Identifying Energy-Efficient Concurrency Levels Using Machine Learning

Matthew Curtis-Maury [1,3], Karan Singh [2,3], Sally A. McKee [2], Filip Blagojevic [1]
Dimitrios S. Nikolopoulos [1], Bronis R. de Supinski [3], Martin Schulz [3]

[1] *Department of Computer Science*
Virginia Tech
Blacksburg, VA 24061
{mfcurt,filip,dsn}@cs.vt.edu

[2] *Computer Systems Lab*
Cornell University
Ithaca, NY 14853
{karan,sam}@csl.cornell.edu

[3] *Lawrence Livermore National Laboratory*
Livermore, CA 94551
{bronis,schulzm}@llnl.gov

Abstract— **Multicore microprocessors have been largely motivated by the diminishing returns in performance and the increased power consumption of single-threaded ILP microprocessors. With the industry already shifting from multicore to many-core microprocessors, software developers must extract more thread-level parallelism from applications. Unfortunately, low power-efficiency and diminishing returns in performance remain major obstacles with many cores. Poor interaction between software and hardware, and bottlenecks in shared hardware structures often prevent scaling to many cores, even in applications where a high degree of parallelism is potentially available. In some cases, throwing additional cores at a problem may actually harm performance and increase power consumption. Better use of otherwise limitedly beneficial cores by software components such as hypervisors and operating systems can improve system-wide performance and reliability, even in cases where power consumption is not a main concern. In response to these observations, we evaluate an approach to throttle concurrency in parallel programs dynamically. We throttle concurrency to levels with higher predicted efficiency from both performance and energy standpoints, and we do so via machine learning, specifically artificial neural networks (ANNs). One advantage of using ANNs over similar techniques previously explored is that the training phase is greatly simplified, thereby reducing the burden on the end user. Using machine learning in the context of concurrency throttling is novel. We show that ANNs are effective for identifying energy-efficient concurrency levels in multithreaded scientific applications, and we do so using physical experimentation on a state-of-the-art quad-core Xeon platform.**

I. Introduction

Modern microprocessors are rapidly increasing in their degree of on-chip, thread-level parallelism. This trend is largely motivated by two observations: first, more performance is expected for a fixed transistor budget through on-chip, thread-level parallelism than through further exploitation of ILP; and second, the replication of less complex circuitry results in potentially more energy-efficient processors. As a result, chip manufacturers are producing multicore processors with a large number of cores per chip – or *many*-core processors. Current predictions estimate CMPs with 10's to 100's of cores becoming available in the next decade [1], and Intel has already demonstrated a working prototype with 80 cores [2].

Multicore microprocessors represent an inflection point for software, since they rely on high levels of parallelism extracted from applications to take full advantage of the cores available. A further, often overlooked requirement is that software needs to scale gracefully with the number of cores, and threads need to interact with the hardware in non-destructive manners. If the application is unable to take advantage of all cores provided by the processor, then either the application should be further parallelized and optimized to improve scalability on that particular architecture or the cores should be allocated differently between application and system software, possibly leaving some cores idle to conserve energy.

In this paper, we perform an in-depth analysis of the scalability of a set of multithreaded scientific applications that have already been extensively optimized for parallelism and locality. We do so on a recently introduced quad-core Xeon processor, and our findings indicate that while ample parallelism is available in the studied applications, threads interfere destructively for shared on-chip resources, often resulting in negligible performance gains through the use of more than two cores, or even significant performance losses when concurrency exceeds some threshold. Somewhat surprisingly, poor scaling occurs even at just four cores, indicating that future many-core microprocessors may expose severe scaling limitations. Furthermore, we observe that the scalability of individual applications is *phase-sensitive*, in that different phases of the parallel code in an application exhibit radically different scaling properties. Simultaneous with the performance consequences of poor scalability comes an increasing trend in power usage when using more cores.

In response to the observed scalability limitations, we present a runtime system that dynamically throttles the level of concurrency when doing so is expected to improve performance. Our runtime system, *ACTOR* (for Adaptive Concurrency Throttling Optimization Runtime system), includes the necessary infrastructure to detect program phases that may not scale well and to determine the level of concurrency that will improve performance as well as the optimal architecture-aware placement of threads onto specific processor cores for each phase. A phase in ACTOR is a user-defined region of parallel code encapsulating either a collection of parallel loops or a collection of basic blocks executed concurrently by multiple threads. Concurrency throttling and optimal thread placement by ACTOR cumulatively improve energy-efficiency by virtue of higher performance with sustained or reduced power consumption when processor cores are left idle.

In previous work we evaluated phase-sensitive concurrency throttling on a system of multiple simultaneous multithreaded processors [3]. To our knowledge, concurrency throttling has not been evaluated on a real multicore processor. Furthermore, in prior work, we rely on regression techniques for predicting optimal levels of concurrency and thread placement. Here we leverage machine learning, specifically artificial neural networks (ANNs). We use ANN-based performance prediction to identify the desired level of concurrency and the optimal thread placement. The ANNs are trained offline to model the relationship between performance counter event rates observed while sampling short periods of program execution and the resulting performance with various levels of concurrency. The derived ANN models allow us to perform online performance prediction for phases of parallel code with low overhead by sampling performance counters. The use of our ANN approach removes the burden of managing the training phase and providing domain-specific knowledge, two steps that are crucial to regression-based prediction strategies [4].

This paper makes three primary contributions. First, we analyze the scalability and energy-efficiency of multithreaded scientific applications on a recently introduced quad-core Intel Xeon processor. Second, we describe an ANN-based runtime adaptation mechanism to throttle concurrency. Third we evaluate its application to identify dynamically more energy-efficient concurrency levels and achieve higher performance with lower energy consumption in those parallel codes.

In the next section, we give a brief overview of related research. Section III discusses the scalability and power characteristics of multithreaded applications on a quad-core Intel Xeon processor. We describe an approach to identify energy-efficient concurrency levels based on applying our ANN approach to a set of performance counter samples in Section IV. In Section V, we present the results of our experiments with ANN-based concurrency throttling.

II. Related Work

Li and Martínez [5] develop a heuristic search approach to improve concurrency and use DVFS to optimize power consumption given a fixed performance requirement. The effectiveness of any search-based strategy is likely to decrease as the number of cores from which to choose grows. Li and Martínez artificially lengthen their benchmarks to provide enough iterations to reach a decision (up to fifty), whereas our prediction-based approach succeeds on applications with as few as ten iterations. They require hardware modifications to gather input on runtime power consumption. Their evaluation, unlike ours, did not use a real multicore system but instead used a simulated machine.

Previous work considered adapting concurrency at runtime via online performance predictions [3]. The major differences in that work are that it utilizes multiple linear regression to make the performance predictions across threading configurations. While the approach is successful, it requires fine-tuning a regression model with detailed architectural knowledge, whereas ANNs provide a non-linear model without user-provided domain knowledge. Further, we perform experiments on a state-of-the-art multicore processor in place of the SMP of Intel Hyperthreaded processors used previously, and we discuss how our results are likely to extend to future platforms with significantly more cores. Finally, we provide detailed analysis of the scalability of the applications and architecture here before presenting adaptation results.

ANNs have previously been used for performance prediction in the context of architectural space exploration [6]. In this work, the authors reduce the number of points that must be simulated in evaluating design alternatives in a thorough sensitivity study. The values of various microarchitectural parameters are used to predict the resulting performance of a given application by sampling (simulating) a subset of points in the design space. Our work, on the other hand, predicts performance based on event rates observed during a live execution using a model trained once that can subsequently be applied to any application.

Lee et al. [7] compare the effectiveness of non-linear regression and ANNs for predicting performance in the context of varying input parameters. Their findings suggest that, while prediction accuracies between the two approaches are comparable, each approach is advantageous in different contexts. However, they report that the training process is significantly simplified through the use of ANNs, and it is for this reason that we propose its use in this paper.

Marin and Mellor-Crummey [8] present a toolkit to measure and to model application characteristics semi-automatically in an architecture-neutral way. They predict application runtime using properties of the architecture, the binary, and the application inputs, and evaluate their predictions against measurements collected using hardware performance counters.

Carrington et al. [9] present an automated framework for predicting scientific application performance. Benchmark probes are used to create machine profiles and generate application signatures. They then use a convolution method to map signatures onto machine profiles. The approach requires generating several traces, and prediction accuracy is dependent on the trace sampling rate.

Yang et al. [10] present a cross-platform performance translation approach based on relative performance between the original and target platforms. They observe relative performance through partial execution of a parallel application by assuming the code is iterative and behaves predictably over time. This observation-based approach does not require program modeling, code analysis, or architectural simulation, but is rendered less accurate for different problem sizes or degrees of parallelization.

III. Multithreaded Scalability

This section presents the performance impact and energy-efficiency analysis of using additional cores for a range of parallel applications from the scientific domain. The recently introduced quad-core platform we use is by no means a many-core processor, but our experimental analysis indicates

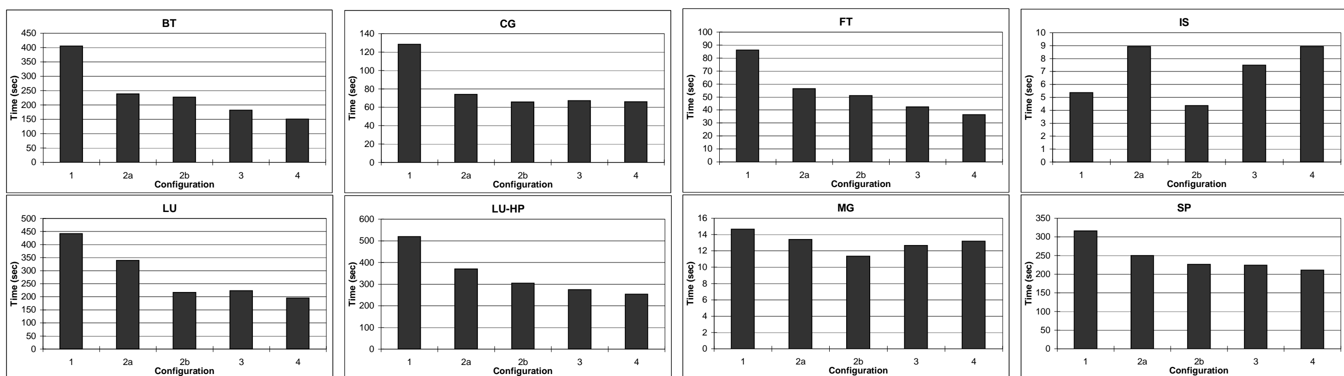

Fig. 1. Execution times by hardware configuration.

that scalability bottlenecks exist for many applications, even at such a small scale. Our experimental platform is a Dell Precision Workstation 390n running Linux kernel version 2.6.18. The particular machine has a single Intel Xeon quad-core processor (QX 6600). The processor is designed as two dual-core processors placed on a single chip. As such, there are two 4MB L2 caches, each shared between two of the cores. Hereafter, we refer to the two cores sharing a single cache as tightly coupled, and as loosely coupled otherwise. Additionally, the system has 2GB of main memory and a 1066MHz frontside bus.

In our evaluations, we use benchmarks from the NAS Parallel Benchmark suite version 3.2 [11] to represent modern scientific applications. The codes are implemented in either C or Fortran, have been parallelized using OpenMP, and have been extensively optimized for parallelism and locality [11]. We execute them under various levels of concurrency and under specific bindings of the threads to cores, performing experiments with five different threading configurations: first, a single thread bound to a single core (configuration 1), two threads bound to two tightly coupled cores (configuration 2a), two threads running on two loosely coupled cores (configuration 2b), three threads (configuration 3), and four threads running on all four cores (configuration 4).

A. Analysis of Application Scalability

Figure 1 displays the execution time results of our experiments. Many applications fail to scale beyond using two threads executing on loosely coupled cores. In fact, of the eight benchmarks, only three (BT, FT, LU-HP) experience substantial gains with the use of additional processor cores. The remaining benchmarks fall into two categories: those whose scalability curves flatten after two cores, and those who see large performance losses when using more cores. We examine each class of applications in turn.

The three applications that scale well are interesting because they show what can be achieved on this architecture. The fact that any applications can improve their performance through the use of each additional core demonstrates that scaling is not inherently limited on this quad-core processor, or on multicore processors in general. Rather, the problem stems from the interaction between particular applications and the underlying architecture. This group may provide insight into the types of program behavior that are amenable to multicore execution, although such an analysis is beyond the scope of this paper. Averaged over this class of application, a speedup of 2.37 times is seen compared to the sequential executions.

The second group of applications sees little performance gain or loss executing on more than two cores (CG, LU, and SP). Specifically, CG sees a speedup of a factor of 1.95 by using all four processor cores, however the same speedup is achieved with only two threads when executed on loosely coupled cores. Overall, this class of applications experiences only a 7.0% average performance improvement from using four cores compared to two.

The final group of applications, those that see substantial performance losses through the use of more processor cores, provides the most interesting results. Both MG and IS see their best times when two threads are executed on loosely coupled cores. The performance of MG improves by 11.3% when it uses four threads compared to the sequential execution, however the two thread execution is still faster by 14.0%. In contrast, IS is extremely communication-intensive and bandwidth-sensitive. The benchmark runs at a 40.0% performance loss using four threads compared to one but its performance improves by 22.8% using two threads. The two thread execution of IS on loosely coupled cores is 2.04 times faster than on tightly coupled cores, which suggests that the destructive interference in the shared L2, and the resulting memory bandwidth saturation, is largely to blame for the poor scalability of IS on this machine.

Averaged over all of the benchmarks, effective scaling only occurs up to two cores, with additional cores providing little to no gain. These results suggest that this architecture is not well suited for applications from the scientific domain. Poor scalability observed in these experiments is not an artifact of outdated systems, as results are obtained on a state-of-the-art system. If next-generation processors contain as many cores as generally expected, and the needs of scientific applications are not addressed, then the increased concurrency will likely lead to even poorer scalability than that observed here. In the next section, we address the power properties of the experimental

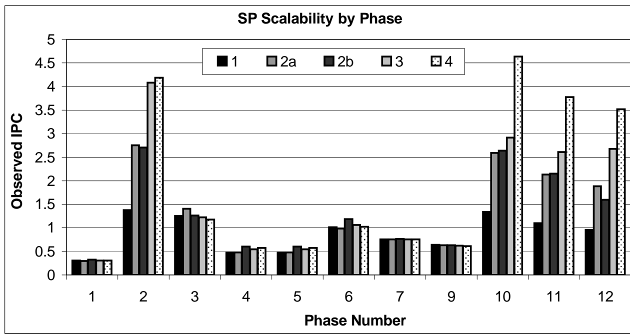

Fig. 2. IPCs observed during phases of SP for each thread configuration.

platform and analyze the consequences of poor scalability on the resulting energy efficiency.

Beyond the results for whole program scalability, we observe that the scalability of phases within an application vary greatly. For example, Figure 2 presents IPCs for each phase of the SP application when executing on each threading configuration. The graph demonstrates wide variations, with the maximum IPC for each phase ranging from 0.32 to 4.64, and the best performances coming on all configurations except those with three threads. We only show results for SP due to space limitations, but this diversity exists for other benchmarks in similar proportions. It is this heterogeneity that motivates us to perform adaptation at the phase granularity, allowing for potentially better performance than any single configuration.

B. Power and Energy Analysis

Figure 3 presents power and energy characteristics of our benchmarks (note that the y-axis does not begin at zero). For the five runs over which we measure execution times, we also collect energy consumption data using a Watts Up Pro multimeter. We compute average power for each application via recorded execution time and energy consumption. Numbers reported here represent a full system power profile, including CPU, memory, power supply, and other components.

Overall, we confirm that using more cores yields higher power consumption. Total system power consumption on four cores is 14.2% higher than on one core. This is unsurprising, since much of processor power is dictated by the activity on the processor. Higher utilization with more concurrency will generally increase power, but the same contention responsible for poor scaling observed above yields reduced power consumption in several cases. This indicates that cores and other processor components remain idle for extended time intervals. In such cases, measuring total system energy consumption during execution provides insight into whether throttling cores benefits both execution time and energy.

Applications that scale best experience the largest increases in power consumption with more cores, while those applications with the poorest scalability see negligible change in power (even power reductions). Consider BT, which achieves a factor of 2.69 speedup on four cores with an associated increase in power of a factor of 1.31, the largest of any application in both respects. This illustrates the potential energy efficiency of multicore architectures, with a decrease in energy consumption of a factor of 2.04. For scalable applications, the performance increase is much greater than the power increase, and energy efficiency improves on more cores.

On the other hand, MG performs best on two loosely coupled cores with a speedup of 1.29, which also represents its highest power-consuming threading configuration. The minimal relative decrease in power of 2.1% on four cores is dwarfed by the 18.1% higher execution time, so the resulting energy efficiency on four cores drops off considerably. Further, IS is 2.04 times faster on configuration 2b than on configuration 4, and consumes slightly less power on fewer cores. These poorly scalable applications demonstrate the potential loss in energy efficiency when using all available processor cores.

The final group of applications, those with flat scalability curves, simply fail to achieve increases in energy efficiency on this architecture. Taken all together, the suite of applications experiences a minor decrease of 0.7% in energy consumption scaling to four cores. Future generation systems with many cores will be further prone to scalability limitations, as applications will have to scale to more threads on architectures with a reduced compute-to-cache ratio [1].

IV. Concurrency Throttling

We now describe the performance prediction component of ACTOR, our runtime system that dynamically throttles concurrency to improve performance and energy efficiency. ACTOR adapts applications by identifying better-performing numbers of threads and thread placements for each phase. Phases are collections of parallel loops or basic blocks assigned for execution to different threads. We focus on a novel approach to concurrency throttling based on runtime performance prediction using ANNs on observed performance counter event rates.

A. Overview of Artificial Neural Networks

Machine learning studies algorithms that *learn* automatically through experience. For our problem, we focus on a particular class of machine learning algorithms called *artificial neural networks* (ANNs). Their many previous uses include microarchitectural design space exploration [6], workload characterization [12], and compiler optimization [13]. ANNs automatically learn to predict one or more targets (here, IPC) for a given set of inputs. We choose ANNs because they are flexible and well suited for generalized nonlinear regression, and their representational power is rich enough to express complex interactions between variables: any function can be approximated to arbitrary precision by a three-layer ANN [14]. They require no knowledge of the target function, take real or discrete inputs and outputs, and deal well with noisy data.

An ANN consists of layers of *neurons*, or switching units: typically, an input layer, one or more hidden layers, and an output layer. Input values are presented at the input layer and predictions are obtained from the output layer. Figure 4 shows an example of a fully connected feed-forward ANN. Every

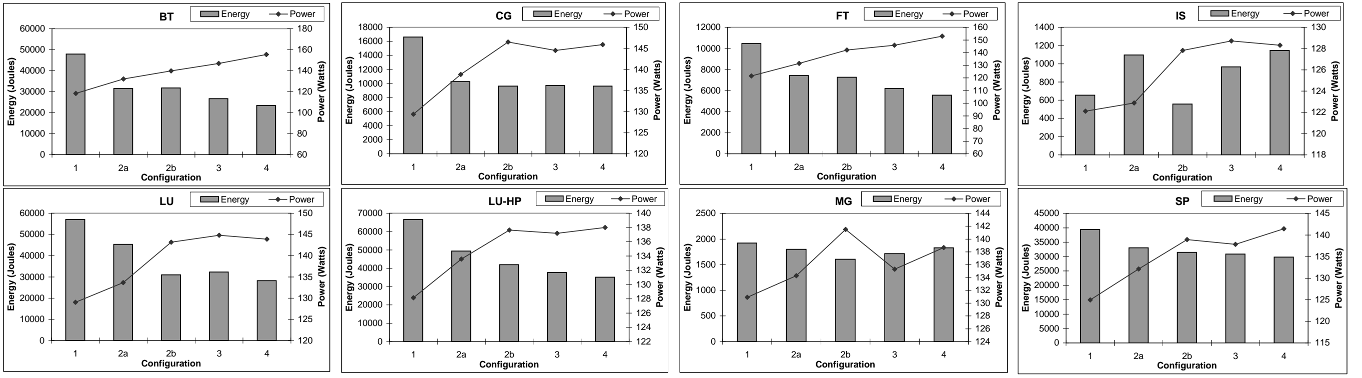

Fig. 3. Power and energy consumption by hardware configuration. The bottom-right graphs shows the geometric mean of the normalized energy and power consumption across all benchmarks.

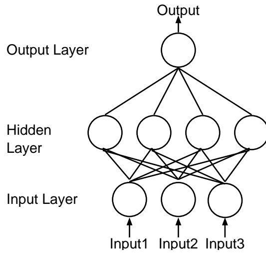

Fig. 4. Simplified diagram of fully connected, feed-forward ANN.

unit in each layer is connected to all units in the next layer by weighted edges. Each unit applies an *activation function* to the weighted sum of its inputs and passes the result to the next layer. Figure 5 [14] shows a unit with a sigmoid activation function. One can use any nonlinear, monotonic, and differentiable activation function. We use the sigmoid activation function for our models.

Training the network involves tuning edge weights via backpropagation, using gradient descent to minimize error between predicted and actual results. In this iterative process, the training samples are repeatedly presented at the input layer, and the error is calculated between the prediction and the actual target. The weights are initialized near zero and are updated using an update rule (similar to the one shown in Equation 1) in the direction of steepest decrease in error. As weights grow, the network becomes increasingly nonlinear.

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \quad (1)$$

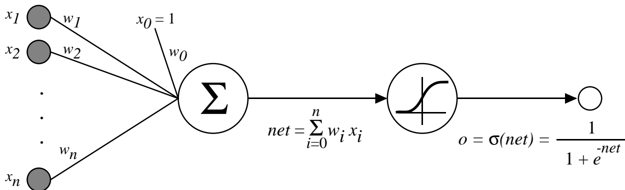

Fig. 5. Example of a hidden unit with a sigmoid activation function.

ANNs have a tendency to *overfit* on training data, leading to models that generalize poorly to new data despite their high accuracy on the training data. This is countered by using *early stopping* [15], where we keep aside a validation set from the training data and halt training as accuracy begins to decrease on this set. However, this means we lose some of our training data to the validation set. To address this, we use an ensemble method called *cross validation* to help improve accuracy and mitigate the risk of overfitting the ANN. This technique consists of splitting the training set into n equal-sized *folds*. Taking n=10, for example, we use folds 1-8 for training, fold 9 for early stopping to avoid overfitting, and fold 10 to estimate performance of the trained model. We train a second model on folds 2-9, use fold 10 for early stopping, and estimate performance on fold 1, and so on. This generates 10 ANNs, and we average their outputs for the final prediction. Each ANN in the ensemble sees a subset of training data, but the group as a whole tends to perform better than a single network because all data has been used to train portions of it. Cross validation reduces error variance and improves accuracy at the expense of training multiple models.

B. Concurrency Throttling Using Neural Networks

We model the effects of changing concurrency and thread placement. Hardware performance counter values collected during a brief sampling period at maximal concurrency become input to our ANN ensemble that predicts IPC for each phase on alternative configurations. The online sample period runs on as many cores as available to represent the greatest possible interference among threads, and resulting predictions estimate the degree to which contention will be reduced by throttling concurrency. Our modeling approach produces the following function for each desired target configuration, T, mapping observed event rates (e_i) on the sample configuration, S, to IPC on the target configuration:

$$I\hat{P}C_T = F_T(IPC_S, e_{(1,S)}, ..., e_{(n,S)}) \quad (2)$$

ACTOR collects predetermined performance counter values for the sample configuration and normalizes observed values

to the elapsed cycle counts, yielding an *event rate* associated with each counter. The prediction module, which we develop offline, uses these rates as input. We sort predictions and select the configuration with the highest predicted IPC for the corresponding program phase. We support the prediction of performance at phase granularity, as execution characteristics are likely to vary considerably from one phase to another within a single application [16]. Once a configuration is selected, our runtime library ensures all subsequent executions of the phase use the chosen concurrency and thread placement.

We derive the prediction module from ANNs that we train on the hardware counter values and IPCs from the target configurations. The performance counters are selected as a collection that represent performance-critical resources, such as caches and buses. We choose training applications representing a variety of runtime characteristics, as identified by the performance counters. During the short training period, patterns in effects of event rates on resulting training benchmark IPCs are observed and encoded in the ANN models.

Our system currently supports applications parallelized using OpenMP and instrumented with calls into ACTOR. Parallel regions in OpenMP tend to have consistent execution properties, and they also represent the finest granularity at which the number of threads can be changed at runtime, therefore we use them as program phases. ACTOR library calls are added at the beginning and end of each phase to initialize our runtime system, to collect performance counter values, to make performance predictions and to enforce concurrency decisions made for each phase.

We have previously experimented with both empirical search-based [17] and statistical prediction-based [3] determination of concurrency levels. Each of these strategies suffers from certain difficulties, and the use of ANNs in this context addresses these limitations. The configuration identification process for empirical searching [17] requires the online testing of a potentially large number of configurations, which is associated with a large degree of overhead that can reduce the gains through adaptation. While at most five configurations would need to be tested on our experimental platform, future generation systems with a large number of cores would require significantly more. Therefore, the benefits of prediction-based adaptation relative to searching will only grow in the future.

Regression-based models for performance prediction, on the other hand [3], have very low overhead. However, they do require significant effort and machine-specific training in the derivation of effective models of performance [4]. This labor-intensive, machine-specific training period may well render regression-based approaches unsuitable for use in many contexts. Since our approach automatically develops a model based on a collection of samples without requiring user-input and domain-specific knowledge, the minor costs associated with using ANNs, along with the comparable online overhead of performance counter collection and model evaluation, may make it more appropriate than regression-based models in a larger number of environments.

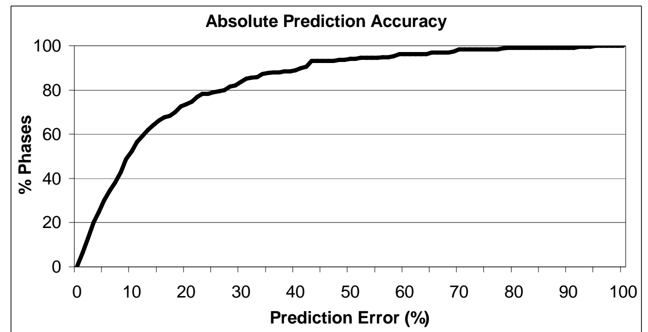

Fig. 6. Cumulative distribution function of prediction error.

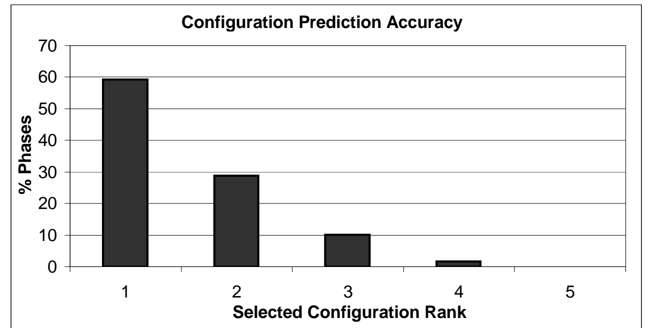

Fig. 7. Percent of phases for which each ranking configuration is selected.

V. Results

A. Evaluation of ANN-based Performance Prediction

For our experimental evaluation of ANN-based performance prediction and concurrency throttling, we used the same Intel quad-core experimental platform and benchmark suite as described in Section III. Performance counters were collected using PAPI version 3.5. We use each benchmark for evaluation by training as many models as there are applications, each time leaving one particular application out of the training process. In this way, we perform prediction for each application with a model that has never seen data from the target application. In practice, the model would generally be trained a single time with a given set of training applications, and would subsequently be used for any desired application, with possible refinements to reflect data from the current workload.

In our evaluation of the ANN-based predictor, we have selected a set of twelve hardware events representing the cache and bus behavior of the application. Our experimental platform only allows the simultaneous recording of two events. As a result, we employ collection across multiple timesteps to record all necessary events. However, several of our benchmarks contain very few iterations, in which case the sample execution period can consume a significant fraction of the overall execution time, thereby limiting the potential benefits of adaptation. In response to this situation, we limit the number of monitored timesteps to at most 20% of the total execution. While reducing the number of counters used in prediction will likely have some minimal effect on the prediction accuracy,

the benefits of using the improved concurrency level for a larger percentage of execution time is likely to outweigh the negative effects on accuracy. In the following evaluation we use a reduced number of events for the applications with fewer iterations (FT, IS, and MG).

Figure 6 gives a cumulative distribution function of the error of our ANN-based predictor, showing the percentage of samples that fall within increasingly higher levels of observed error. Specifically, we make predictions for four target configurations (1, 2a, 2b, and 3) and these results are accumulated over all predictions made. For each sample, error is calculated as $|(IPC_{obs} - IPC_{pred})/IPC_{obs}|$. Overall, the median error error is only 9.1%. Further, 29.2% of the predictions exhibit errors of less than 5%.

An alternative metric for evaluating the accuracy of the predictor in the context of concurrency throttling is the rate at which the optimal configuration is selected. Figure 7 shows the percentage of phases where each ranking configuration is selected. In 59.3% of phases, the single best configuration is correctly identified, and the second best configuration is selected in an additional 28.8%. In only one case out of 59 is the second worst configuration selected, and the worst is, in fact, never identified as optimal. These results show that ANN-based performance prediction is effective at identifying optimal or near-optimal concurrency levels.

B. *Concurrency Throttling Evaluation*

Figure 8 displays the results of our prediction-based concurrency throttling approach normalized by the four core execution, as well as those of the alternative execution strategies. We compare against using all available cores for multithreaded execution, which would normally be the default for a performance-oriented developer. We present results for two approaches based on the use of oracle-derived configurations. The one that we call the global optimal uses the best static configuration for an entire application. The second, the phase optimal, uses the best configuration for each phase. In each of these cases, this information would not normally be available, however they serve as points of comparison to evaluate the effectiveness of the library.

By using our approach for low overhead identification of improved concurrency levels, we see an average performance gain of 6.5% compared to the default strategy of simply using all available cores. Even BT, which scaled well on the four core machine, sees a substantial gain of 4.7% through our phase-aware adaptation strategy, which successfully identifies phases in BT that can be improved by concurrency throttling. Additionally, SP sees minor gains from more cores, however ACTOR is able to improve its performance by 5.2%.

When compared to the two oracle-derived strategies, we can see that ACTOR falls short of these oracular approaches, coming in 2.5% and 4.9% slower on average than the global and phase optimals, respectively. This shows potential benefits of improving prediction accuracy. Further, reduced online overhead of sampling is possible on architectures with more counter registers to reduce the number of rotations necessary for event collection.

One surprising result is that no power is saved through concurrency throttling, on average. We successfully leave cores idle, but it is likely that by changing the binding of threads, we are interfering with cache warmth. This, in turn, causes increases in bus and memory accesses, thereby increasing off-chip power consumption. So, while on-chip power consumption is reduced by small amounts, this is overcome by the off-chip increase. There are also cases, as pointed out in Section III, where power is increased through selecting reduced threading configurations with better performance. Together, these situations result in an average increase in power consumption of 1.5%. However, given the considerable improvement in execution time, the total energy consumption goes down by an average of 5.2%.

A popular metric in power-aware HPC is energy-delay-squared (ED^2), which considers power consumption but is more influenced by performance, commensurate with the heavy emphasis on performance in HPC. Given the large improvements in execution time, with very minor increases in power consumption, we experience significant reductions in ED^2, saving an overall 17.2%. However, it is clear that further gains are possible through this approach as the phase optimal execution sees a 29.0% improvement compared to using four cores. The most significant result occurs with IS, which shows that for applications that scale poorly, concurrency throttling is imperative to achieve energy-efficiency with a 71.6% improvement in ED^2.

VI. Conclusions

In this paper, we have evaluated the scalability and energy-efficiency of multithreaded scientific applications on a recently introduced Intel quad-core processor. As the number of cores per chip is continuing to increase, such a study is vital to understanding the future of both power-aware and high-performance computing. We found that for a large portion of our evaluation suite, scalability is quite poor and the resulting energy-efficiency at high degrees of concurrency suffers as a result. We improved the energy-efficiency for many of our applications by predicting the optimal number and placement of threads at runtime, and improved the average ED^2 by 17.2%. The success of our approach is largely due to a new performance prediction model based on applying ANNs to a set of performance counters collected online, which we show achieves high accuracy in terms of IPC prediction as well as identification of the optimal threading configuration. A major advantage of our approach over existing work is that, through ANNs, we significantly reduce the end-user cost without sacrificing accuracy.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-233024).

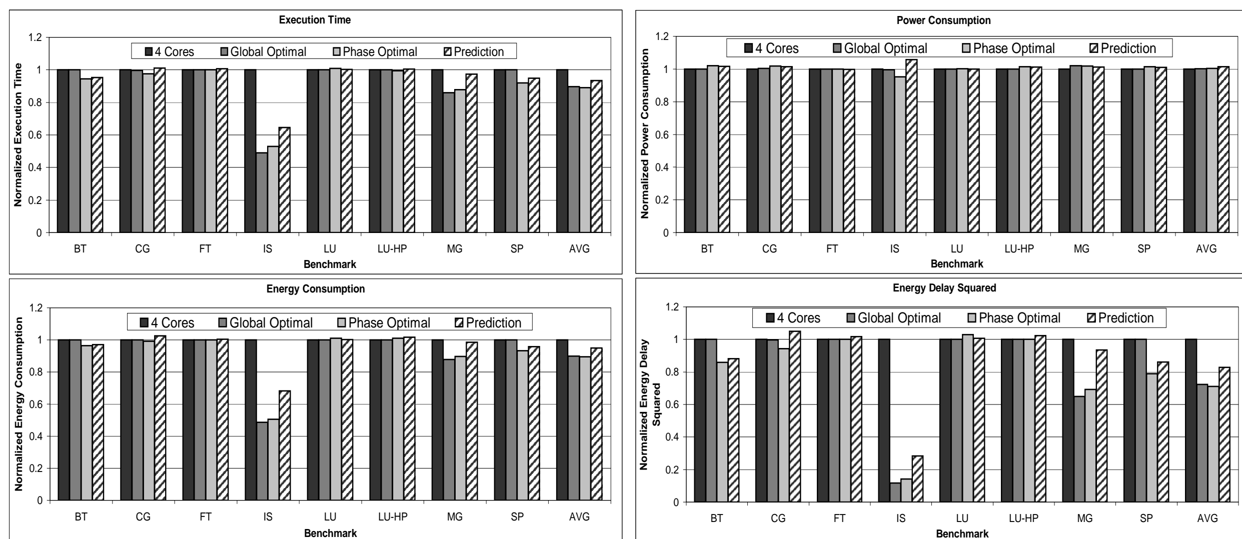

Fig. 8. Execution time, power consumption, energy consumption, and ED^2 of prediction-based adaptation compared to alternative execution strategies.

REFERENCES

[1] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling Scalability and Performance in a Large Scale CMP Environment," in *Proc. of the European Conference on Computer Systems*, Lisbon, Portugal, Mar. 2007.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. J. S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," in *Proc. of the International Solid State Circuits Conference*, San Francisco, CA, 2007, pp. 5–7.

[3] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction," in *Proc. of the 20th ACM International Conference on Supercomputing*, Queensland, Australia, June 2006.

[4] B. Lee and D. Brooks, "Accurate and Efficient Regression Modelling for Microarchitectural Performance and Power Prediction," in *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, June 2006.

[5] J. Li and J. Martínez, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors," in *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, Austin, TX, Feb. 2006.

[6] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," in *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, June 2006.

[7] B. Lee, D. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of Inference and Learning for Performance Modeling of Parallel Applications," in *Proc. of the International Symposium on Principles and Practices of Parallel Programming*, Mar. 2007.

[8] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, June 2004, pp. 2–13.

[9] L. Carrington, N. Wolter, A. Snavely, and C. Lee, "Applying an automatic framework to produce accurate blind performance predictions of full-scale HPC applications," in *Department of Defense Users Group Conference*, June 2004.

[10] T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2005.

[11] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," NASA Ames Research Center," Technical Report NAS-99-011, Oct. 1999.

[12] R. M. Yoo, H. Lee, K. Chow, and H.-H. S. Lee, "Constructing a non-linear model with neural networks for workload characterization." in *IISWC*, 2006, pp. 150–159.

[13] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *CF '07: Proceedings of the 4th international conference on Computing frontiers*. New York, NY, USA: ACM Press, 2007, pp. 131–142.

[14] T. Mitchell, *Machine Learning*. Boston, MA: WCB/McGraw Hill, 1997.

[15] R. Caruana, S. Lawrence, and C. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Proc. Neural Information Processing Systems Conference*, Oct. 2000.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[17] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online Strategies for High-Performance Power-Aware Thread Execution on Emerging Multiprocessors," in *Proc. of the Workshop on High-Performance Power-Aware Computing*, Rhodes, Greece, Apr. 2006.