space based on working-set size. For example, hardware monitors can show when the L2 cache is struggling by having high L2 data misses. However, if there are few L2 data misses, this could be because the working-set is small or because it just barely fits in L2. If another thread, which needs a large piece of L2, were to be co-scheduled, then these threads may or may not conflict depending on the working-set size of the co-scheduled thread. What further confounds predicting the value of SMT with regard to cache space is that the working-set might change when the problem is split into more threads or processes. Given the inability to account directly for working-set size using hardware monitors combined with an inability to detect the change in memory behavior when dividing the task into more processes or threads, the success of these prediction models is all the more interesting.

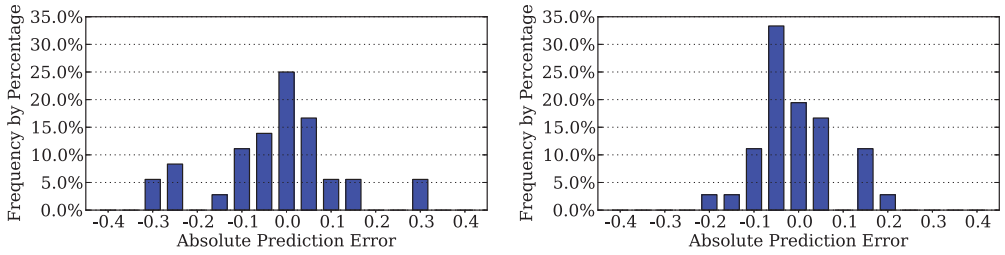


Fig. 13. Error histograms comparing predicted versus actual SMT speedup for our modeling techniques of multinode runs of large applications on **HPC-SB**.

5.4. Multinode Predictions of Large Applications

Examining the effectiveness of the prediction models for large MPI applications on multiple nodes posed a few challenges. The first is that our previously examined hardware monitors do not capture communication patterns. To address this, we also collect communications summary statistics using a software wrapper library [Tikir et al. 2009] for the message passing interface (MPI). We note that this library collects counts and timings of the message passing activity while introducing negligible overhead.

The second challenge is that our microkernels did not scale beyond one node, leaving our model training and evaluation set to be a mix of different sizes of the class D NPBs and different sizes of large applications from the NERSC and HPCMP Production workload. With a relatively smaller total of applications (36), the models could not be built using the 40%/60% train/test set as done in prior sections. Moreover, with almost 90 input variables (45 hardware monitors and 44 MPI-related inputs), overfitting is a significant concern. To address these issues, the MPI-related inputs were reduced into 6 classes—peer-to-peer, collective, init/fini, communicator manipulation, barrier, and wait—allowing us to reduce the 44 input parameters into 6. Those 6 inputs were used to build a model for communication time only (producing a communication speedup). We also performed our standard variable reduction using PCA to use only 9 of the 45 hardware monitors. These 9 inputs were used to model for computation time only (producing a computation speedup). Finally, we combine the communication and computation predictions by simply adding up the predicted time of each.

For our 36 applications, the range of actual speedups for SMT was much wider than for our single-node applications. The average SMT speedup was 1.05, the standard deviation was 0.26, and 22 of the 36 applications benefited from SMT. We find that our models, both linear and cubist, are quite accurate for these applications. Our average absolute error for the linear and cubist models are 10.5% and 7.5%. The error histograms for these models appear in Figure 13, again showing the Cubist model to be better than Linear, although both provide reasonable accuracy.

5.5. SMT Speedup as a Classification Problem

In the prior sections, we focused on absolute prediction accuracy for our models. Another useful view of the models is to determine whether they correctly predict the simple classification problem of whether or not SMT speeds up an application. Consider two scenarios:

(1) The prediction is for a speedup of 1.10 and the actual speedup is 1.25. The error is relatively innocuous, and the user may even be pleasantly surprised.
(2) The prediction is for a speedup of 1.10, but the actual speedup is 0.95. The error in the model leads the user to the wrong decision.

Table VI. Accuracy of PCA Model Predictions to Use, or Not to Use, SMT for an Application

One-Node Sandy Bridge		One-Node Nehalem		Multinode Sandy Bridge	
Linear	Cubist	Linear	Cubist	Linear	Cubist
77.2%	82.6%	81.6%	86.4%	75%	83%

Table VII. Average Cubist PCA Model Predictions and Average Actual Speedups for Multinode Sandy Bridge, Grouped by Prediction and Outcome

	Correct Prediction		Incorrect Prediction	
Prediction	Avoid SMT	Use SMT	Avoid SMT	Use SMT
Actual Result	Avoid SMT	Use SMT	Use SMT	Avoid SMT
Apps in Group	12	18	4	2
Avg. Pred. Speedup	0.87	1.29	0.90	1.01
Avg. Actual Speedup	0.86	1.25	1.04	0.89

Despite having the same absolute prediction error, these two scenarios have quite different outcomes. Moreover, as our system-wide decision to enable SMT is based on users correctly making this very decision to use SMT, we are most concerned with how often the models correctly predict to use, or not to use, SMT.

Table VI provides the accuracy of each model in predicting whether to employ SMT. All models predict whether to use SMT or not with at least 75% accuracy, and Cubist models are at least 82% accurate. However, we point out that these raw numbers may be misleading. For example, if the model predicts a speedup of 0.98 and the actual speedup is 1.01, this is treated as an error even though the performance loss is minor. To explore this facet more closely, we examined the multinode Sandy Bridge results and grouped the average predicted speedup and average actual speedup for the four possible combined outcomes (predict use/avoid SMT versus the actual correct choice to use/avoid SMT). These results appear in Table VII.

Table VII shows that the model is more accurately predicting applications with a large SMT performance benefit (1.25) and large SMT performance loss (0.86) than those with less benefit (1.04) and (slightly) less loss (0.89). Although we do not factor in this difference (which applications are predicted better/worse) into the next step of our analysis, it implies our estimates of the impact these models may have on production workloads may be pessimistic. Finally, this also implies that adding a confidence measure to the predictions could be a useful direction for future work.

Overall, this section has shown two critical pieces for our proposed framework. The first is that models of SMT performance for HPC applications, on different systems and for different sizes of applications, can be used to accurately predict the performance impact from using SMT for a specific application. The second is that only a small set of performance monitors are required to make this prediction; facilitating the collection of these monitors in the background while running users' jobs and then informing them of the prediction after the run completes.

5.6. Combining Workload Analysis with Predictions

At the end of Section 4.4, the result from the workload analysis on DoD production workload running on DoD HPC system was that users need to accurately employ SMT more than 61% of the time to break even for the power costs/benefits associated with enabling SMT. To aid user decisions, we built a number of performance models to predict the benefit of SMT on multiple systems and multiple groups of applications and all models exceed this 61% threshold. We again note that results may vary by system

and by workload, so this analysis would need to be performed for each particular system and workload. Although the result for our DoD system and workload favored enabling SMT system-wide, other workloads and systems may encounter different results.

The multinode Sandy Bridge Cubist model (built on our production HPC system) was 83% accurate on predicting the value of SMT. Referring back to Figure 9, we can infer the impact of this level of user accuracy on our production system. Assuming this prediction model were employed on the production system and used by HPC users, average performance could improve by 7% and energy consumption could lower by 4%. The raw energy consumption at this level of user accuracy is 97% that of ideal (were users 100% correct).

6. FUTURE WORK

Modeling Phases: This work has shown that a model for predicting SMT's benefit can be built based on a given HPC workload. Should the workload change, at what point should a new prediction engine be created? We anticipate that future work on this topic will use either changes in the types of applications (should application computational domain be known), the sizes of applications, and/or occasionally sampling predictions to see if the model remains applicable.

Architectural SMT Features: In this work, we have modeled and reasoned about existing SMT hardware implementations. One question unaddressed in this work is what changes could be made to SMTs implementation to assist users and system designers in reasoning about its value. We suspect answering this problem requires addressing tradeoffs between SMT's versatility and consistency. For example, statically partitioning resources might make it easier to reason about SMTs behavior, but in turn, might limit its ability to adapt to particular workloads.

Hybrid Systems: An alternative to making a single SMT decision for an entire computing center would be to partition some nodes as SMT enabled and others as SMT disabled. By having some nodes with SMT disabled, power costs associated with running without using SMT could be avoided by scheduling those jobs on those systems. Unfortunately, such a hybrid system would prevent jobs wishing to use SMT on more than the available SMT-enabled nodes from being able to do so. Workload characterization would be necessary to determine what percentage of jobs would benefit by having such a system. A more flexible solution may be to support dynamically enabling/disabling of SMT through architectural and/or OS modifications.

7. CONCLUSION

The potential of SMT to improve processor utilization by allowing threads to compete for idle resources has made SMT a popular feature in high-end processors, resulting in its presence in the majority of the world's fastest supercomputers. However, SMT results in an increase in power draw whether it confers extra performance to running jobs. As this work has shown, SMT may also result in increased power consumption when it is merely *enabled* but not used. The variability in the benefit of SMT and the additional power costs makes it difficult to assess the tradeoffs for enabling SMT on an HPC system. This work has presented a full system-wide methodology for quantifying the performance and power benefits of SMT and applied the methodology to a production workload on a production system. Ultimately, the value, system-wide, for SMT depends on how well users employ SMT for their applications.

However, predicting the benefit of SMT to an HPC application that has exclusive access to the processor is challenging, as the act of doubling the number of threads changes the characteristics of the application. Unlike the well-studied problem of predicting how two distinct threads will interact on an SMT context, for HPC

applications, this problem involves predicting the impact of doubling the number of threads for the application that changes both the characteristics of the application and their interaction on SMT contexts. This article has presented an approach to predicting the benefit SMT confers to HPC applications, using statistical modeling techniques based on lightweight performance monitors. These statistical models are able to predict the speedup from SMT to within 8% accuracy while relying only on monitors available, transparently, from a single run of the application. Our system-level workload analysis suggests that these models offer overall performance and energy within 97% of an oracle predictor.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful suggestions and contributions to this work.

REFERENCES

Emile H. L. Aarts and Jan K. Lenstra. 1997. *Local Search in Combinatorial Optimization*. Princeton University Press.

Katie Antypas, John Shalf, and Harvey Wasserman. 2008. *NERSC-6 Workload Analysis and Benchmark Selection Process.* Technical Report, Lawrence Berkeley National Laboratory.

David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, et al. 1991. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 5th Conference on Supercomputing*.

Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd International Conference on Supercomputing*.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The Parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

Gary Bradski. 2000. The OpenCV library. *Doctor Dobbs Journal* 25, 11 (2000).

Laura Carrington, Michael Laurenzano, Allan Snavely, Roy L. Campbell, and Larry P. Davis. 2005. How well can simple metrics represent the performance of HPC applications? In *Proceedings of the 19th International Conference on Supercomputing*.

Laura Carrington, Michael Laurenzano, and Ananta Tiwari. 2013. Characterizing large-scale hpc applications through trace extrapolation. *Parallel Processing Letters* 23, 4 (2013).

Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. 2004. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture*.

Onur Celebioglu, Amina Saify, Tau Leng, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. 2004. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*.

Eric P. Chassignet, Harley E. Hurlburt, Ole Martin Smedstad, George R. Halliwell, Patrick J. Hogan, Alan J. Wallcraft, Remy Baraille, and Rainer Bleck. 2007. The HYCOM (hybrid coordinate ocean model) data assimilative system. *Journal of Marine Systems* 65, 1 (2007).

Seungryul Choi and Donald Yeung. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd International Symposium on Computer Architecture*.

M. J. Cordery, B. Austin, H. J. Wassermann, C. S. Daley, N. J. Wright, S. D. Hammond, and D. Doerfler. 2013. Analysis of Cray XC30 Performance Using Trinity-NERSC-8 Benchmarks and Comparison with Cray XE6 and IBM BG/Q. In *Proceedings of the 4th International Workshop on Performance Modeling, Benchmarking and Simulation.*

Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th International Conference on Supercomputing*.

Matthew Curtis-Maury, Tanping Wang, Christos Antonopoulos, and Dimitrios Nikolopoulos. 2005. Integrating multiple forms of multithreaded execution on multi-SMT systems: A study with scientific applications. In *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems*.

Larry P. Davis, Cray J. Henry, Roy L. Campbell, and William A. Ward. 2007. High-Performance computing acquisitions based on the factors that matter. *Computing in Science & Engineering* 9, 6 (2007).

Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen. 2006. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*.

Chris Ding and Xiaofeng He. 2004. K-means clustering via principal component analysis. In *Proceedings of the 21st International Conference on Machine Learning*.

Norman Richard Draper and Harry Smith. 1981. *Applied Regression Analysis* (2nd ed.). John Wiley and Sons.

Peter E. Duda and Richard O. Hart. 1973. *Pattern Classification and Scene Analysis*. John Wiley and Sons.

Stijn Eyerman and Lieven Eeckhout. 2009. Per-thread cycle accounting in smt processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Stijn Eyerman and Lieven Eeckhout. 2012. Probabilistic modeling for job symbiosis scheduling on smt processors. *ACM Transactions on Architecture and Code Optimization* 9, 2 (June 2012).

Stijn Eyerman and Lieven Eeckhout. 2014. The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Paweł Gepner, David L. Fraser, Michał F. Kowalik, and Kazimierz Waćkowski. 2011. Evaluating new architectural features of the Intel® Xeon® 7500 processor for HPC workloads. *Computer Science* 12 (2011).

Ryan E. Grant and Ahmad Afsahi. 2005. Characterization of multithreaded scientific workloads on simultaneous multithreading intel processors. In *Proceedings of the Workshop on Interaction between Operating System and Computer Architecture*.

Alan Gray, J. Hein, M. Plummer, A. Sunderland, L. Smith, A. Simpson, and A. Trew. 2006. *An Investigation of Simultaneous Multithreading on HPCx*. Technical Report 0604, EPCC–University of Edinburgh.

Charles Hoke, Victor Burnley, and Gregory Schwabacher. 2004. Aerodynamic analysis of complex missile configurations using AVUS (air vehicles unstructured solver). In *Proceedings of the 22nd Applied Aerodynamics Conference and Exhibit*.

Ian Jolliffe. 2005. *Principal Component Analysis*. Wiley Online Library.

Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. 2010. Power7: IBM's next-generation server processor. *IEEE Micro* 30, 2 (2010).

Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Mike Gittings. 2001. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 15th International Conference on Supercomputing*.

Darren J. Kerbyson and Philip W. Jones. 2005. A performance model of the parallel ocean program. *International Journal of High Performance Computing Applications* 19, 3 (2005).

David Koufaty and Deborah T. Marr. 2003. Hyperthreading technology in the NetBurst microarchitecture. *IEEE Micro* 23, 2 (2003).

Michael A. Laurenzano, Mitesh Meswani, Laura Carrington, Allan Snavely, Mustafa M. Tikir, and Stephen Poole. 2011. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In *Euro-Par 2011 Parallel Processing*.

Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. 2007. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th International Symposium on Principles and Practice of Parallel Programming*.

Peter Mardahl, Andrew Greenwood, Tony Murphy, and Keith Cartwright. 2003. Parallel performance characteristics of ICEPIC. In *Proceedings of the User Group Conference*.

Gabriel Marin and John Mellor-Crummey. 2004. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 32.

Harry M. Mathis, Alex E. Mericas, John D. McCalpin, Richard J. Eickemeyer, and Steven R. Kunkel. 2005. Characterization of simultaneous multithreading (SMT) efficiency in power5. *IBM Journal of Research and Development* 49, 4.5 (2005).

Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. 2014. The Top 500 List. Retrieved from http://www.top500.org.

Kent F. Milfeld, Chona S. Guiang, Avijit Purkayastha, and John R. Boisseau. 2003. Exploring the effects of hyper-threading on scientific applications. *Cray User Group 2003* 112 (2003).

Tipp Moseley, Joshua L. Kihm, Daniel A. Connors, and Dirk Grunwald. 2005. Methods for modeling resource contention on simultaneous multithreading processors. In *Proceedings of the 2005 International Conference on Computer Design: VLSI in Computers and Processors*.

Steve Plimpton, Paul Crozier, and Aidan Thompson. 2007. LAMMPS-large-scale Atomic/Molecular Massively Parallel Simulator. Sandia National Laboratories.

Louis-Noël Pouchet. 2012. Polybench: The Polyhedral Benchmark Suite. Retrieved from http://www.cs.ucla.edu/~pouchet/software/polybench/.

Steven E. Raasch and Steven K. Reinhardt. 2003. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*.

Ryan Rakvic, Qiong Cai, José González, Grigorios Magklis, Pedro Chaparro, and Antonio González. 2010. Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors. *ACM Transactions on Architecture and Code Optimization* 7, 2 (Oct. 2010).

RuleQuest Research. 2012. Data Mining with Cubist. Retrieved from http://rulequest.com/cubist-info.html.

Subhash Saini, Johnny Chang, and Haoqiang Jin. 2013. *Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications*. White paper, NASA Ames Research Center. (2013).

Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra, and Rupak Biswas. 2011. The impact of hyper-threading on processor resource utilization in production applications. In *Proceedings of the 18th International Conference on High Performance Computing*.

Robert Schöne, Daniel Hackenberg, and Daniel Molka. 2011. Simultaneous multithreading on x86_64 systems: An energy efficiency evaluation. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*.

Ronak Singhal and Senior Principal Engineer. 2008. Inside Intel core microarchitecture (Nehalem). In *A Symposium on High Performance Chips*, Vol. 20.

Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. 2002a. A framework for performance modeling and prediction. In *Proceedings of the 16th International Conference on Supercomputing*.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Allan Snavely, Dean M. Tullsen, and Geoff Voelker. 2002b. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

ThinkTank Energy Products Inc. 2014. Watts up? Product. Retrieved from http://www.wattsupmeters.com.

Xinmin Tian, Milind Girkar, Sanjiv Shah, Douglas Armstrong, Ernesto Su, and Paul Petersen. 2003. Compiler and runtime support for running OpenMP programs on Pentium-and Itanium-architectures. In *Proceedings of the International Symposium on Parallel and Distributed Processing*.

Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. 2009. PSINS: An open source event tracer and execution simulator for MPI applications. In *Euro-Par 2009 Parallel Processing*.

Nathan Tuck and Dean M. Tullsen. 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*.

Dean M. Tullsen and Jeffery A. Brown. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture*.

Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*.

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, Vol. 23.

Augusto Vega, Alper Buyuktosunoglu, and Pradip Bose. 2013. SMT-centric power-aware thread placement in chip multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*.

Huaping Wang, Israel Koren, and C. Mani Krishna. 2008. An adaptive resource partitioning algorithm for SMT processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

Received June 2014; revised October 2014; accepted October 2014