Comparing Scalability Prediction Strategies on an SMP of CMPs

Karan Singh[1], Matthew Curtis-Maury[2], Sally A. McKee[3], Filip Blagojević[4], Dimitrios S. Nikolopoulos[5], Bronis R. de Supinski[6], and Martin Schulz[6]

[1] Computer Systems Lab
Cornell University
Ithaca, NY USA
karan@csl.cornell.edu
[2] NetApp, Inc
Research Triangle Park, NC USA
mcm@netapp.com
[3] Computer Science and Engineering
Chalmers University of Technology
Gothenburg, SWEDEN
mckee@chalmers.se
[4] Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA USA
fblagojevic@lbl.gov
[5] Institute of Computer Science
FORTH
Haraklion, GREECE
dsn@ics.forth.gr
[6] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA USA
{bronis,schulzm}@llnl.gov

Abstract. Diminishing performance returns and increasing power consumption of single-threaded processors have made chip multiprocessors (CMPs) an industry imperative. Unfortunately, poor software/hardware interaction and bottlenecks in shared hardware structures can prevent scaling to many cores. In fact, adding a core may harm performance *and* increase power consumption. Given these observations, we compare two approaches to predicting parallel application scalability: multiple linear regression and artificial neural networks (ANNs). We throttle concurrency to levels with higher predicted power/performance efficiency. We perform experiments on a state-of-the-art, dual-processor, quad-core platform, showing that both methodologies achieve high accuracy and identify energy-efficient concurrency levels in multithreaded scientific applications. The ANN approach has advantages, but the simpler regression-based model achieves slightly higher accuracy and performance. The approaches exhibit median error of 7.5% and 5.6%, and improve performance by an average of 7.4% and 9.5%, respectively.

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 143–155, 2010.

1 Introduction

Modern high performance computing generally trades performance for lower power consumption. However, with now-prevalent chip multiprocessors (CMPs), the best performance often yields the greatest energy efficiency. *Dynamic Concurrency Throttling* (DCT) increases parallelism for computationally intensive code phases that scale well, but reduces it when memory bandwidth demands available capacity. Concurrency throttling benefits small-scale shared memory platforms (SMPs) [2], and increases in importance with the degree of on-chip, thread-level parallelism.

While concurrency throttling has demonstrated benefits, the best implementation remains an open question. Any concurrency throttling technique must have low runtime cost, but complex interactions between problem size, control flow, and parallel efficiency make static or off-line techniques unlikely to deliver maximal performance or power efficiency. Techniques that search for the optimal concurrency levels online via timing analysis can be impractical on large scale systems, particularly for applications with phase-sensitive behavior or sensitivity to placement of threads on cores [2]. Here we investigate phase-sensitive online mechanisms to *predict* the optimal degree of concurrency of isolated code regions. We implement several phase-sensitive predictors of optimal concurrency within the context of the Adaptive Concurrency Throttling Optimization Runtime (*ACTOR*) system [2]. We use those predictions to control concurrency levels and thread placement. Here, we define a phase to be 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.

This paper makes four primary contributions. First, we rigorously compare the efficacy of two prediction techniques for online concurrency throttling: multiple linear regression and Artificial Neural Networks (ANNs). Second, we provide an improved understanding of phase-sensitivity with respect to scalability and energy efficiency for multithreaded scientific applications on multicore SMPs. Third, we evaluate the effectiveness of concurrency throttling in such an environment through direct measurement of power consumption. Fourth, we conclude that (for this study) a multiple linear regression model is sufficiently accurate to achieve significant performance gains, and, on average, outperforms an ANN model for overall performance and power consumption in an online DCT system. Nonetheless, the ANN-based model delivers the optimal DCT decision more frequently, and requires minimal effort on the modeler's part when compared to linear regression.

2 Related Work

Curtis-Maury et al. [2] evaluate a runtime scalability prediction model for adapting concurrency. They develop a linear regression-based approach for mapping observed hardware event counters to performance estimates at varying levels of concurrency and different thread placements. We leverage their approach, and compare it to using ANNs, all in the unified context of concurrency throttling.

Li and Martínez [12] propose a heuristic search approach to improve concurrency and DVFS levels for optimizing power with respect to a fixed performance target. The

effectiveness of any search-based strategy likely decreases as the number of cores increases. Search-based strategies require $O(cores)$ samples of an application's execution phases, whereas prediction-based approaches require $O(1)$ samples [2].

Sasaki et al. [15] adapt our multiple linear regression models [2] to predict performance at varying DVFS settings. Their models use performance counter data to predict scalability levels with lowest power consumption that deliver given performance levels (on a per-phase basis). They focus on concepts and modeling, and omit empirical validation. Such prediction-based DVFS adaptation is orthogonal to our research, and could be applied synergistically with prediction-based concurrency throttling to identify better hardware/software configurations.

ANNs have been used for performance prediction in the context of architectural design space exploration. Ipek et al. [6] reduce the number of design points that must be simulated in evaluating design alternatives via thorough sensitivity studies. 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. Lee et al. [10] use piecewise polynomial regression to achieve similar goals. Our contribution is a fair, empirical analysis of two online performance-prediction schemes based on models derived from event rates observed during live execution.

Lee et al. [11] compare the effectiveness of piecewise polynomial regression and ANNs to predict performance in the context of varying input parameters. Their results suggest that prediction accuracies between the approaches are comparable, but each approach is advantageous in different contexts. However, the training process is greatly simplified for ANNs. There is overlap between their work and what we present herein, but we evaluate performance predictors in a different context. Specifically, we compare training schemes for online predictors that assess parallel application scalability to choose an appropriate number of parallel threads to run. We also study overheads for applying the models at runtime, paying particular attention to model sensitivity to phase changes and phase lengths. The linear regression model we evaluate reduces the end-user burden during model specification, but still delivers accurate predictions.

3 Multithreaded Scalability

Processor vendors now provide increasing degrees of parallelism within a single chip. As a result, the scalability of multithreaded applications becomes a critical issue. Processors with tens or even hundreds of cores may become available within the decade [14], but we do not yet know whether we can capitalize on the parallelism afforded by these architectures. We must consider the practical scalability and energy efficiency of representative applications for next-generation systems.

3.1 Experimental Setup

The quad-core processors we use in this work are by no means many-core, but our experimental analysis indicates that scalability bottlenecks exist for many applications, even at this scale. Our experimental platform contains two Intel Xeon E5320 quad-core processors running at 1.86GHz. The cores are paired, with each pair sharing a 4MB L2

cache. The system has 4GB of main memory and a 1066MHz frontside bus, and it runs Linux kernel version 2.6.22. We assume a dedicated system, but adapting our approach to general time-sharing systems is straightforward. In all experiments, we collect full system energy per run using a Watts Up Pro power meter [5], and we compute average power consumption based on execution time and total energy consumption.

We use benchmarks from NAS v3.2 [7] to analyze the scalability of parallel applications on our experimental platform. We use class B inputs, which consume between 5% and 30% of system memory. The C and Fortran codes are parallelized using OpenMP, and have been extensively optimized for parallelism and locality [7]. We choose a subset of the suite, since applications like LU (which uses a pipeline structure with stages identified by thread IDs) are not amenable to our approach. We execute the benchmarks under various levels of concurrency and under specific bindings of threads to cores, experimenting with 10 threading configurations. The notation (P, C) indicates execution using P processors, each with C cores. Given the architecture's asymmetry, two threads can execute on one chip with shared or private caches, denoted $2s$ and $2p$, respectively.

3.2 Analysis of Application Scalability

Figure 1 shows the execution times of our experiments. We find that scalability is limited in most cases. Of our six benchmarks, only one (BT) scales to the maximum number of available cores. The remaining benchmarks fall into two categories: those whose scalability curves flatten after two cores, and those who suffer significant performance losses when using more cores. We examine each class.

BT illustrates that scalability on the quad-core processors is not inherently limited, since it exhibits a high computational intensity. Lowest energy consumption also occurs at full concurrency, because scaling achieves significantly higher performance with only incremental additional power consumption. Poor scalability of other applications occurs due to particular execution properties (most notably high memory access rates). CG, FT, and SP exhibit limited performance scaling from additional concurrency, and

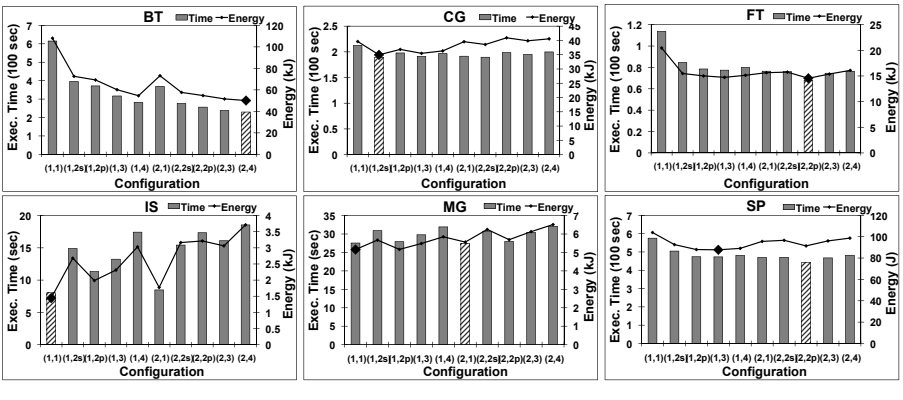

Fig. 1. Execution times and energy consumption by hardware configuration. Configurations with the best performance and energy are marked with stripes and large diamonds, respectively.

speedups plateau when concurrency crosses the boundary of a single quad-core processor. When comparing to single core performance, these benchmarks have a mean speedup of 1.25× when using all cores. FT and SP obtain speedups of 1.42× and 1.20×, respectively, from using the four cores of a single processor, but see minimal benefit from using the second processor. In these three benchmarks, using fewer cores reduces energy consumption without sacrificing performance. IS and MG exhibit 2.31× and 1.17× slowdowns, respectively, due to memory intensity and the limited memory bandwidth when running on all cores on our platform. Furthermore, IS observes a 1.32× performance improvement when the entire cache is allocated to a single core, as compared to sharing the cache between two cores: destructive interference in the shared L2 causes memory bandwidth saturation and poor scalability. IS and MG both achieve minimal energy consumption using a single thread; additional concurrency increases energy consumption by 157.1% and 26.3%, respectively.

Execution properties are not static within a given application [16]. Many exhibit phased behavior: program characteristics vary at repeating intervals. In our test cases, program phases exhibit widely varying scalability and energy efficiency characteristics, even within a single application. For example, SP contains phases that experience power/performance optimality at six distinct configurations, with full concurrency yielding speedups ranging from 0.68× to 3.24×. The optimal configuration for any given program phase may differ from surrounding phases. Runtime identification of poorly scalable phases could support fine-grain dynamic concurrency throttling to execute each phase with a more efficient threading configuration.

4 Dynamic Concurrency Throttling

Section 3 demonstrates improved performance when using fewer cores. This is due to limited scalability of several parallel execution phases, including phases with collective operations that force processor serialization, phases that incur contention for shared on-chip or off-chip resources, and phases with inherently limited algorithmic concurrency.

Concurrency throttling, like dynamic voltage and frequency scaling (DVFS), has beneficial processor power management properties. By throttling processors, software can reduce both dynamic and static power consumption faster than DVFS. DVFS mainly targets dynamic power consumption. Increases in static (leakage) power consumption with each processor generation diminish DVFS's potential for reducing power without penalizing performance. In contrast, concurrency throttling may still achieve substantial power savings on both fronts [4].

Compilers can be used to effect concurrency throttling on optimized, parallel codes [8]. Alternatively, runtime phase analysis via direct search algorithms [12] or performance prediction across system and program configurations with varying degrees of concurrency [2] may prove more effective. Compiler methods are effective for codes with simple memory access and thread execution patterns. However, they are constrained by limitations of compiler analysis and compiler-based performance prediction.

Runtime search methods can discover optimal or near-optimal concurrency levels for phases of parallel code separated by synchronization or communication operations. However, search methods may require many executions of a phase to converge on an

efficient operating point. The number of executions depends both on the number and on the topology of the cores [2]. Different mappings of a given number of threads on a given topology may vary dramatically in performance. With tiled embedded processors with 64 to 512 cores (Tilera's Tile64 and Rapport's Kilocore) exhaustive or heuristic search of program and system configurations may be prohibitively expensive.

Runtime performance prediction overcomes limitations of direct search methods at the potential cost of reduced accuracy in identifying optimal operating points. These approaches test fewer configurations to reduce online overhead, but their efficacy depends on prediction accuracy. We present two scalability prediction models in the next section, evaluating them for prediction accuracy and success at identifying optimal configurations per phase when used with the ACTOR concurrency throttling runtime system [2].

Unfortunately, concurrency throttling is not feasible in all applications and programming models. In principle, concurrency throttling can be applied transparently to applications where neither the parallel computation nor data distribution depend on the number and topology of the processors. Shared-memory programming models such as OpenMP and Transactional Memory meet these requirements, whereas distributed-memory programming models such as MPI need application and/or runtime system modifications to benefit. Programming models where parallelism is expressed independently of the number and type of processors are essential to simplifying the process of parallel programming [1]. Our work supports such programming models.

5 Strategies for Scalability Prediction

We develop scalability predictors to estimate performance at the granularity of program phases and use those estimates to guide changes in concurrency and thread placement. Predictor inputs are hardware performance monitoring counter (PMC) values collected during a brief, online sampling period that uses a subset of all possible configurations. We predict IPC (excluding busy-waiting portions of the execution) for each phase on the remaining system configurations to estimate the performance impact of throttling concurrency (note that we do not use IPC to *measure* performance). Our modeling approach produces a function for each target configuration. These functions map observed event rates from sample configurations to cumulative IPC on the target configuration. During monitoring, we account for synchronization (and the resulting busy-waiting), so IPC becomes proportional to performance [2].

The predictors are components of the ACTOR runtime system [2]. This system collects performance counter values for the sample configurations and normalizes observed values with respect to elapsed cycle counts. This yields an *event rate* associated with each counter. The prediction module, which we train offline, uses these rates as input. We sort predictions and select the configuration with highest predicted IPC for the corresponding program phase. We facilitate performance prediction at the granularity of parallel blocks of code separated by synchronization points. After selecting a configuration, the runtime system ensures that all subsequent executions of the phase use the chosen concurrency and thread placement. Our runtime system currently supports OpenMP applications that are instrumented with calls into ACTOR.

We systematically select configurations for online sampling of predictor training data. We choose sample configurations that provide best prediction accuracy in the training set across both processors and cores. Multiple tests capture effects of concurrency throttling at different levels of parallelism in the hardware, such as cores sharing an L2 cache in the same processor, cores not sharing cache space in the processor, and cores residing on different physical processors. As numbers of cores continue to increase, it may become appropriate to reduce overhead by pruning configuration space (e.g., via uniform sampling). Such measures are unlikely to negatively impact the effectiveness of our approach.

We train the prediction model using hardware performance counter (PMC) event rates and IPCs collected on sample and target configurations. We select performance counters that show the highest correlation with target IPCs in the training samples. We choose training applications that contain phases with a variety of runtime characteristics, such as scalability, IPC, memory boundedness, and locality, as identified by the hardware counters. During the short, offline training period, each model learns from patterns of effects in training benchmark event rates on IPCs. We construct models using artificial neural networks and linear regression, and compare their advantages.

5.1 Training with Linear Regression

We model the IPC on a given target configuration as a linear function of the input event rates observed for each sample configuration. By minimizing the sum of squares of the residuals (errors observed for each training sample) linear regression derives a set of coefficients by which to multiply those rates. Each coefficient specifies the effect of an event rate on performance, and provides additional insight into the hardware/software interaction for a given architecture [11].

Linear regression has its limitations. First, it only considers terms that are explicitly included in the input, and deriving an accurate model requires encoding detailed architectural knowledge. Second, we must explicitly model interactions between multiple events or between repetitions of a particular event across sample configurations. These required specifications of architectural knowledge and event interactions increase the modeler's burden. Further, it can be hard to decipher the correlation between event rates and IPC. We find that event rates have a multiplicative rather than additive effect on the sample configuration IPC. Thus, we include a term representing the product of each event rate with IPC [3]. Another limitation is that the derived model cannot capture non-linear effects of predictor variables on the response variable. Nonetheless, the effect on events can be captured sufficiently accurately using a linear model [9].

5.2 Training with ANNs

Artificial Neural Networks (ANNs) automatically learn to predict one or more targets (here, IPC) for a given set of inputs. ANNs are flexible and well suited for generalized nonlinear regression. Their representational power is rich enough to express complex interactions between variables; for instance, three-layer ANNs can approximate any function to arbitrary precision [13]. They require no knowledge of their target functions

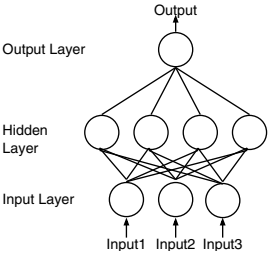

Fig. 2. Simplified diagram of fully connected, feed-forward ANN

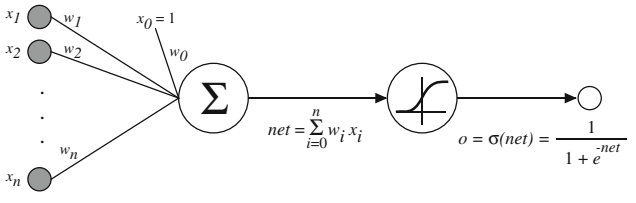

Fig. 3. Example of a hidden unit with a sigmoid activation function

and can handle real or discrete inputs and outputs. ANNs have high potential for predicting scalability because they can represent arbitrary functions at arbitrary precision and can handle noisy inputs robustly.

An ANN consists of layers of *neurons*, or switching units: an input layer, one or more hidden layers, and an output layer. Input values (here the observed event rates on sample configurations) are presented at the input layer, and predictions are obtained from the output layer. Figure 2 shows an example multilayer, fully-connected, feed-forward ANN. Every unit in each layer is connected to all units in the next by weighted edges. Each unit applies an *activation function* to the weighted sum of its inputs, passing the result to the next layer. Any nonlinear, monotonic, and differentiable activation function suffices. We use the sigmoid activation function shown in Figure 3 [13]. We construct our models using the *Fusion Predictive Modeling Tools* [17] with default network and training parameters: a three-layer ANN with one hidden layer of 16 units.

This approach automatically develops models by training on a collection of samples, requiring no domain-specific knowledge. Further, ANNs can capture complicated and non-linear relationships between inputs and outputs. However, ANNs require a relatively large amount of computation. Another limitation is that ANNs act as black boxes, yielding little insight into how inputs and interactions affect predictions.

6 Results

We compare performance of models trained using multiple linear regression and ANNs. We first compare the accuracy of each model in terms of both absolute accuracy and

optimal configuration identification. Then we compare performance and energy improvements observed by performing dynamic concurrency throttling using each approach. Experiments presented in this section are run on the platform in Section 3.1.

6.1 Performance Prediction Evaluation

For model training, we use all the phases from the UA benchmark. We use phases from all remaining benchmarks for evaluation. We select UA for training because it contains many phases exhibiting wide variation in execution properties (such as memory intensity, throughput, and scalability). For test configurations, we use one at full concurrency and a second using a single processor with three cores active. This provides variation in both the number of processors and cores. We make predictions for all of the remaining eight configurations. Our experimental platform only has two hardware event counter registers. We record instructions retired and L1 data cache accesses, since we find them to have the strongest correlation to IPC.

Figure 4 presents results for both prediction methods. The left graph gives a CDF of prediction error, showing the percentage of test cases that fall within increasingly higher levels of observed error. We calculate error as $|(IPC_{obs} - IPC_{pred})/IPC_{obs}|$, where IPC_{obs} corresponds to the measured cumulative IPC and IPC_{pred} corresponds to the cumulative IPC estimated by the model. Regression is slightly more accurate for scalability prediction. The percentage of tests for regression within any given threshold is always greater than or equal to that for ANNs. Overall, the median error for regression is 5.6% (9.4% mean) and for ANNs is 7.5% (12.8% mean): we achieve low error rates despite very complex scaling patterns.

An alternative metric for evaluating our models is how often correctly identify the best configuration (or one with very similar performance). The right graph of Figure 4 presents the percentage of tests for which the approaches select each ranking configuration. This graph shows comparable accuracy for both training approaches. The predictors identify nearly optimal configurations most of the time. Regression correctly identifies the best configuration in 28.6% of phases, and selects one of the top five for 85.7% of phases. ANNs select the best configuration for 32.1% of phases, and one of the top five for 75.0%. ANNs are more effective at identifying the single best operating point, but less so for the following top configurations. For phases with poor scalability it becomes difficult for the models to differentiate among multiple configurations with near-identical performance. Fortunately, mispredicting the optimal configuration

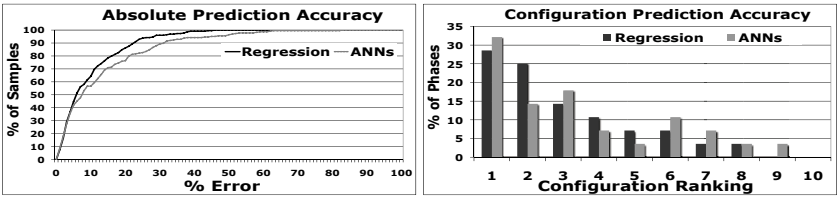

Fig. 4. Cumulative distribution function (left) and percent of phases for which each ranking configuration is selected (right) for each prediction strategy

in phases with near-identical performance across different configurations does not harm application performance significantly, making the overall impact tolerable.

One interesting difference between regression and ANNs is the computational overhead of model evaluation. We find that performing all eight necessary predictions for each application phase takes approximately 100K cycles with regression and 1M cycles with ANNs. While this ten-fold difference is quite large, in neither case will the overhead impact the performance of long-running parallel applications. We find that in the context of scalability prediction both approaches achieve high accuracy, but regression is slightly more accurate, on average. This results from the relatively small training size, which proves sufficient to derive an accurate model using the simple linear approach, but insufficient for training an ANN for high accuracy. The tradeoff lies in the significant effort required on the modeler's part for linear regression compared to ANNs. Also, our results could be different if we had more training data or performed a different study, but we expect them to remain comparable in terms of accuracy.

6.2 Concurrency Throttling Evaluation

Figure 5 displays the results of each concurrency throttling approach normalized to execution with all available cores. Figure 6 presents the geometric mean of these

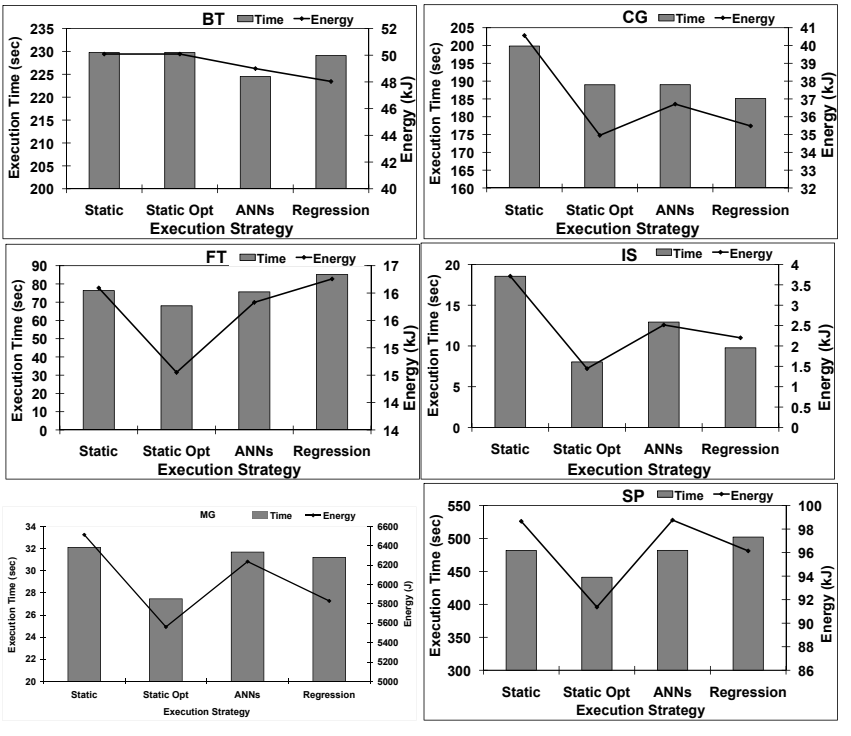

Fig. 5. Execution time and energy consumption of various execution strategies

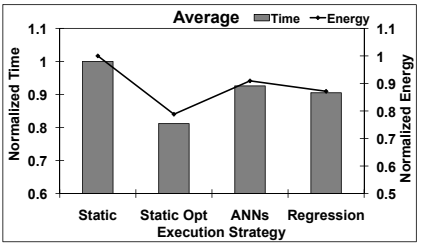

Fig. 6. Geometric mean of execution time and energy consumption

results. We compare to using all cores for multithreaded execution, a natural choice for a performance-oriented developer. To put these results in perspective, we compare them to an approach based on oracle-derived configurations (*static optimal*), where we use the best static configuration for the whole application. Acquiring this data requires an exhaustive search of the configuration space using complete application executions. We present this information as a point of comparison to evaluate the effectiveness of our approach.

DCT using a linear regression predictor yields a mean performance improvement of 9.5% over full concurrency. By comparison, ANNs provide a 7.4% improvement. The performance of regression-based prediction suggests that a linear model can accurately capture the effects of event rates on performance. If the data did not tend to be linear trend, then ANNs would have an advantage — even with a small training size — and would achieve higher accuracy and better performance than regression.

Despite the accuracy and performance advantage of regression overall, each approach wins in three of six cases. The mean difference between models is largely due to IS, for which regression improves performance by 47.3%, compared to 30.3% for ANNs. In all cases, ANNs maintain or improve performance relative to maximum concurrency, while regression causes a 4.2% and 11.6% slowdown for SP and FT, respectively. ANNs deliver a modest 2.3% speedup for BT, the one benchmark that scales fairly well to all cores, demonstrating the potential advantage of performing adaptation at the phase level. Linear regression improves CG performance over the static optimal configuration. Results for individual benchmarks are inconclusive, although ANNs appear to predict high scalability better, whereas linear regression predicts poor scalability better. Regression-based prediction optimizes energy consumption better than ANNs. The former lowers consumption in five of six benchmarks, suggesting that it better identifies opportunities to conserve power even if execution time is not always reduced.

Two reasons lead both prediction-based adaptive approaches to fall short of the static optimal configuration in all but one benchmark. First, even though the prediction-based approaches have relatively minimal overhead (the two test configurations), this overhead can be significant for applications with few iterations; an oracle derives the static optimal so it has no overhead. Second, any prediction-based approach has some error, which limits potential savings relative to static offline approaches.

7 Conclusions

We compare multiple linear regression and artificial neural networks for predicting parallel application scalability on a multicore multiprocessor. The models use hardware event counter values collected on various threading configurations and capture the relationships between the counters and per-phase scalability. Linear regression and ANNs, respectively, exhibit median error rates of only 5.6% and 7.5% while delivering performance improvements of 9.5% and 7.4% and reducing power consumption. These results demonstrate the potential of dynamic concurrency throttling on multicore systems. In this study, multiple linear regression proves slightly more effective than artificial neural networks on average. We conclude that a linear model sufficiently captures the relationship of hardware events to performance. Nonetheless, the ANN-based model delivers the optimal DCT decision more frequently, and requires minimal effort on the part of the modeler. We will continue to explore predictors on larger multicore systems to understand how to adapt our prediction methods for huge configuration spaces.

Acknowledgments

This research is supported by grants from the National Science Foundation (CCR-0346867, CCF-0702616, CCF-0715051, CNS-0521381, CNS-0720750, CNS-0720673), the U.S. Department of Energy (DE-FG02-06ER25751, DE-FG02-05ER25689), the European Commission (FP7-224759, FP7-217068, FP7-216181, FP6-27648), IBM, and Virginia Tech (VTF-874197). The work presented here was partly performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-401561).

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Science, University of California at Berkeley (December 2006)
2. Curtis-Maury, M., Blagojevic, F., Antonopoulos, C.D., Nikolopoulos, D.S.: Prediction-based power-performance adaptation of multithreaded scientific codes. IEEE Transactions on Parallel and Distributed Systems (October 2008)
3. Curtis-Maury, M., Singh, K., McKee, S.A., Blagojevic, F., Nikolopoulos, D.S., de Supinski, B.R., Schulz, M.: Identifying energy-efficient concurrency levels using machine learning. In: Proc. International Workshop on Green Computing (September 2007)
4. de Langen, P., Juurlink, B.: Leakage-aware multiprocessor scheduling for low power. In: Proc. 20th IEEE/ACM International Parallel and Distributed Processing Symposium (April 2006)
5. Electronic Educational Devices. Watts Up PRO (May 2009), `http://www.wattsupmeters.com/`
6. İpek, E., McKee, S.A., Singh, K., Caruana, R., de Supinski, B.R., Schulz, M.: Efficient architectural design space exploration via predictive modeling. ACM Transactions on Architecture and Code Optimization 4(4) (2008)

7. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report NAS-99-011, NASA Ames Research Center (October 1999)
8. Kadayif, I., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Kolcu, I.: Exploiting processor workload heterogeneity for reducing energy consumption on chip multiprocessors. In: Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition (February 2004)
9. Karkhanis, T.S., Smith, J.E.: A first-order superscalar processor model. In: Proc. 31st IEEE/ACM International Symposium on Computer Architecture (June 2004)
10. Lee, B.C., Brooks, D.: Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems (June 2006)
11. Lee, B.C., Brooks, D., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: Methods of inference and learning for performance modeling of parallel applications. In: Proc. ACM Symposium on the Principles and Practice of Parallel Programming (March 2007)
12. Li, J., Martínez, J.F.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: Proc. 12th IEEE Symposium on High Performance Computer Architecture (February 2006)
13. Mitchell, T.M.: Machine Learning. WCB/McGraw Hill, Boston (1997)
14. Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A.M., Rajagopalan, M., Hudson, R.L., Petersen, L., Menon, V., Murphy, B.R., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling scalability and performance in a large scale CMP environment. In: Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems (March 2007)
15. Sasaki, H., Ikeda, Y., Kondo, M., Nakamura, H.: An intra-task DVFS technique based on statistical analysis of hardware events. In: Proc. ACM Computing Frontiers Conference (May 2007)
16. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proc. 8th ACM Symposium on Architectural Support for Programming Languages and Operating Systems (October 2002)
17. Singh, K., McKee, S.A., de Supinski, B.R., Schulz, M.: Using machine learning to explore huge parameter spaces for high end computing applications: Tools and examples. Technical Report CSL-TR-2007-1049, Cornell Computer Systems Lab (July 2007)