Proceedings of the 4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART'10)

co-located with HiPEAC'10

Pisa, Italy
January 24th, 2010

http://cTuning.org/workshop-smart10

Table of contents:

Workshop foreword:

Welcome to the Fourth Workshop on Statistical and Machine Learning Approaches to ARchitectures and CompilaTion (SMART'10). The workshop series is intended as a forum for the growing community that studies the application of machine learning techniques to translation and the design of machines. It also has as a goal to increase awareness about the tremendous importance of advanced techniques to address the complexity of today's machines and compilers.

This year, we received eight submissions; each was evaluated by at least four members of the program committee with an average of 4.6 reviews. Five of the submitted papers were accepted, based on their quality and focus, for presentation at the workshop. Besides the papers, the program this year also includes a keynote presentation by Prof. Keith Cooper on the PACE project.

We hope that this year's attendees will find the ideas presented in the papers and the keynote presentation interesting and useful. Best wishes for a productive meeting!

John Cavazos, Grigori Fursin, David Whalley (program chair)
SMART'10 organizers

Automatic Selection of Machine Learning Models for Compiler Heuristic Generation [⋆]

Paul Lokuciejewski[1], Marco Stolpe[2], Katharina Morik[2], Peter Marwedel[1]

[1] Computer Science 12 (Embedded Systems Group)
[2] Computer Science 8 (Artificial Intelligence Group)
TU Dortmund University
D-44221 Dortmund, Germany
`FirstName.LastName@tu-dortmund.de`

Abstract. Machine learning has shown its capabilities for an automatic generation of heuristics used by optimizing compilers. The advantages of these heuristics are that they can be easily adopted to a new environment and in some cases outperform hand-crafted compiler optimizations. However, this approach shifts the effort from manual heuristic tuning to the model selection problem of machine learning – i. e., selecting learning algorithms and their respective parameters – which is a tedious task in its own right.
In this paper, we tackle the model selection problem in a systematic way. As our experiments show, the right choice of a learning algorithm and its parameters can significantly affect the quality of the generated heuristics. We present a generic framework integrating machine learning into a compiler to enable an automatic search for the best learning algorithm. To find good settings for the learner parameters within the large search space, optimizations based on evolutionary algorithms are applied. In contrast to the majority of other approaches aiming at a reduction of the average-case execution time (ACET), our goal is the minimization of the worst-case execution time (WCET) which is a key parameter for embedded systems acting as real-time systems. A careful case study on the heuristic generation for the well-known optimization *loop invariant code motion* shows the challenges and benefits of our methods.

1 Introduction

Optimizing compilers transform a program written in a source language into a semantically equivalent program in a target language. The generated code should exhibit a high performance. Since finding optimal solutions to compiler optimizations is provably hard, compiler writers are forced to use heuristics as approximate solutions. The development of heuristics for compiler optimizations is a tedious task requiring both a high amount of expertise and an extensive trial-and-error tuning. The reasons are twofold. First, heuristics often use simplified architecture models of complex systems, which do not sufficiently capture all relevant architectural features. Second, compiler optimizations are typically executed within a sequence of interfering optimizations. Since the mutual interactions are hardly predictable, compiler writers develop heuristics based on conservative assumptions. Such heuristics avoid negative effects but also prevent the exploration of the optimization potential.

[⋆] The research leading to these results has received funding from the European Community's Artist Design Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n^o 216008.

Machine learning (ML) techniques have recently raised considerable research interest in the compiler community since they can help to automatically find good optimization heuristics. Given a set of characteristics (called *static features*) about the code to be optimized, machine learning tools automatically learn a mapping from these features to heuristic parameters. For today's rapidly evolving processor market, these machine learning based (MLB) heuristics offer two advantages. First, they often outperform hand-crafted heuristics [21]. Second, they can be automatically adopted to new environments.

A central questions in the heuristic generation is that of *model selection* which covers the choice of the learning algorithm, its parameters, and the features. Over the last decades, a vast spectrum of different machine learning algorithms was developed. The learner selection for the generation of high-performance heuristics is not trivial and becomes even more complicated since most learners are equipped with numerous parameters considerably affecting the learner's behavior. The consequence is that typically one or two learners are applied using standard parameter settings [18, 15, 6]. However, this approach does not exploit the full learners' potential and possibly misses optimization opportunities.

In this paper, we systematically explore the performance of different learners and their parameters for compiler heuristic generation. We use the open-source machine learning tool RapidMiner [17]. It includes not only a large number of learning algorithms, evaluation procedures, and feature transformation operators, but also operators for self-optimization regarding, e. g., parameter settings of learning algorithms. Since the true quality of learners depends on the quality of their predictions [12], the considered learners are *directly* involved in the model selection run on real-life benchmarks. Using this approach allows to find the best learner with the highest performance increase for a particular optimization.

While machine learning was studied in the past in the context of ACET minimization, this work focuses on embedded systems acting as hard real-time systems. Besides efficiency requirements, these systems are characterized by their critical timing constraints which are expressed by the worst-case execution time. Especially for safety-critical application domains, such as automotive and avionics, the satisfaction of the WCET must be guaranteed to avoid system failure. Thus, we concentrate on an automatic generation of MLB heuristics that promise the highest WCET improvement. The main contributions of this paper are as follows:

1. For the first time, we address the well-known problem of selecting an appropriate learning algorithm for the generation of optimization heuristics [20] in a systematic way.
2. We evaluate six popular learning algorithms. The study indicates that different learners and their parameter settings significantly affect the program performance.
3. Since the search space for the learners is typically too large for an extensive search, we apply a parameter optimization based on evolutionary algorithms.
4. To demonstrate the efficiency of our approach, MLB heuristics for the well-known optimization *loop invariant code motion* are generated. In contrast to previous works, our work aims at a WCET minimization.
5. Due to the integration of a machine learning tool into the novel WCET-aware compiler framework, the exploitation of a vast range of learning techniques is established. Also, the framework can be easily adopted to generate heuristics for other compiler optimizations.

6. The presented concepts can be easily adapted to ACET optimizations. Therefore, our work can be seen as a general contribution to compiler research independent of the considered objective.

The rest of this paper is organized as follows: Section 2 gives a survey of the related work. In Section 3, an overview of the current state of machine learning employed within a compiler is provided and problems arising from the selection of learners are discussed. To overcome these problems, we propose a new methodology for an automatic selection of parametric learners in Section 4. The optimization loop invariant code motion is introduced in Section 5. A description of our experimental environment and results achieved on real-world benchmarks are given in Sections 6 and 7, respectively. Finally, Section 8 summarizes this paper and gives directions for future work.

2 Related Work

ACET Minimization by Machine Learning: The application of machine learning techniques in compiler design was mainly studied in the context of the ACET minimization. Vaswani [23] uses empirical regression models to characterize interactions between optimizations in the GCC compiler. The search for good compiler optimization sequences, also called iterative compilation, has been thoroughly studied in the past. Kulkarni [11] uses genetic algorithms to avoid an exhaustive search. In [3], a characterization of the search space is used to find good compilation sequences more efficiently. Leather [13] applies fixed sampling plans while Cavazos [5] exploits performance counters to accelerate the search. In contrast, Agakov [2] reduces the number of evaluations using machine learning approaches by focusing on promising areas of the search space.

MLB Heuristic Selection for ACET Minimization: A vast application field of machine learning in compilers is the automatic generation of optimization heuristics, known in literature as *heuristic selection*. Monsifrot [18] used a supervised classification to generate heuristics for *loop unrolling* which decide whether unrolling should be performed. This approach was extended by Stephenson [15] to find MLB heuristics that predict the best unrolling factor for a given loop. Machine learning techniques (e. g., reinforcement learning) were also studied in the context of instruction scheduling [6]. In [12], a grammar-based mechanism using genetic programming is presented that automatically extracts features for machine learned heuristics.

WCET Reduction: Typically, compiler optimizations aim at an automatic reduction of the ACET. With the growing importance of embedded systems acting as real-time systems, the worst-case execution time must be considered as a crucial objective. WCET-aware compilation is a novel research area with an increasing academic and industrial interest. Approaches in this domain rely on *feedback data*, the WCET, which is provided by a static analyzer. A sophisticated WCET analyzer, also used in this work, is *aiT* [1]. Most approaches to WCET minimization operate on assembly level and exploit memory hierarchies. For example, the authors of [4] presented an algorithm for static locking of I-caches based on a genetic algorithm while compile-time cache analysis combined with static data cache locking was presented in [24]. Other approaches exploit fast scratchpad memories (SPM) for WCET minimization. Greedy algorithms for a WCET-aware SPM allocation of data are presented in [7], while optimal approaches based on an ILP formulation are explored for data and program code in [22] and [8], respectively.

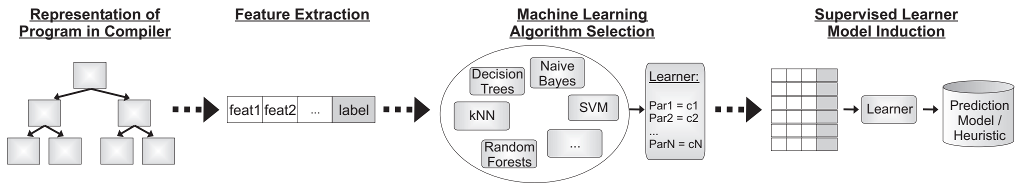

Representation of Program in Compiler
Feature Extraction
Machine Learning Algorithm Selection
Supervised Learner Model Induction
feat1
feat2
...
label
Decision Trees
Naive Bayes
kNN
SVM
Random Forests
...
Learner:
Par1 = c1
Par2 = c2
...
ParN = cN
Learner
Prediction Model / Heuristic

Fig. 1. Overview of Machine Learning Based Compiler Heuristic Generation

WCET Minimization by Machine Learning: The potential of machine learning for WCET minimization is sparsely explored within today's literature with only a few publications. Zhao [26] used a genetic algorithm for the search of standard low-level optimization sequences that aim at an effective reduction of the program WCET. In [14], supervised learning was used to infer heuristics for *function inlining*. The latter paper is most related to our current work since the objective of finding MLB heuristics for WCET reduction is pursued. However, there are also several significant differences. Most importantly, in [14] just a single supervised learner with its standard parameters was considered. Moreover, contrary to our compiler framework providing a seamless integration of a machine learning tool, the related paper used a compiler that was completely decoupled from the ML tool. Thus, the generated inlining heuristics had to be integrated into the compiler by hand. In addition, the optimizations are performed at different abstraction levels of the code. Function inlining was considered at the source code level whereas we consider *loop invariant code motion* as an optimization performed at assembly level.

3 Machine Learning in Compilers

In this section, an overview of supervised machine learning techniques in the compiler design is provided. Section 3.1 summarizes the common approach of incorporating machine learning techniques into a compiler and describes the workflow required to automatically generate a heuristic. A shortcoming of this workflow is the model selection problem which will be discussed in Section 3.2.

3.1 Current Workflow for Heuristic Generation

The heuristic generation begins with the obvious decision for which compiler optimization an improved heuristic should be generated. An overview of incorporating machine learning techniques into a compiler framework is depicted in Figure 1. For a **representation of the program** by internal compiler data structures, such as high- or low-level intermediate representations or abstract syntax trees, the developer has to decide which features best characterize the parts of the program to be optimized. The features must be transformed into a proper vector representation serving as input for the ML tool. This process is called **feature extraction**. In addition, for each feature vector a label representing the desired output, e. g., *YES/NO*, has to be determined. This phase transforms a set of benchmarks to become the *training set*. Next, a **selection of a learning algorithm and its parameters** is required. The machine learning community has developed a large portfolio of different learners over the last decades. Moreover, many learners have several user-defined parameters, leading to models with different performance. Due to the large number of possible combinations, the selection of the appropriate learning algorithm is not straightforward. Finally, the chosen classifier (learner) induces a **prediction model** representing a heuristic which can be used to predict if/how the considered optimization should be performed for unseen data.

3.2 Problem Specification: Model Selection

A key aspect of the framework shown in Figure 1 is the problem of selecting a learning algorithm and parameters such that the induced model performs best in terms of the considered objective, e. g., the WCET. Due to the complex structure of learning algorithms and the non-trivial impact of their parameters, the performance of the induced model cannot be predicted statically with sufficient precision. Rather, a heuristic must be generated and its performance must be evaluated on a set of benchmarks [12].

As a consequence, the current state for the MLB heuristic generation can be seen as a *trial-and-error* approach. The compiler writer chooses a learner, induces a prediction model and evaluates the impact of the generated heuristic on benchmarks. If the heuristic did not yield the expected performance results, the compiler writer either tunes the learner parameters or even selects another learner and repeats the evaluation hoping for better results. Obviously, repeatedly evaluating different learners manually is time-consuming, error-prone, and it is often not clear if further tuning pays off. In literature [18, 15, 6, 12], typically one or two learners are employed without a detailed reasoning why exactly these algorithms including their parameters were chosen.

The exploitation of machine learning for heuristic generation is attractive since it relieves the compiler writer from the tedious task of developing heuristics manually and it also enables an easy and efficient adoption to changes in the compiler framework or the underlying system. However, the effort is now shifted from the manual tuning of heuristics to the model selection problem of learning. Here, we propose a new framework which systematically evaluates models induced by different learners and parameter settings through integration of a compiler and an ML tool.

4 Automatic Model Selection

In this section, we describe our methodology for an automatic selection of the best model. In Section 4.1, we summarize the key characteristics of the machine learning algorithms that we consider for comparison. In Section 4.2, performance evaluation of supervised learners is discussed. As will be described in Section 4.3, this evaluation can be exploited for evolutionary parameter optimization and the final model selection. For a detailed discussion of the learning methods, see standard literature [10].

4.1 Learning Algorithms

In this paper, we consider popular learning algorithms which have been successfully applied in the past for various applications and that rely on different principles.

Decision Trees partition the examples into axes-parallel rectangles by recursively splitting the training set into sub-trees. Frequently, information gain, based on the entropy (impurity) of a node, is used as splitting criterion. Additionly, there might be stopping criteria like the maximal depth of a tree and thresholds for the minimal number of examples in a node to split it further (minimal size for split), the minimal number of examples in a leaf (minimal leaf size), the minimal information gain for splitting (minimal gain), and the number of alternative nodes considered (prepruning alternatives). Furthermore, a confidence level for post tree pruning can be specified.

Random Forests consist of several unpruned decision trees which are constructed from different bootstrap samples. The algorithm uses a randomly chosen subset of features to find the best split for each node and it is robust against overfitting. Like decision trees, random forests can still classify new examples very fast by majority voting over the predictions made by each tree in the forest. Only two parameters have to be optimized: the number of trees in the forest and the number of considered features for node splitting.

Linear Support Vector Machines (SVM) find a hyperplane which separates the examples such that those with the label $y = +1$ are in the positive half and those with the label $y = -1$ are in the negative half of the instance space. The hyperplane is determined by $\beta \cdot x + \beta_0$. The learning task is to estimate β and β_0, such that the error is minimal (i. e., the instances are placed on the correct side of the hyperplane) and that the learned model is of minimal complexity (i. e., the distance between the closest instance to the hyperplane is maximal). Those examples which are closest to the hyperplane are called support vectors. In order to allow some misclassified instances, the soft margin SVM offers a parameter C which gives a weight to the error as opposed to the complexity. Internal optimization compares all examples pairwise using a kernel function. For the linear SVM, the kernel function is the dot product $x_i \cdot x_j$.

SVMs with RBF kernel operates on not linearly separable data by including another kernel function into the SVM. The radial basis function (RBF) covers areas of instances by a Gaussian distribution: $K_{RBF}(x_i, x_j) = exp(\gamma(x_i, x_j)^2)$. Hence, the parameter of the Gaussian's width, γ, is decisive: for a low γ, almost every example is covered by its own RBF region, for a large γ, interesting regions cover a set of examples.

k-Nearest Neighbor (kNN) stores all examples and classifies a new input by looking at k most similar examples. The majority class of these k examples becomes the predicted class. If k is too small, the error is reduced, but the prediction becomes biased, e. g., by outliers. If k is too large, the error might also become large. Thus, the setting of the appropriate k is crucial for the learner's performance.

Naive Bayes predicts for an example x the class y such that the likelihood $P(y|x)$ is maximal. According to Bayes' theorem, it suffices to maximize the probability $P(X|y_i)P(y_i)$, since the a priori probability of the labels in Y (e. g., $P(y_i = YES)$ or $P(y_i = NO)$) are the same for all training examples. Implicitly, Naive Bayes assumes the independence of all example's features. Due to its simple calculation, Naive Bayes is a very fast algorithm and has typically no parameters for configuration.

4.2 Performance Evaluation

There are different metrics for performance evaluation of learners. Which metric to choose, depends on the requirements imposed by the exploiting system. The standard performance measurement of learning algorithms is accuracy. It is calculated on the basis of the test set. Examples $x_{n+1}, x_{n+2}, ..., x_{n+m}$ are handed over to the learned function f, delivering $\hat{y} = f(x)$. Then, the known true value y is compared to the predicted $\hat{y}$. Drawing training and test set under the same distribution D leads to an estimate of the true performance of the learner indicating, e. g., how often $\hat{y} = y$. The estimation is determined by generating a set of examples and splitting it into training and test set. This is done in *cross validation*: N-fold cross validation randomly partitions an example set into N sets, uses $N-1$ sets for training and the remaining set for testing. The estimated performance of a learner is the average of the measurements of the N training and test cycles.

In case of compiler optimizations, program run time is crucial. For embedded systems acting as real-time systems, the main goal is to find a learner that yields highest WCET reduction. The WCET of a program is the longest execution time that can ever occur. Since the input to the program leading to the worst-case behavior is often not known and an exhaustive testing of all inputs is not feasible, measurements are not suitable for a WCET determination. To obtain the WCET, formal methods are used instead. The control flow graph of the program is statically analyzed taking addition information from the user, like loop iteration counts, into account. To cover all possible input data, abstraction from concrete values is used. Thus, the determination of the actual WCET is lifted to the derivation of an upper bound on the execution time of the program. In this paper, we use the term WCET as a synonym for a safe WCET estimation of the actual WCET.

The performance evaluation of a learner based on the accuracy is not appropriate since it does not allow to draw conclusions about the program's WCET.

Example 1: *Assume that a learning model acting as a compiler heuristic has to take three optimization decisions. The costs (impact on program's WCET in cycles) for the correct prediction/misprediction of the decisions are:* $Cost_A = 1/-1$, $Cost_B = 1/-1$, *and* $Cost_C = 10/-10$. *Predicting A and B correctly, but not C, results in an accuracy of* $66,\overline{6}\%$ *and a negative impact on the WCET of* $(1+1-10=)\ -8$ *cycles, while predicting just C correctly yields a worse accuracy of* $33,\overline{3}\%$ *but a positive impact on the WCET of* $(-1-1+10=)\ 8$ *cycles.*

Due to this missing correlation between the accuracy and the program (worst-case) execution time, learners should be evaluated by directly measuring the program performance but not their accuracy.

Moreover, the classical N-fold cross validation has to be applied in a modified fashion for the performance evaluation of learners used as optimization heuristics. For each of the N benchmarks of the example set, all examples belonging to one benchmark are excluded (test set), a ML model using the remaining examples (training set) is learned and this model is finally applied by a compiler to evaluate its impact on the WCET of the excluded benchmark. In more detail, the compiler computes the WCET $WCET_{MLB}$ for this benchmark using the new MLB heuristic and compares this value against a reference value $WCET_{ref}$. If $\Delta WCET_n < 1$, with $\Delta WCET_n = WCET_{MLB}/WCET_{ref}$, then the MLB heuristic was successful. The final performance is determined by performing the cross validation N times and computing the average relative WCET: $performance = \sum_{i=1}^{N} \Delta WCET_n/N$. Using this *benchmark-wise* cross validation is a common approach to estimate the *generalization* ability of a learning algorithm, i. e., by applying the models to *unseen* benchmarks it can be inferred how well new examples will perform using this model.

4.3 Parameter Optimization

Exhaustively searching over all combinations of user-defineable classifier parameters is not feasible. We therefore apply an evolutionary strategy [16]. Our approach is depicted in Figure 2. Each individual pn in a population of size (*pop_size*) represents a combination of parameter values, e. g., C and γ in case of the *SVM with RBF kernel.* In the beginning, the parameters of each individual are initialized randomly. To create a new

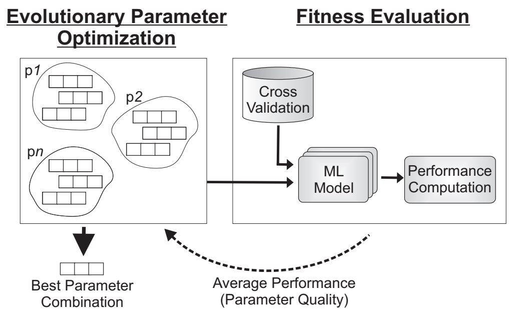

Evolutionary Parameter Optimization
Fitness Evaluation
p1
p2
pn
Cross Validation
ML Model
Performance Computation
Best Parameter Combination
Average Performance (Parameter Quality)

Fig. 2. Evolutionary Parameter Optimization

generation, a fraction of the individuals repeatedly takes part in a tournament selection which chooses fittest individuals (as parents) as long as *pop_size* individuals are selected. In a crossover step, individuals mate with a specified probability (*crossover_prob*). They produce children that contain the exchanged parameter values of their parents. These children are added to the current population. Then, all individuals are cloned and the clones are mutated by adding values from a Gaussian distribution to all parameters. The fitness of each individual is evaluated by a cross-validation which is based e. g., on the accuracy or, as in our case, on the reduction of the program's WCET. More accurately, for each individual a machine learning model is induced based on $N - 1$ benchmarks and evaluated for the left-out benchmark. This performance computation is repeated N times and its average value represents the quality of a parameter combination. The whole process maintains the best individuals (*elitist selection*) and terminates if either a specified maximum number of generations (*max_gen*) is reached or there was no improvement over *imp* generations.

5 Case Study: Loop Invariant Code Motion

Loop invariant code motion (LICM) is a well-known ACET optimization. It recognizes computations within a loop that produce the same result each time the loop is executed. These computations are called *loop invariant code* and can be moved outside the loop body without changing the program semantics [19].

Definition 1. *An instruction i is said to be* ***loop invariant*** *iff: (a) its operands are constants, or (b) all instructions that define the operands of instruction i are outside the loop, or (c) all instructions that define the operands of instruction i are themselves loop invariant.*

LICM can be applied at the source code level to expressions, or at the assembly level, in particular to addressing computations that access elements of arrays. The positive effects are a reduced execution frequency of the moved loop-invariant code. Another positive effect of the optimization is that it might shorten the live ranges of variables leading to a decreased register pressure.

Besides these positive effects on the code, LICM may also degrade performance. This is mainly due to two reasons. First, the newly created variables to store the loop-invariant results outside the loop increase the register pressure in the loops since their live range spans across the entire loop nest. This might possibly lead to additional register spill code. This is an issue especially relevant for embedded systems with a

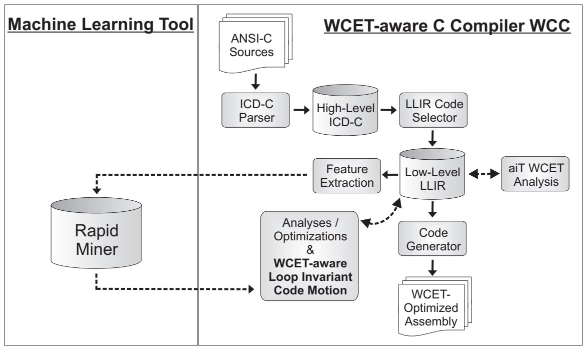


Fig. 3. Overview of Compiler Framework

small register file. For example, the TriCore processor, that is also used in this work, has 8 data and 8 address registers serving as general purpose registers. The remaining 8 data and address registers have special purposes, like storage of function arguments or return addresses, and are thus only partially exploited by the register allocation. Second, moving the loop invariant code might lengthen other paths of the control flow graph when the invariants are moved from a less executed to a more frequently executed path, e. g., moving instructions above a loop's *zero-trip test*.

Issues like the impact on the register pressure emphasize the dilemma compiler writers are faced with during the development of good heuristics. Performing loop invariant code motion has conflicting goals and it can not be easily predicted if this transformation is beneficial. LICM heuristics are also missing in standard compiler literature [19]. In addition, most compilers do not model the complex interactions between different parts of the code and the loop invariants, but perform LICM whenever invariants are found without using any heuristics that might avoid the adverse effects.

We tackle the difficult task of finding heuristics for loop invariant code motion using machine learning. The goal is to find a heuristic that exploits the positive effects of LICM on the one hand and prohibits the transformation for adverse situations on the other hand. In contrast to related works dealing with optimizations for which different heuristics are well-studied, e. g., loop unrolling, we have no hints which strategies for the LICM heuristic might be promising.

6 Experimental Environment

To demonstrate the practical use of our approach, experiments on a large number of different benchmarks were conducted. The 39 benchmarks come from the test suites DSPstone, MediaBench, MiBench, MRTC WCET Benchmark Suite, UTDSP and NetBench. On the one hand, the benchmarks are used to construct the data set for machine learning (cf. Section 6.2), which serves as training data for the LICM heuristic generation. On the other hand, they are used in the cross validation phase to evaluate the performance of the heuristic for WCET minimization. The training set based on these benchmarks comprises 3491 examples and its construction took about 50 hours on two Intel Xeon 2.13GHz quad cores. However, please note that the data set construction has to be performed once off-line.

All experiments were performed in the WCET-aware C compiler WCC [9] for the Infineon TriCore TC1796 processor. The framework including the integration of the machine learning tool RapidMiner is depicted in Figure 3. The compiler shown on the right-hand side of the figure is provided with C source files. After parsing the C code, it is translated into the high-level intermediate representation *ICD-C*. At this level, standard compiler analyses and source code ACET optimizations (not shown in the figure) can be applied. Next, the code selector translates the code into the low-level intermediate representation *LLIR*. At this abstraction level, again different analyses and optimizations are available. In total, the compiler features 43 different optimizations which are activated in the highest optimization level $O3$. Loop invariant code motion, which is a low-level optimization within WCC, is performed as one of the last optimizations in the optimization chain. Since it is executed before register allocation, LICM operates on low-level code which does not contain physical registers but temporary variables (aka. *virtual registers*). For benchmarking, $O3$ is enabled, thus our WCET-aware LICM operates on highly optimized code.

The key feature of the WCC compiler is the tight integration of the static WCET analyzer aiT into the compiler backend. This way, WCET timing data is available in the compiler backend and can be exploited for analyses and optimizations.

6.1 Available Features

The presented compiler framework for the automatic selection of machine learning models is generic, i. e., it can be exploited to generate heuristics for a large number of low-level optimizations without any major adaption. To enable this option, a large set of features extracted from the compiler must be provided. These features must be chosen such that they cover a wide range of various characteristics of the program. Our *feature extractor* (cf. Figure 3) generates 73 features in total which describe characteristics of single instructions, basic blocks, loops, or functions depending on which low-level construct is passed to the feature extractor. The features can be classified as follows (given some examples):

1. **Structural features**: Type of instruction (arithmetic, load/store, jumps, floating point, etc.), size of given construct, number of block successors/predecessors, number of operands in given construct
2. **Liveness analysis related**: Liveness information (*live-in* and *live-out*) of instruction, number of *defs* and *uses* in instructions/blocks, information about register live times (for register pressure estimation)
3. **Loop features**: Loop nest levels, loop iteration counts
4. **Misc**: Length of critical path in loop, outcome of static branch prediction for jump instruction

This set of features is variable, i. e., depending on the application all features or just a subset can be used. The feature extractor was designed in a flexible way such that new features can be easily added. For learning algorithms that can only handle numerical values, nominal features are first transformed into discrete numerical values and then normalized by a linear transformation into $[0, 1]$.

6.2 Construction of Training Set for WCET-aware LICM

For the loop invariant code motion, 39 benchmarks are involved in the training set construction. We used WCC's feature analyzer with the full set of all 73 features. Each example of the training set was created by analyzing each loop invariant instruction i_{inv} separately. To do so, corresponding features for i_{inv} as well as for the basic block b_{pred}, to which i_{inv} is moved, were extracted. The label was determined by estimating

Table 1. Learner-specific parameters, the explored value ranges by the evolutionary search, and the best found parameter combinations yielding highest WCET reduction.

Parameter	Range	Best	Parameter	Range	Best
Decision Trees			**SVM with RBF kernel**		
max. depth	[1;20]	16	C	[0;10,000]	2405.15
min. split size	[4;100]	19	γ	[0;74]	30.08
min. leaf size	[2;100]	31	**Linear SVM**		
min. gain	[0;0.03]	0.014	C	[0;10,000]	616.11
prepr. altern.	[3;10]	4	**kNN**		
confidence	[0.1;0.5]	0.476	k	[3;100]	11
Random Forests			**Naive Bayes**		
no of trees	[1;100]	7	*no configurable parameters*		
features	[1;73]	39			

the WCET of *region reg* before and after LICM. The region *reg* is defined either as the loop to which b_{pred} belongs to or, if i_{inv} was moved completely outside a loop, *reg* represents the function were i_{inv} is located. Using outer loops for *reg* instead of the entire function makes the label extraction more reliable since it captures the effects of LICM more precisely. A decreased WCET after LICM means that the transformation is beneficial (label *YES*) for i_{inv} in its current context (b_{pred}). If the WCET does not change, also the label *YES* is used to perform such code motion which possibly enable optimization potential for subsequent LICM candidates. If a WCET increase due to adverse LICM effects was identified, the feature vector is labeled with *NO* to indicate that the code motion should be avoided for similar cases. For the next example, i_{inv} is kept in its new position and the next loop invariant instruction is considered.

6.3 Evolutionary Parameter Optimization for WCET-aware LICM

After the construction of the training set for the loop invariant code motion using exclusively the WCC compiler, the evolutionary parameter optimization for the selection of the best ML model requires a communication with RapidMiner.

The parameter optimization is performed for each of the six considered machine learners to find the model yielding the highest WCET reduction. The evolutionary algorithm generates different valid parameter combinations which are employed for the fitness evaluation. For our experiments we used the following parameters for the evolutionary algorithm: population size *pop_size*=20, number of generations *max_gen*=5, tournament selection performed on 30% of population size with a crossover probability *crossover_prob*=90%, and termination if no improvement for *imp*=2 generations was observed (cf. Section 4.3).

The fitness evaluation is based on the *benchmark-wise* cross validation (cf. Section 4.2). For a given combination of parameters determined by the evolutionary algorithm, a model based on the training set of benchmarks is learned and validated against the benchmark from the test set, i. e., WCC computes the WCET $WCET_{MLB}$ for this benchmarks using $O3$ and the LICM heuristic based on the current model. This step is repeated for each of the N=39 benchmarks. To determine the quality of the model, $WCET_{MLB}$ is compared against a reference value $WCET_{ref}$ representing the WCET for this benchmark using $O3$ and disabled LICM. Finally, the fitness value which represents the quality of a given parameter combination is computed by:

Table 2. Performance results for different parameter combinations as found by evolutionary search.

Learner	Best	Worst	Avg.	Acc.
Decision Tree	96.17%	99.78%	97.42%	63.16%
Random Forests	96.60%	98.96%	97.69%	60.43%
Linear SVM	98.24%	98.62%	98.34%	53.50%
SVM with RBF kernel	**95.36%**	98.80%	97.12%	57.78%
kNN	97.32%	98.94%	97.98%	**67.48%**
Naive Bayes	98.17%	98.17%	98.17%	54.31%

$fitness = \sum_{i=1}^{N} \Delta WCET_n / N$. Obviously, this is a minimization problem, with smaller $fitness$ being better.

The output of the evolutionary parameter search is the machine learning model using the detected parameter settings that led to the highest WCET reduction. Our framework automatically performs the parameter optimization for each considered learner to find the model that exhibits the overall best WCET improvement. This model (heuristic) is finally integrated into the compiler. For future use of the novel WCET-aware LICM, the WCC compiler performs a feature extraction and consults RapidMiner to retrieve a prediction whether the considered loop invariant instruction promises a WCET reduction. The communication between WCC and RapidMiner is established in an efficient way, thus the additional overhead is marginal.

7 Results

In a first phase, the machine learning model selection was performed to find the best learner. Table 1 gives an overview of the considered learners, their parameters, and the explored parameter values by the evolutionary search (column *Range*). Please note that Naive Bayes does not provide any parameters to be optimized. However, the algorithm was considered due to its popularity and its specific functionality.

Table 2 summarizes the results of the evolutionary parameter optimizations for the six considered learners. The results in the second, third, and fourth column represent the performance values, i. e., the averaged relative WCET results obtained during the benchmark-wise cross validation (cf. Section 6.3) when comparing the WCET using the MLB heuristic against the code compiled with $O3$ and without LICM. In more detail, the second column (*Best*) represents the highest improvement of the WCET observed during the evolutionary search of each learner. These values were achieved using the parameter combinations shown in the third column of Table 1. For example, 95.36% for *SVM with RBF kernel* means that the WCET was reduced on average by 4.64%. The third and fourth column (*Worst*, *Avg.*) of Table 2 depict the worst and average WCET reduction (over all runs) found by the evolutionary search. Finally, the last column (*Acc.*) describes the classification accuracy that was computed for the parameter combination that lead to the best WCET reduction shown in the second column. The bold numbers point out the best results observed for all learners.

Three main conclusions can be drawn from this table. **First**, it can be seen that the WCET improvements significantly vary between the learners. For the considered learners and their best parameters, the relative WCET for the 39 benchmarks varies for the best parameters between 95.36% for *SVM with RBF kernel* as best model and 98.24% for the *Linear SVM*. Thus, a comparison of various learners is required

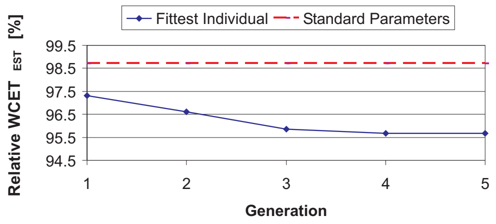


Fig. 4. Progress of Evolutionary Parameter Optimization

for the determination of the best model. Even though the difference of 2.9% might seem small, it should be taken into account that standard LICM achieves on average a WCET reduction of merely 0.6% (as will be shown later). Thus, the variation between the learners can be considered substantial and for other compiler optimizations with stronger effects on the program performance even considerably larger differences can be expected. Note also that the variance of 2.9% can not be referred to noise since statically computed WCET estimations are deterministic as the analysis always assumes the same worst-case run-time environment. **Second**, a comparison between the second and third column in Table 2 emphasizes the importance of a parameter optimization. For example, the choice of parameter settings for the learner *Decision Tree* generates LICM heuristics for which the relative WCET ranges between 96.17% and 99.78%, i. e., selecting inappropriate parameters may *waste* up to 3.61% on average of the optimization potential w. r. t. the WCET reduction. **Third**, a comparison between the WCET performance in the second column and the accuracy in the last column indicates that there is no direct correlation between these two performance metrics (cf. Section 4.2). For example, the highest accuracy of 67.48% was achieved for the *kNN* learner, while the average WCET reduction of 2.68% is poor compared to the other learners. Thus, finding the best model can be only accomplished when the model is directly evaluated against the considered objectives, in our case the WCET.

Figure 4 depicts the progress of the evolutionary parameter optimizations over 5 generations for the best learner (*SVM with RBF kernel*). The plot depicts the fittest individual (parameter combination) in each generation. As can be seen, the performance of the fittest individual is successively improved in the first four generations before no better parameters can be found in the last generation. This monotonically decreasing curve suggests that the evolutionary parameter optimization is the right choice for the search of good parameter settings in a large space. Also, a comparison between the performance of 98.71% for the standard SVM parameter settings ($C = 0$, $\gamma = 1$) and the performance of 95.36% for the best parameter combination found by the evolutionary search emphasizes the benefits of this approach. In order to evaluate the effectiveness of our machine learning based LICM heuristic, we measured the impact of our MLB heuristics for LICM on the WCET estimates ($WCET_{EST}$) of the considered 39 benchmarks. Figure 5 shows a comparison between the standard ACET LICM (*Standard-LICM*) and our optimization (*MLB WCET-LICM*) using the best heuristic generated by the *SVM with RBF kernel* learner. The reference mark of 100% corresponds to the WCET estimates for $O3$ with disabled LICM. Due to the challenges for the manual generation of an appropriate heuristic (cf. Section 5), the standard approach for LICM in many compilers is the application of the code transformation whenever possible. The light bars representing the MLB-LICM show WCET estimates computed during the benchmark-wise cross validation. By learning a model

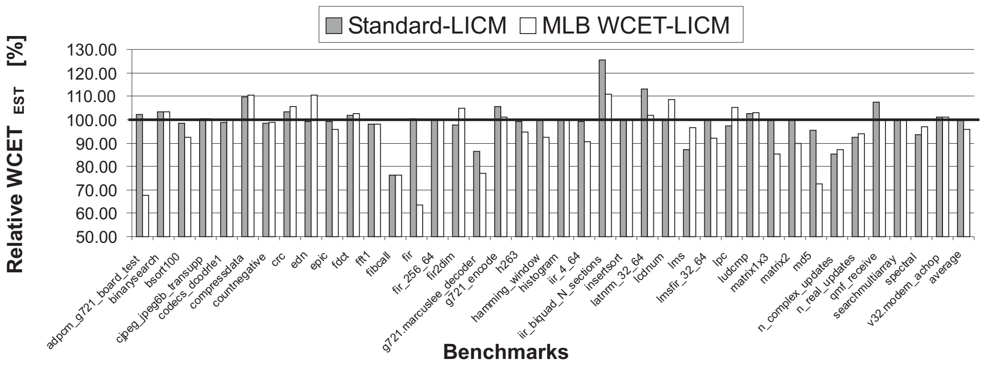


Fig. 5. Relative WCET Estimates for Standard and MLB LICM

and validating it on the excluded benchmark, the light bars indicate how good the heuristic performs on *unseen* data. As can be seen in the figure, in most cases the new MLB-LICM outperforms the standard LICM optimization, with up to 36.98% for the *fir* benchmark from the MRTC WCET Benchmark Suite. On average, the standard LICM achieves a WCET reduction of merely 0.56%, while our MLB-LICM reduces the WCET by 4.34%, as already shown in Table 2.

Most of the time for the evolutionary search was consumed by the WCET analyses. For one run of the benchmark-wise cross validation, i. e., inducing 39 models and using them for the WCET estimation of each benchmark in the test set, about 50 minutes on a single Intel Xeon 2.13GHz core of a system with 8GB RAM were required. Depending on the development of the evolutionary search, the maximal run time of 146 hours was observed for the evaluation of the learner *Random Forests*.

8 Conclusions and Future Work

Recent work has shown that machine learning can be exploited for the automatic generation of high-performance and easily adaptable compiler optimization heuristics. A central questions in this domain is that of model selection, i. e., which learners and their respective parameters should be used. This paper is the first one to address this well-known problem in a systematic way. We explore the potential of six popular learning algorithms using an evolutionary parameter optimization. In a case study, we exploit our novel compiler framework for the generation of heuristics for loop invariant code motion aiming at a WCET reduction. In contrast to standard LICM yielding an average WCET reduction of 0.56% on 39 real-life benchmarks, our new heuristics achieve a WCET reduction of 4.34% on average.

In the future, we intend to integrate further learning algorithms into our framework to explore their potential. Also, these algorithms require further evaluation to figure out why some learners work well or why not. Another important issue for future work is the integration of further compiler optimizations, e. g., register allocation, to study the generality of our methodology. The investigation of optimizations with bigger pay-offs can possibly better highlight the potential of our system. Moreover, we want to tackle another important issue of the model selection problem, the *feature selection*, which finds promising features from the set of extracted features. In [25] it has been shown that using an appropriate representation for training is beneficial for every learner and that particular learners show different preferences for the representation of features.

References

1. AbsInt Angewandte Informatik GmbH: Worst-Case Execution Time Analyzer aiT for TriCore. `http://www.absint.com/ait` (2009)
2. Agakov, F., Bonilla, E., Cavazos, J., et al.: Using Machine Learning to Focus Iterative Optimization. In: Proc. of CGO. New York, USA (2006)
3. Almagor, L., Cooper, K.D., Grosul, A., et al.: Finding Effective Compilation Sequences. In: Proc. of the LCTES. Ottawa, Canada (2004)
4. Campoy, A., Puaut, I., Ivars, A., Mataix, J.: Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In: Proc. of ECRTS (2005)
5. Cavazos, J., Fursin, G., Agakov, F., et al.: Rapidly Selecting Good Compiler Optimizations using Performance Counters. In: Proc. of CGO. San Jose, USA (2007)
6. Cavazos, J., Moss, J.E.B.: Inducing Heuristics to Decide Whether to Schedule. SIGPLAN Not. 39(6) (2004)
7. Deverge, J.F., Puaut, I.: WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In: Proc. of ECRTS. Pisa, Italy (2007)
8. Falk, H., Kleinsorge, J.C.: Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: Proc. of DAC. San Francisco, USA (2009)
9. Falk, H., Lokuciejewski, P., Theiling, H.: Design of a WCET-Aware C Compiler. In: Proc. of ESTIMEDIA (2006)
10. Hastie, T., Tibshirani, R., Friedman, R.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics, Springer, Berlin (2008)
11. Kulkarni, P.A., Hines, S.R., Whalley, D.B., et al.: Fast and Efficient Searches for Effective Optimization-phase Sequences. ACM TACO 2(2) (2005)
12. Leather, H., Bonilla, E., O'Boyle, M.: Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In: Proc. of CGO. Seattle, USA (2009)
13. Leather, H., O'Boyle, M., Worton, B.: Raced Profiles: Efficient Selection of Competing Compiler Optimizations. In: Proc. of LCTES. Dublin, Ireland (2009)
14. Lokuciejewski, P., Gedikli, F., Marwedel, P., Morik, K.: Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. In: Proc. of SMART. Paphos, Cyprus (2009)
15. Mark Stephenson and Saman Amarasinghe: Predicting Unroll Factors Using Supervised Classification. In: Proc. of CGO. San Jose, USA (2005)
16. Mierswa, I.: Non-Convex and Multi-Objective Optimization in Data Mining. Ph.D. thesis, Technische Universität Dortmund (2008)
17. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: Rapid Prototyping for Complex Data Mining Tasks. In: Proc. of KDD. Philadelphia, USA (2006)
18. Monsifrot, A., Bodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Proc. of AIMSA. Varna, Bulgaria (2002)
19. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, USA (1997)
20. Srikant, Y.N., Shankar, P.: The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition. CRC Press, Inc., Boca Raton, FL, USA (2007)
21. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta Optimization: Improving Compiler Heuristics with Machine Learning. SIGPLAN Not. 38(5) (2003)
22. Suhendra, V., Mitra, T., Roychoudhury, A., Chen, T.: WCET Centric Data Allocation to Scratchpad Memory. In: Proc. of RTSS. Miami, USA (2005)
23. Vaswani, K., Thazhuthaveetil, M.J., Srikant, et al.: Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In: Proc. of CGO. San Jose, USA (2007)
24. Vera, X., Lisper, B., Xue, J.: Data Cache Locking for Higher Program Predictability. In: Proc. of SIGMETRICS (2003)
25. Wurst, M., Morik, K.: Distributed Feature Extraction in a P2P Setting - A Case Study. Future Generation Computer Systems, Special Issue on Data Mining 23(1) (2007)
26. Zhao, W., Kulkarni, P., Whalley, D., et al.: Tuning the WCET of Embedded Applications. In: Proc. of RTAS. Toronto, Canada (2004)

Application Heartbeats

A Generic Interface for Expressing Performance Goals and Progress in Self-Tuning Systems

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal

Massachusetts Institute of Technology
{hank,eastepjm,santa,jasonm,agarwal}@csail.mit.edu

Abstract. Self-tuning, self-aware, or adaptive computing has been proposed as one method to help application programmers confront the growing complexity of multicore software development. Such systems have been proposed for architectures, compilers, and operating systems to ease the application programmer's burden by providing services that automatically customize to meet the needs of the application. However, these systems often rely on ad hoc methods for understanding and monitoring an application and thus struggle to incorporate the true performance goals of the applications they are designed to support. This paper presents the Application Heartbeats API which addresses the need to provide a standardized interface for applications to communicate with supportive adaptive systems. The Application Heartbeats framework provides a simple, standard programming interface that applications can use to indicate their performance and which system software can use to query that performance. Several experiments demonstrate the simplicity and efficacy of the Application Heartbeat approach.

1 Introduction

As multicore processors become increasingly prevalent, system complexities are skyrocketing. It is no longer practical for an average programmer to balance all of the system constraints and produce an application that performs well on a variety of machines, in a variety of situations. One approach to simplifying the programmer's task is the use of *self-tuning*, or *adpative* hardware and software. Self-tuning systems take some of the burden off of programmers by monitoring themselves and optimizing or adapting as necessary to meet their goals.

As described in [1], adaptive systems must be able to *monitor* their environment as well as *detect* significant changes. Despite this need, there is no standardized, general approach for applications and systems to measure how well they are meeting their goals. Existing approaches are largely ad hoc: either handcrafted for a particular system or reliant on architecture-specific performance counters. Not only are these approaches fragile and unlikely to be portable to other systems, they frequently do not capture the actual goal of the application. For example, measuring the number of instructions executed in a period of time does not tell you whether those instructions were doing useful work or spinning on a lock; reliance on CPU utilization or cache miss rates has similar drawbacks. The problem with mechanisms such as performance counters is that they attempt to infer high-level application performance from low-level machine performance. What is needed is a portable, universal method of monitoring an application's actual progress towards its goals.

This paper introduces a software framework called *Application Heartbeats* (or just *Heartbeats* for short) that provides a simple, standardized way for applications to monitor their performance and make that information available to external observers. The framework allows programmers to express their application's goals and the progress that it is making using a simple API. As shown in Figure 1, this progress can then be observed by either the application itself or an external system (such as the OS or another application) so that the application or system can tune its behavior to make sure the goals are met. Application-specific goals may include throughput, power, latency, quality-of-service, or combinations thereof. Application Heartbeats can also help provide fault tolerance by providing information that can be used to predict or quickly detect failures.

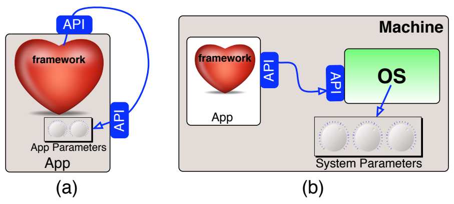


Fig. 1. (a) Self-optimizing application using the Application Heartbeats framework. (b) Optimization of machine parameters by an external observer.

This paper makes the following contributions:

1. A simple, standardized Heartbeats API for specifying and monitoring application-specific performance metrics.
2. Examples of ways that the framework can be used, both within an application and by external services, to develop self-optimizing applications. Experimental results demonstrate the effectiveness of the Application Heartbeats approach.

The rest of this paper is organized as follows. Section 2 identifies key system components that will benefit from the Application Heartbeats framework. Section 3 describes the Application Heartbeats API in greater detail. Section 4 presents our experimental results. Section 5 compares Application Heartbeats to related work. Finally, Section 6 concludes.

2 Heartbeat Usage in System Software and Hardware

The Application Heartbeats framework is a simple end-to-end feedback mechanism that can potentially have a large impact on the design of adaptive and self-tuning computer systems. This section explores ideas for novel computer architectures, operating systems, and compilers which may exploit the Heartbeats framework as a feedback mechanism to enable self-tuning.

Self-tuning Architectures. We envision a multicore microarchitecture that can adapt properties of its TLB, L1 cache, and L2 cache structures such as associativity, size, replacement policy, etc. to improve performance or minimize energy for a given performance level. We envision a multicore microarchitecture that can adapt its execution pipeline in a way similar to the heterogeneous multicores proposed in [2]. Lastly, we envision a multicore microarchitecture where decisions about dynamic frequency and voltage scaling are driven by the performance measurements and target heart rate mechanisms of the Heartbeats framework. [3, 4] are examples of frequency and voltage scaling to reduce power. Driving these new microarchitectures with an end-to-end mechanism such as a heartbeat, as opposed to indicators such as cache misses or utilization, ensures that microarchitectural optimizations focus on aspects of execution most important to meeting application goals.

Organic Operating Systems. Heartbeats provides a framework for novel operating systems with *organic* features such as self-healing and intelligent resource management. Heartbeats allow an OS to determine when applications fail and quickly restart them. Heartbeats provide the feedback necessary to make decisions about how many cores to allocate to an application. An organic OS would be able to automatically and dynamically adjust the number of cores an application uses based on an individual application's changing needs as well as the needs of other applications competing for resources. The OS would adjust the number of cores and observe the effect on the application's heart rate. An organic OS could also take advantage of the Heartbeats framework in the scheduler. Schedulers could be designed to run an application for a specific number of heartbeats (implying a variable amount of time) instead of a fixed time quanta. Schedulers could be designed that prioritize time allocation based on the target heart rate requirements of different applications. Locking mechanisms provided through the OS can be improved using Heartbeats. For example, Smartlocks [5], an adaptive locking framework, uses the Heartbeats API to obtain a direct measure of program performance and adapt locking and scheduling policies to meet the performance goals of the application. Heartbeats provide Smartlocks with a direct measure of application performance as opposed to using statistics gathered within the lock library such as lock contention.

Adaptive Compilation. Adaptive [6] and dynamic [7] compilation techniques have emerged to increase portability and address some challenges that cannot be met by traditional static compilation. Heartbeats could be used to improve the design of these compilers in several ways. First, by providing a standard interface the Heartbeat API allows one program to work with multiple compilers without source-level code changes. Second, by providing a mechanism

Table 1. Heartbeat API functions

Function Name	Arguments	Description
HB_initialize	window[int], local[bool]	Initialize the Heartbeat runtime system and specify how many heartbeats will be used to calculate the default average heart rate
HB_heartbeat	tag[int], local[bool]	Generate a heartbeat to indicate progress
HB_current_rate	window[int], local[bool]	Returns the average heart rate calculated from the last *window* heartbeats
HB_set_target_rate	min[int], max[int], local[bool]	Called by the application to indicate to an external observer the average heart rate it wants to maintain
HB_get_target_min	local[bool]	Called by the application or an external observer to retrieve the minimum target heart rate set by HB_set_target_rate
HB_get_target_max	local[bool]	Called by the application or an external observer to retrieve the maximum target heart rate set by HB_set_target_rate
HB_get_history	n[int], local[bool]	Returns the timestamp, tag, and thread ID for the last *n* heartbeats

for specifying program goals Heartbeats allow dynamic compilers to know when to stop optimizing, allowing the system to save energy by avoiding unnecessary work. Third, the Heartbeat API allows application code to specify the regions of the application where performance is critical, again allowing the system to avoid unnecessary optimization. As an example, the SpeedPress compiler inserts a runtime system, called SpeedGuard, into an application which uses Heartbeats to detect performance changes. The SpeedGuard runtime can then trade quality-of-service for performance in order to maintain the real-time goals of a system in the face of faults like core-failures and clock frequency changes [8].

3 Heartbeats API

Since heartbeats are meant to reduce programmer effort, they must be easy to insert into applications. The basic Heartbeat API consists of only a few functions (shown in Table 1) that can be called from applications or system software. To maintain a simple, conventional programming style, the Heartbeats API uses only standard function calls and does not rely on complex mechanisms such as OS callbacks.

The key function in the Heartbeat API is *HB_heartbeat*. Calls to *HB_heartbeat* are inserted into the application code at significant points to register the application's progress. Each time *HB_heartbeat* is called, a heartbeat event is logged. Each heartbeat generated is automatically stamped with the current time and thread ID of the caller. In addition, the user may specify a tag that can be used to provide additional information. For example, a video application may wish to indicate the type of frame (I, B or P) to which the heartbeat corresponds. Tags can also be used as sequence numbers in situations where some heartbeats may be dropped or reordered. Using the *local* flag, the user can specify whether the heartbeat should be counted as a local (per-thread) heartbeat or as a global (per-application) heartbeat.

We anticipate that many applications will generate heartbeats in a regular pattern. For example, the video encoders may generate a heartbeat for every frame of video. For these applications, it is likely that the key metric will be the average frequency of heartbeats or *heart rate*. The *HB_current_rate* function returns the average heart rate for the most recent heartbeats.

Different applications and observers may be concerned with either long- or short-term trends. Therefore, it should be possible to specify the number of heartbeats (or *window*) used to calculate the moving average. Regarding window size there may be some tension between the application registering the heartbeats and the system service reading the heartbeats. We assume that the application knows which window size is most appropriate for the computation it is performing; however, the system service responding to this information may want to override this window if it is trying to make adjustments on a different granularity. Therefore, the API allows the application to set the window size and this size is the default used whenever an external system requests the current heartrate. An additional API call allows system software to override the window size.

Applications with real-time deadlines or performance goals will generally have a target heart rate that they wish to maintain. For example, if a heartbeat is produced at the completion of a task, then this corresponds to completing a certain number of tasks per second. Some applications will observe their own heartbeats and take corrective action if they are not meeting their goals. However, some actions (such as adjusting scheduler priorities or allocated resources) may require help from an external source such as the operating system. In these situations, it is helpful for the application to communicate its goals to an external observer. For this, we provide the *HB_set_target_rate* function which allows the application to specify a target heart rate range. The external observer can then take steps on its own if it sees that the application is not meeting (or is exceeding) its goals.

When more in-depth analysis of heartbeats are required, the *HB_get_history* function can be used to get a complete log of recent heartbeats. It returns an array of the last n heartbeats in the order that they were produced. This allows the user to examine intervals between individual heartbeats or filter heartbeats according to their tags. Most systems will probably place an upper limit on the value of n to simplify bookkeeping and prevent excessive memory usage. This provides the option to efficiently store heartbeats in a circular buffer. When the buffer fills, old heartbeats are simply dropped.

Multithreaded applications may require both per-thread and global heartbeats. For example, if different threads are working on independent objects, they should use separate heartbeats so that the system can optimize them independently. If multiple threads are working together on a single object, they would likely share a global heartbeat. Thus, each thread should have its own private heartbeat history buffer and each application should have a single shared history buffer. Threads may read and write to their own buffer and the global buffer but not the other threads' buffers.

Some systems may contain hardware that can automatically adapt using heartbeat information. For example, a processor core could automatically adjust its own frequency to maintain a desired heart rate in the application. Therefore, it must be possible for hardware to directly read from the heartbeat buffers. In this case the hardware must be designed to manipulate the buffers' data structures just as software would. To facilitate this, a standard must be established specifying the components and layout of the heartbeat data structures in memory. Hardware within a core should be able to access the private heartbeats for any threads running on that core as well as the global heartbeats for an application. We leave the establishment of this standard and the design of hardware that uses it to future work.

4 Experimental Results

This section presents several examples illustrating the use of the Heartbeats framework. First, a brief study is presented using Heartbeats to instrument the PARSEC benchmark suite [9]. Next, an adaptive H.264 encoder is developed to demonstrate how an application can use the Heartbeats framework to modify its own behavior. Then an adaptive scheduler is described to illustrate how an external service can use Heartbeats to respond directly to the needs of a running application. Finally, the adaptive H.264 encoder is used to show how Heartbeats can help build fault-tolerant applications. All results discussed in this section were collected on an Intel x86 server with dual 3.16 GHz Xeon X5460 quad-core processors.

4.1 Heartbeats in the PARSEC Benchmark Suite

To demonstrate the applicability of the Heartbeats framework across a range of multicore applications, it is applied to the PARSEC benchmark suite (version 1.0). For each benchmark, we read the description of the application, find the outermost loop and insert the heartbeats in this loop. Table 2 shows where the heartbeat was inserted in terms of the application's processing and the average heart rate that the benchmark achieved over the course of its execution running the "native" input data set on the eight-core x86 test platform[1]. We note that placement of heartbeats is flexible and can be tailored to the specific needs of the application.

For all benchmarks presented here, the Heartbeats framework is low-overhead. For eight of the ten benchmarks the overhead of Heartbeats was negligible. For the `blackscholes` benchmark, the overhead is negligible when registering a heartbeat every 25,000 options; however, in the first attempt a heartbeat was registered after every option was processed and this added an order of magnitude

[1] Two benchmarks are missing as neither `freqmine` nor `vips` would compile on the target system due to issues with the installed version of `gcc`.

Table 2. Heartbeats in the PARSEC Benchmark Suite

Benchmark	Heartbeat Location	Average Heart Rate (beat/s)
`blackscholes`	Every 25000 options	561.03
`bodytrack`	Every frame	4.31
`canneal`	Every 1875 moves	1043.76
`dedup`	Every "chunk"	264.30
`facesim`	Every frame	0.72
`ferret`	Every query	40.78
`fluidanimate`	Every frame	41.25
`streamcluster`	Every 200000 points	0.02
`swaptions`	Every "swaption"	2.27
`x264`	Every frame	11.32

slow-down. For the other benchmark with measurable overhead, `facesim`, the added time due to the use of Heartbeats is less than 5 %.

Adding heartbeats to the PARSEC benchmark suite is easy, even for users who are unfamiliar with the benchmarks themselves. The PARSEC documentation describes the inputs for each benchmark. With that information it is simple to find the key loops over the input data set and insert the call to register a heartbeat in this loop. The total amount of code required to add heartbeats to each of the benchmarks is under half-a-dozen lines. The extra code is simply the inclusion of the header file and declaration of a Heartbeat data structure, calls to initialize and finalize the Heartbeats run-time system, and the call to register each heartbeat.

In summary, the Heartbeats framework is easy to insert into a broad array of applications and our reference implementation is low-overhead. The next section provides an example of using the Heartbeats framework to develop an adaptive application.

4.2 Internal Heartbeat Usage

This example shows how Heartbeats can be used within an application to help a real-time H.264 video encoder maintain an acceptable frame rate by adjusting its encoding quality to increase performance. For this experiment the x264 implementation of an H.264 video encoder [10] is augmented so that a heartbeat is registered after each frame is encoded. x264 registers a heartbeat after every frame and checks its heart rate every 40 frames. When the application checks its heart rate, it looks to see if the average over the last forty frames was less than 30 beats per second (corresponding to 30 frames per second). If the heart rate is less than the target, the application adjusts its encoding algorithms to get more performance while possibly sacrificing the quality of the encoded image.

For this experiment, x264 is launched with a computationally demanding set of parameters for Main profile H.264 encoding. Both the input parameters and the video used here are different than the PARSEC inputs; both are chosen to be more computationally demanding and more uniform. The parameters include the use of exhaustive search techniques for motion estimation, the analysis of all macroblock sub-partitionings, x264's most demanding sub-pixel motion estimation, and the use of up to five reference frames for coding predicted frames. Even on the eight core machine with x264's assembly optimizations enabled, the unmodified x264 code-base achieves only 8.8 heartbeats per second with these inputs.

As the Heartbeat-enabled x264 executes, it reads its heart rate and changes algorithms and other parameters to attempt to reach an encoding speed of 30 heartbeats per second. As these adjustments are made, x264 switches to algorithms which are faster, but may produce lower quality encoded images.

Figures 2(a) and 2(b) illustrate the behavior of this adaptive version of x264 as it attempts to reach its target heart rate of 30 beats per second. The first figure shows the average heart rate over the last 40 frames as a function of time (time is measured in heartbeats or frames). The second figure illustrates how the change in algorithm affects the quality (measured in peak signal to noise ratio) of the encoded frames.

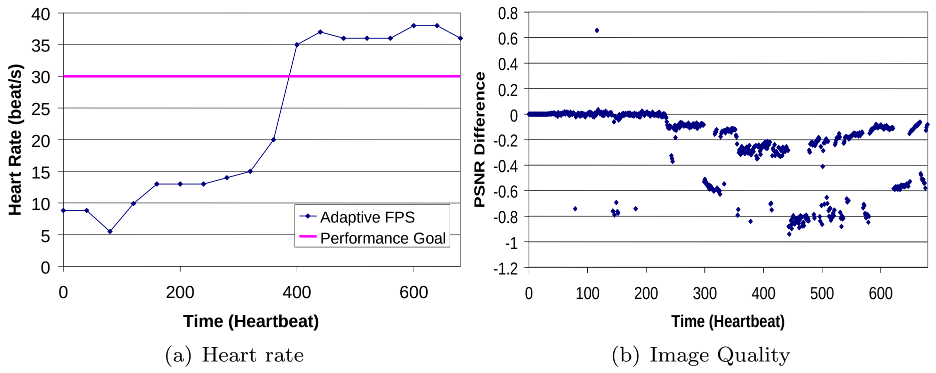


Fig. 2. Heart rate and image quality of adaptive x264. (a) shows how the heart rate of x264 changes as the program adapts to meet its goals. (b) shows the difference in PSNR between the unmodified x264 code base and our adaptive version.

As shown in Figure 2(a) the adaptive implementation of x264 gradually increases its speed until frame 400, at which point it makes a decision allowing it to maintain a heart rate over thirty-five beats per second. Given these inputs and the target performance, the adaptive version of x264 tries several search algorithms for motion estimation and finally settles on the computationally light diamond search algorithm. Additionally, this version of x264 stops attempting to use any sub-macroblock partitionings. Finally, the adaptive encoder decides to use a less demanding sub-pixel motion estimation algorithm.

As shown in Figure 2(b), as x264 increases speed, the quality, measured in PSNR, of the encoded images decreases. This figure shows the difference in PSNR between the unmodified x264 source code and the Heartbeat-enabled implementation which adjusts its encoding parameters. In the worst case, the adaptive version of x264 can lose as much as one dB of PSNR, but the average loss is closer to 0.5 dB. This quality loss is just at the threshold of what most people are capable of noticing. However, for a real-time encoder using these parameters on this architecture the alternative would be to drop two out of every three frames. Dropping frames has a much larger negative impact on the perceived quality than losing an average of 0.5 dB of PSNR per frame.

This experiment demonstrates how an application can use the Heartbeats API to monitor itself and adapt to meet its own needs. This allows the programmer to write a single general application that can then be run on different hardware platforms or with different input data streams and automatically maintain its own real-time goals. This saves time and results in more robust applications compared to writing a customized version for each individual situation or tuning the parameters by hand.

Videos demonstrating the adaptive encoder are available online. These videos are designed to capture the experience of watching encoded video in real-time as it is produced. The first video shows the heart rate of the encoder without adaptation[2]. The second video shows the heart rate of the encoder with adaptation[3].

4.3 External Heartbeat Usage

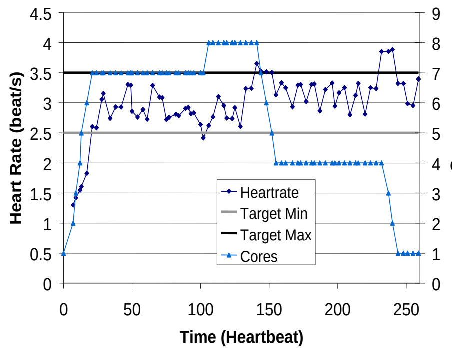
In this example, Heartbeats are used to help an external system allocate resources while maintaining required application performance. The application communicates performance information and goals to an external observer which attempts to keep performance within the specified range using the minimum number of cores possible. Three of the Heartbeat-enabled PARSEC benchmarks are run while an external scheduler reads their heart rates and adjusts the number of cores allocated to them. The applications tested include the PARSEC benchmarks `bodytrack`, `streamcluster`, and `x264`.

bodytrack The `bodytrack` benchmark is a computer vision application that tracks a person's movement through a scene. For this application a heartbeat is registered at every frame. Using all eight cores of the x86 server, the `bodytrack` application maintains an average heart rate of over four beats per second. The external scheduler starts this benchmark on a single core and then adjusts the number of cores assigned to the application in order to keep performance between 2.5 and 3.5 beats per second.

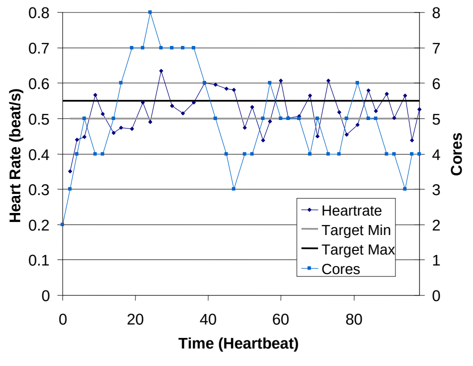
The behavior of `bodytrack` under the external scheduler is illustrated in Figure 3(a). This figure shows the average heart rate as a function of time measured in beats. As shown in the figure, the scheduler quickly increases the assigned cores until the application reaches the target range using seven cores. Performance stays within that range until heartbeat 102, when performance dips below 2.5

[2] Available here: http://www.youtube.com/watch?v=c1t30MDcpP0

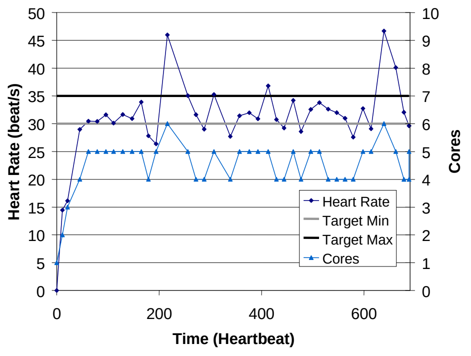
[3] Available here: http://www.youtube.com/watch?v=Msr22JcmYWA



(a) bodytrack



(b) streamcluster



(c) x264

Fig. 3. Behavior of selected PARSEC applications coupled with an external scheduler.

beats per second and the eighth and final core is assigned to the application. Then, at beat 141 the computational load suddenly decreases and the scheduler is able to reclaim cores while maintaining the desired performance. In fact, the application eventually needs only a single core to meet its goal.

The `streamcluster` benchmark solves the online clustering problem for a stream of input points by finding a number of medians and assigning each point to the closest median. For this application one heartbeat is registered for every 5000 input points. Using all eight cores of the x86 server, the `streamcluster` benchmark maintains an average heart rate of over 0.75 beats per second. The scheduler starts this application on a single core and then attempts to keep performance between 0.5 and 0.55 beats per second.

The behavior of `streamcluster` under the external scheduler is displayed in Figure 3(b). This figure shows the average heart rate as a function of time (measured in heartbeats). The scheduler adds cores to the application to reach the target heart rate by the twenty-second heartbeat. The scheduler then works to keep the application within the narrowly defined performance window. The figure illustrates that the scheduler is able to quickly react to changes in application performance by using the Heartbeats interface.

x264 The x264 benchmark is the same code base used in the internal optimization experiment described above. Once again, a heartbeat is registered for each frame. However, for this benchmark the input parameters are modified so that x264 can easily maintain an average heart rate of over 40 beats per second using eight cores. The scheduler begins with x264 assigned to a single core and then adjusts the number of cores to keep performance in the range of 30 to 35 beats per second.

Figure 3(c) shows the behavior of x264 under the external scheduler. Again, average heart rate is displayed as a function of time measured in heartbeats. In this case the scheduler is able to keep x264's performance within the specified range while using four to six cores. As shown in the chart the scheduler is able to quickly adapt to two spikes in performance where the encoder is able to briefly achieve over 45 beats per second. A video demonstrating the performance of the encoder running under the adaptive external scheduler has been posted online[4].

These experiments demonstrate a fundamental benefit of using the Heartbeats API for specifying application performance: external services are able to read the heartbeat data and adapt their behavior to meet the application's needs. Furthermore, the Heartbeats interface makes it easy for an external service to quantify its effects on application behavior. In this example, an external scheduler is able to adapt the number of cores assigned to a process based on its heart rate. This allows the scheduler to use the minimum number of cores necessary to meet the application's needs. The decisions the scheduler makes are based directly on the application's performance instead of being based on priority or some other indirect measure.

4.4 Heartbeats for Fault Tolerance

The final example in this section illustrates how the Heartbeats framework can be used to aid in fault tolerance. This example reuses the adaptive H.264 encoder developed above in Section 4.2. The adaptive encoder is initialized with a parameter set that can achieve a heart rate of 30 beat/s on the eight-core testbed. At frames 160, 320, and 480, a core failure is simulated by restricting the scheduler to running x264 on fewer cores. After each core failure the adaptive encoder detects a drop in heart rate and adjusts its algorithm to try to maintain its target performance.

The results of this experiment are shown in Figure 4. This figure shows a moving average of heart rate (using a 20-beat window) as a function of time for three data sets. The first data set, labeled "Healthy," shows the behavior of unmodified x264 for this input running on eight cores with no failures. The second data set, labeled "Unhealthy," shows the behavior of unmodified x264 when cores "die" (at frames 160, 320, and 480). Finally, the data set labeled

[4] Available here: http://www.youtube.com/watch?v=l3sVaGZKgkc

"Adaptive" shows how the adaptive encoder responds to these changes and is able to keep its heart rate above the target even in the presence of core failures.

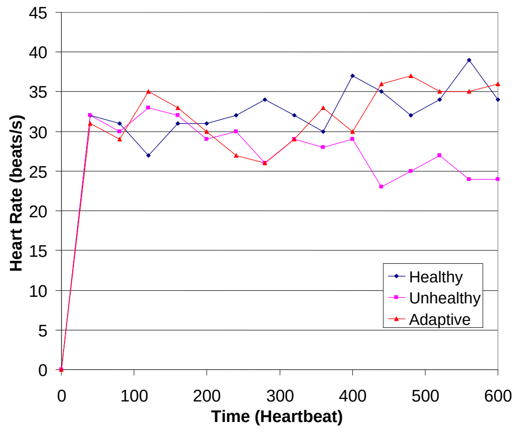


Fig. 4. Using Heartbeats in an adaptive video encoder for fault tolerance. The line labeled "Healthy" shows the performance of the encoder under normal circumstances. The line labeled "Unhealthy" shows the performance of the encoder when cores fail. The line labeled "Adaptive" shows the performance of an adaptive encoder that adjusts its algorithm to maintain a target heart rate of greater than 30 beats/s.

Figure 4 shows that in a healthy system, x264 is generally able to maintain a heart rate of greater than 30 beat/s. Furthermore, the performance in the healthy case actually increases slightly towards the end of execution as the input video becomes slightly easier at the end. In the unhealthy system, where cores die, the unmodified x264 is not able to maintain its target heart rate and performance falls below 25 beat/s. However, the adaptive encoder is able to change the algorithm and maintain performance in the face of hardware failures.

The adaptive encoder does not detect a fault or attempt to detect anything about which, or how many, cores are healthy. Instead, the adaptive encoder only attempts to detect changes in performance as reflected in the heart rate. The encoder is then able to adapt its behavior in order to return performance to its target zone.

The generality of this approach means that the encoder can respond to more than just core failures. For example, if a cooling fan failed and the hardware lowered its supply voltage to reduce power consumption, the encoder would detect the loss of performance and respond. Any event that alters performance will be detected by this method and allow the encoder a chance to adapt its behavior in response. Thus, the Heartbeats framework can aid fault tolerance and detection by providing a general way to detect changes in application performance.

5 Related Work

The problem of performance monitoring is fundamental to the development of parallel applications, so it has been addressed by a variety of different approaches. This work includes research on monitoring single- and multi-core architectures [2, 11, 12], networks [13], complex software systems and operating systems [14–20]. Most of this work focuses on off-line collection and visualization of performance data. More complex monitoring techniques have been presented in [21, 19]. This work represents a shift in approach as the research community moves from using simple hardware-based metrics, *i.e.,*cache miss rate, to more advanced statistics. Hardware assistance for system monitoring, often in the form of event counters, is included in most common architectures. However, counter-based techniques suffer common shortcomings [22]: too few counters, sampling delay, and lack of address profiling. In addition, adaptive systems based on hardware event counters must infer application performance from low-level hardware statistics.

A software approach for application monitoring is proposed in [14]. This work proposes an assertion based framework which can be used to verify that the runtime performance meets expected performance. Using the assertions, the programmer specifies performance expectations which the application can use at runtime to adapt itself. This framework allows a rich description of the program performance in terms of hardware specific parameters like the expected rate of floating point operations; however, use of the framework requires extensive code annotation and only allows the application to make internal updates to itself. In contrast, the Heartbeat framework is designed to specify performance in terms of a simple, and general mechanism and directly communicate this performance to external systems which can customize their behavior to meet those goals. We envision the use of the Heartbeat framework within a broader context where multiple applications can be executed in parallel, each using heartbeats to communicate performance and relying on the external services to help them meet their goals.

The rise of adaptive computing systems creates new challenges and demands for system monitoring [23]. One example of these emerging adaptive systems can be found in the self-optimizing memory controller described in [24]. This controller optimizes its scheduling policy using reinforcement learning to estimate the performance impact of each action it takes. As designed, performance is measured in terms of memory bus utilization. The controller optimizes memory bus utilization because that is the only metric available to it, and better bus utilization generally results in better performance. However, it would be preferable for the controller to optimize application performance directly and the Heartbeats API provides a mechanism with which to do so. Furthermore, the Heartbeats API is kept simple, which makes it easy for not only the end-users to get started with the framework but also to hook up third party auto-tuning tools such as Orio [25], Autopilot [26], Active Harmony [27], to make the application adaptation decisions based on the observed heartbeat rates. The fact that this research could be built on top of the Heartbeat interface demonstrates the API's usefulness.

System monitoring, as described in this section, is a crucial task for several very different goals: performance, security, quality of service, etc. Different ad hoc techniques for self-optimization have been presented in the literature, but the Heartbeats approach is the only one that provides a simple, unified framework for reasoning about and addressing all of these goals.

6 Conclusion

Our prototype results indicate that the Heartbeats framework is a useful tool for both application auto-tuning and externally-driven optimization. Our experimental results demonstrate three useful applications of the framework: dynamically reducing output quality (accuracy) as necessary to meet a throughput (performance) goal, optimizing system resource allocation by minimizing the number of cores used to reach a given target output rate, and tolerating failures by adjusting output quality to compensate for lost computational resources. The authors have identified several important applications that the framework can be applied to: self-optimizing microarchitectures, self-tuning software libraries, smarter system administration tools, novel "Organic" operating systems and runtime environments, and more profitable cloud computing clusters. We believe that a unified, portable standard for application performance monitoring is crucial for a broad range of future applications.

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2) (2009) 1–42
2. Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., Tullsen, D.: Processor power reduction via single-isa heterogeneous multi-core architectures. Computer Architecture Letters **2**(1) (Jan-Dec 2003) 2–2
3. Govil, K., Chan, E., Wasserman, H.: Comparing algorithm for dynamic speed-setting of a low-power CPU. In: MobiCom '95: Proceedings of the 1st Annual Inter. Conf. on Mobile Computing and networking. (1995) 13–25
4. Pering, T., Burd, T., Brodersen, R.: The simulation and evaluation of dynamic voltage scaling algorithms. In: ISLPED '98: Proceedings of the 1998 Inter. Symp. on Low Power Electronics and Design. (1998) 76–81
5. Eastep, J., Wingate, D., Santambrogio, M.D., Agarwal, A.: Smartlocks: Self-aware synchronization through lock acquisition scheduling. Technical Report MIT CSAIL, MIT (Nov 2009)
6. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: PetaBricks: A language and compiler for algorithmic choice. In: Conf. on Programming Language Design and Implementation. (Jun 2009)
7. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: Proceedings of the international symposium on code generation and optimization. (2003)

8. Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., Rinard, M.: Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT (September 2009)
9. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: PACT-2008: Proceedings of the 17th Inter. Conf. on Parallel Architectures and Compilation Techniques. (Oct 2008)
10. x264. Online document, `http://www.videolan.org/x264.html`
11. Intel Inc.: Intel itanium architecture software developer's manual (2006)
12. Azimi, R., Stumm, M., Wisniewski, R.W.: Online performance analysis by statistical sampling of microprocessor performance counters. In: ICS '05: Proceedings of the 19th Inter. Conf. on Supercomputing. (2005) 101–110
13. Wolski, R., Spring, N.T., Hayes, J.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems **15**(5–6) (1999) 757–768
14. Vetter, J., Worley, P.: Asserting performance expectations. In: Supercomputing, ACM/IEEE 2002 Conference. (Nov. 2002) 33–33
15. Caporuscio, M., Di Marco, A., Inverardi, P.: Run-time performance management of the siena publish/subscribe middleware. In: WOSP '05: Proc. of the 5th Inter. Work. on Software and performance. (2005) 65–74
16. De Rose, L.A., Reed, D.A.: SvPablo: A multi-language architecture-independent performance analysis system. In: Inter. Conf. on Parallel Processing. (1999)
17. Cascaval, C., Duesterwald, E., Sweeney, P.F., Wisniewski, R.W.: Performance and environment monitoring for continuous program optimization. IBM J. Res. Dev. **50**(2/3) (2006) 239–248
18. Krieger, O., Auslander, M., Rosenburg, B., W., R.W.J., Xenidis, Silva, D.D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., Uhlig, V.: K42: Building a complete operating system. In: EuroSys '06: Proc. of the 1st ACM SIGOPS/EuroSys Euro. Conf. on Computer Systems. (2006)
19. Wisniewski, R.W., Rosenburg, B.: Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In: SC '03: Proc. of the ACM/IEEE conf. on Supercomputing. (Nov 2003)
20. Tamches, A., Miller, B.P.: Fine-grained dynamic instrumentation of commodity operating system kernels. In: OSDI '99: Proc. of the third symp. on Operating systems design and implementation. (1999)
21. Schulz, M., White, B.S., McKee, S.A., Lee, H.H.S., Jeitner, J.: Owl: next generation system monitoring. In: CF '05: Proc. of the 2nd conf. on Computing Frontiers. (2005)
22. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro **22**(4) (Jul/Aug 2002) 64–71
23. Dini, P.: Internet, GRID, self-adaptability and beyond: Are we ready? (Aug 2004)
24. Ipek, E., Mutlu, O., Martnez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. In: ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch. (2008)
25. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using orio. In: IPDPS '09: Proc. of the Inter. Symp. on Parallel&Distributed Processing. (2009)
26. Ribler, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: adaptive control of distributed applications. In: High Performance Distributed Computing. (Jul 1998)
27. Hollingsworth, J., Keleher, P.: Prediction and adaptation in active harmony. In: High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on. (Jul 1998) 180–188

Automated Timer Generation for Empirical Tuning *

Josh Magee Qing Yi R. Clint Whaley

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX
{jmagee,qingyi,whaley}@cs.utsa.edu

Abstract. This paper presents a framework that significantly reduces the time required for automatically applying empirical tuning to improve the performance of large scientific applications, where the overall performance is often critically determined by a small number of individual routines that are either repetitively invoked or include a large number of loop iterations. Our framework allows these critical routines to be evaluated separately from their original applications by automatically generating timing drivers that accurately replicate their execution environment from within the whole applications. We have explored several alternatives to precisely simulate the input parameters, cache states of machines, and working environment of critical routines from both ATLAS and SPEC2006. Our experiments show that our timing drivers can accurately replicate the performance of these routines when invoked directly within whole applications, while reducing the time required to tune these routines by multiple orders of magnitude.

1 Introduction

In recent years, empirical tuning [9, 2, 4, 18, 19, 8, 11] has become a de facto approach that both developers and optimizing compilers adopt to extract high performance for scientific applications on a wide variety of modern computing platforms. However, since auto-tuning typically requires differently optimized code to be recompiled and re-executed hundreds or even thousands of times, the cost of experimentally evaluating a large optimization space could be prohibitive, especially for large scientific applications that take minutes or even hours to complete each run. In particular, for one of the SPEC2006 applications, we have found the time required to run the entire application is more than 175,000 times longer than evaluating the routine of interest independently. It is therefore critically important to reduce the cost of each empirical evaluation of the optimized code, so that a sufficiently large optimization space can be explored to identify the desirable optimization configurations.

* This research is supported by the National Science Foundation under grant No. CCF-0833203, CCF-0747357, and CNS-0551504.

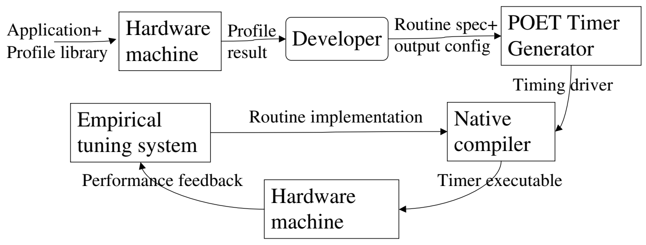


Fig. 1: The Timer Generation Framework.

We present a framework that significantly reduces the time required to empirically evaluate the performance of differently optimized code. Our framework is based on the observation that large scientific applications often critically depend on a few computationally intensive routines that are either invoked numerous times by the application and/or include a significant number of loop iterations. Since these routines are often chosen as the target of empirical performance tuning, the tuning time can be greatly reduced by separately studying the performance of an individual routine independent of the original application. Our experimental results show multiple orders of magnitude speedup in execution time when collecting performance feedback of a routine using an independent timing driver instead of instrumenting the whole application.

Separately studying the performance of individual routines requires a timing driver that 1)invokes the routine with an appropriate execution environment and 2)accurately reports the performance of each invocation. Whaley and Castaldo [17] showed that the measured performance could be seriously skewed when the runtime state of the system, especially caches, is not properly controlled by the timer before calling the routine. Unless the measured performance of the routine accurately reflects its expected performance when invoked directly within the whole application, the feedback could mislead the auto-tuning system into producing sub-optimal code.

Our framework automatically generates independent timing drivers that employ several approaches to precisely simulate the input parameters, cache states of machines, and working environment of individual routines so that their performance accurately matches those observed when instrumenting the original application. In particular, after identifying an individual routine to be tuned from within a large application, the work flow of our framework is shown in Figure 1. The framework starts with an instrumentation library which is invoked in a profiling run of the whole application to collect details of the routine's execution environment. The profiling result is then used to produce a *routine specification*, which is then used as input to a POET (Parameterized Optimizations for Empirical Tuning) [20] code generator to automatically produce a timing driver. The timing driver can then be compiled and repetitively used in the iterative performance tuning process, where every time a differently optimized code is generated for the routine, the routine implementation is compiled and linked with the timing driver to produce an executable (i.e., the *timer*), which measures the performance of the routine implementation on the targeted machine and reports performance back to the tuning system.

Our framework currently requires manual user intervention to profile the application and to write the routine specification after profiling. However, it provides a complete instrumentation library to collect the routine's execution environment (i.e., all the information required to write a *routine specification*) and provides a general-purpose code generator to automatically produce a reconfigurable implementation of the timing driver. Additionally, our framework provides a library that uses automatic *application checkpointing*[5] to control the execution environment of routines that are data-sensitive (e.g., an array sorting algorithm or a pointing chasing algorithm). We show that by checkpointing several instructions or iterations before invoking the routine of interest, the accuracy of timing results can be greatly improved.

We have applied our framework to tune several routines from ATLAS and SPEC2006. Our experiments show that our framework can accurately replicate the performance of these routines when invoked directly within whole applications, while significantly reducing the time required to tune these routines.

2 Related Work

Whaley and Castaldo [17] explored methods for achieving accurate and context-sensitive timing for code optimization. We have automated the generation of context-sensitive timers and have investigated both the efficiency and effectiveness of such timers in replicating the expected performance of routines invoked within applications. Apart from [17], the literature dedicated to accurately measuring performance has been limited to benchmarking systems [6, 13, 22].

The automatically generated timers presented in this paper can be integrated within a large body of existing auto-tuning frameworks, including many general-purpose iterative compilation frameworks [9, 1, 3, 11, 14, 19], and a large number of domain-specific tuning systems, e.g., ATLAS [18, 17], SPIRAL [10, 8], FFTW [4], PHiPAC [2], OSKI [15], and X-Ray [22], to provide performance feedback for the routines being optimized. The timers employed in ATLAS [16] are used in our research as a baseline for comparison.

When used to measure performance, most profiling systems, such as HPC-Toolkit [7], need to evaluate the entire application even when the performance of only a single routine is required. In contrast, our framework supports the timing of routines independent of their original applications. We use profiling only to collect the execution environment of routines.

Our application checkpointing approach (see Section 3.2) follows that documented in the ROSE compiler [12]. We have additionally introduced a delayed checkpointing approach that increases the accuracy of obtained timings.

3 Profiling Performance of Applications

The challenge faced when timing routines independently of their applications is ensuring that appropriate input values and operand workspaces are provided when invoking the routine. In particular, the supplied values should not result in abnormal execution (e.g., exceptions, segmentation faults) and should reflect the common usage pattern of the routine.

We have developed an instrumentation library to collect information on the common usage patterns (as well as the performance) of routines when invoked within an application. We separate these routines into two rough categories. The first category of routines is *data insensitive*; that is, the amount of computation within the routine is determined by a few integer parameters controlling problem size, but is not noticeably affected by the particular values stored in the input data structures. An example of such routines is dense matrix multiplication. The second category includes routines that are much more *data sensitive*, e.g., a sorting algorithm whose performance is largely determined by the specific data values being operated upon, as the algorithm may exit immediately after finding out the data are already sorted. Other examples include complex pointer-chasing algorithms whose input can only be roughly approximated.

Based on whether the routine of interest is data sensitive, we use different approaches to simulate its execution environment. In particular, for data-insensitive routines (e.g., matrix multiply), our *default timer* recreates their performance by reproducing the same array sizes, initializing arrays using a random number generator, and carefully controlling the memory hierarchy state to match those used in the whole applications. For data sensitive routines (e.g., a pointer-chasing algorithm), we use a *checkpointing* approach, discussed in Section 3.2. Note that the default timing approach may still allow reasonable tuning of some data-sensitive routines, as illustrated by our experimental results in Section 5.

3.1 The Default Timing approach

```
1: $n \leftarrow$ number of routines
2: $r \leftarrow$ list of routines to instrument
3: $p \leftarrow$ maximum number of parameters $\times$ sizeof(largest parameter type)
4: $i \leftarrow$ estimated calls to routine + sizeof(double precision floating point)
5: on application start, Setup instrumentation data buffer $D[n \times p \times i]$
6: for each routine $i$ in $r$ do
7:     $v \leftarrow$ parameter values of $r_i$
8:     time $r_i$
9:     $t \leftarrow$ wall clock time of $r_i$
10:    $D \leftarrow$ key-value pair $r_i(v) : t$, avoid polluting cache
11: end for
12: on application termination, Write contents of $D$ to stdout or file
```

Fig. 2: Instrumentation Algorithm.

For data insensitive routines, we instrument their invocations within whole applications to record the integer parameter values and the wall clock times spent in evaluating each invocation. The collected information is then used to determine what values to supply when independently tuning these routines. The performance of routines collected within the applications is also used as reference in section 5 to determine the accuracy of the performance reported by our auto-generated timers.

Figure 2 presents the default approach to instrumenting the applications, where a data buffer D is created at the start of the application. The data buffer is of a configurable size, which is based upon the number of routines profiled, the maximum number of *sampled* parameters, and the estimated number of calls to each routine. Note that the maximum number of parameters include only parameters that are of interest (in most cases integer type parameters). Once setup is complete, every time a routine in r (the list of routines to instrument) is invoked, the routine parameter values (v) and execution time of the routine (t) are recorded. These values are written as a key-value pair to the data buffer (D) using instructions that avoid cache pollution. Upon application termination, the contents of data buffer D are written to a file or *stdout*.

3.2 The Checkpointing Approach

When invoked within whole applications, a data-sensitive routine sometimes has complex data structures (e.g., linked lists, trees and graphs) which are almost impossible to replicate without the original application. For these routines, we have adopted the *checkpointing* approach, a technique most commonly associated with providing fault tolerance in systems, to precisely capture the memory snapshot before the application invokes the routine. A similar checkpointing approach was first successfully employed in an end-to-end empirical optimization system built using the ROSE compiler [12].

Checkpointing works by saving a “snapshot” image of a running application. This image can be used to restart the application from a saved context. Our framework utilizes the Berkeley Lab Checkpoint/Restart (BLCR) library [5]. It facilitates checkpointing by providing a tiny library that can be used to generate a context image of an application.

```
enter_checkpoint(CHECKPOINTING_IMAGE_NAME);
.....
starttime=GetWallTime();
retval = mainGtU(i1, i2, block, quadrant, nblock, budget);
endtime=GetWallTime();
.....
stop_checkpoint();
```

Fig. 3: Creating a contextualized snapshot of a routine.

Figure 3 demonstrates how a context image for a call to the routine *mainGtU()* can be created using two calls: *enter_checkpoint* and *stop_checkpoint*. Specifically, the created image includes all the data in memory before calling *enter_checkpoint* and all the instructions between *enter_checkpoint* and *stop_checkpoint*. The image can be used in a specification file to our POET timer generator, discussed in Section 4, to automatically generate a checkpoint driver. The driver then contains code that loads and restarts the checkpoint image and reports the time spent in evaluating the routine of interest. Note that while the intended usage involves checkpointing a call to a routine, this approach can be used to measure the performance of any critical region of code.

It is possible to call *enter_checkpoint* immediately prior to the region of interest. However, this is not the case in Figure 3. Since restoring a checkpoint memory image does not restore any of the data into cache (and indeed may force normally-cached data onto disk), it essentially destroys the cache state of the original program. To restore the cache state before calling the routine of interest, it is better to call *enter_checkpoint* several loop iterations or instructions ahead of the region of interest, so that execution of these instructions can help restore the original cache state before the timed region is reached. How far in advance to place the *enter_checkpoint* call is a trade-off between reproduction accuracy and sample time.

To calculate the amount of "delay" between the *enter_checkpoint* call and the region of interest, the distance between the checkpoint and the region is incrementally increased until there is no noticeable variance in performance. In detail, the *enter_checkpoint* call is placed immediately prior to the region of interest and a timing is obtained. Next the *enter_checkpoint* call is placed in the previous iteration (when the region is in a loop) or the previous function invocation (when the region is not in a loop) and a new timing is obtained. The process is applied iteratively until the variation between timings falls below a chosen threshold. This process is semi-automated: once the boilerplate calls (*enter_checkpoint* and *stop_checkpoint*) are in place the best "delay" can be selected. We refer to placing the checkpoint immediately prior to the region of interest as "immediate checkpointing" and to placing the checkpoint several instructions prior to the region as "delayed checkpointing."

4 Automatically Generating Timers

After profiling an application to collect information on the execution environment of individual routines, we express such information in the format of a *routine specification*, which is then used by our POET timer generator (see Figure 1) to automatically produce a reconfigurable implementation (currently in the C language) of the timing driver, so that performance of varying routine implementations can be tuned independently of their original applications.

```
routine=void ATL_USERMM(const int M,const int N, const int K,
              const double alpha, const double* A,const int lda,
              const double* B,const int ldb, const double beta,
              double* C, const int ldc);
init={
  M=Macro(MS,72); N=Macro(NS,72); K=Macro(KS,72);
  lda=MS; ldb=KS; ldc=MS; alpha=1; beta=1;
  A=Matrix(double, M, K, RANDOM, flush|align(16));
  B=Matrix(double, K, N, RANDOM, flush|align(16));
  C=Matrix(double, M, N, RANDOM, flush|align(16));
} ;
flop="2*M*N*K+M*N";
```

Fig. 4: gemm.spec: Sample specification for the *GEMM* driver.

An example routine specification for a matrix multiplication kernel is shown in Figure 4, which includes a routine declaration, a section to allocate, initialize, and control the cache states of routine parameters, and a formula for computing the MFLOPS (*millions of floating point operations per second*). In particular, three integer parameters, M, N, and K, are initialized with environmental macros whose values can be dynamically adapted by simply re-compiling the timing driver; three matrices A, B, and C are allocated with appropriate sizes, initialized with pseudo-randomly generated data, aligned to a 16 byte boundary, and are flushed between timings.

Our timer generator in Figure 1 is essentially a translator written in POET [20], an interpreted transformation language designed for building ad-hoc translators between arbitrary languages (e.g. C/C++, Java) as well as applying sophisticated transformations to programs in these languages. The POET translator is extensively parametrized with variables whose values can be redefined via command-line options. Therefore, a wide variety of timing driver implementations can be manufactured from a single routine specification by redefining values of the command-line parameters, including the input/output file names, the output language, cache size, Instruction Set Architecture, processor clock rate, and timing methods (e.g., whether to use wall or CPU time).

A template of the auto-generated timing driver is shown in Figure 5 , which can be instantiated with the concrete syntax of different programming languages (our current work uses the C language) as well as different implementations (e.g., elapsed wall time, cycle-accurate wall time, or CPU cycles) to measure the performance of the invoked routine. Note that some of the control-flow statements (e.g., `for-endfor` and `if-endif`) in Figure 5 are not part of the generated code. They are used by the POET timer generator to selectively (in the case of `if-endif`) or repetitively (in the case of `for-endfor`) produce necessary statements in the resulting driver.

```
Require: Routine definition $R$
  for each routine parameter $s$ in $R$ do
    if$s$ is a pointer or array variable then allocate memory $m_s$ for $s$ endif
    if$s$ needs to be initialized then initialize $m_s$ endif
  end for
  for each repetition of timing do
    if Cache flushing = true then Flush Cache endif
    $time_s \leftarrow$ current time
    call $R$
    $time_e \leftarrow$ current time
    $times_r \leftarrow time_e - time_s$
  end for
  Calculate $min$, $max$, and $average$ times from $times_r$
  if flops is defined then
    Calculate Max MFLOPS as $flops \times \frac{1,000,000}{min}$
    Calculate Average MFLOPS as $flops \times \frac{1,000,000}{average}$
  end if
  Print All timings
```

Fig. 5: Template of auto-generated timing Driver.

The generated driver repetitively invokes the routine, optionally flushes the cache for each invocation, and measures the elapsed time spent while invoking the routine. Once all timing measurements are collected, the minimum, average, and maximum timings are calculated and reported. If a formula to calculate the number of floating point operations is included in the routine specification, the maximum and average MFLOPS are also computed and reported. Further, if the execution time of a single routine invocation is under clock resolution, multiple invocations can be collectively measured to increase timing accuracy.

We adopt sophisticated cache flushing mechanisms as discussed in [17] in our timing drivers. Specifically, the cache flushing strategy is re-configurable by command-line and every strategy makes sure that the flushing does not stand in the way of accurately measuring the performance of the routine being timed.

5 Experimental Evaluation

The goal of our evaluation is to validate that our POET-generated timers can not only significantly reduce the tuning time for large applications, they can accurately reproduce the performance of the timed routine when invoked within whole applications. To achieve this goal, Section 5.2 shows the reduction in tuning time achieved using our framework. Section 5.3 compares the timing results using our auto-generated timer for a matrix multiplication kernel with those obtained using the ATLAS timer, which is known to accurately reflect the common usage patterns of the kernel. Sections 5.4, 5.5, and 5.6 present performance results for three routines selected from different SPEC2006 benchmarks.

Our results show that our timers can perform similarly to the ATLAS timer in accurately reporting performance of data-insensitive routines in scientific computing, and that the performance results reported by our timers closely reproduce those collected directly from within the SPEC2006 benchmarks.

5.1 Methodology

We performed our evaluation on two multicore platforms: a 3.0Ghz Dual-Core AMD Opteron 2222 and a 3.0Ghz Quad-Core Intel Xeon Mac Pro. The timings are obtained in serial mode using a single core of each machine.

To compare our auto-generated *default timer* (see Section 3.1) with the ATLAS timer, we selected five different implementations of the ATLAS Matrix Multiply (referred to as *MMK*) kernel automatically generated using techniques presented in [21], where each implementation differs only in the cache blocking factor. The routine specification of this kernel is shown in Figure 4. The implementation of the kernel has been heavily optimized using techniques described by Yi and Whaley in [21], and achieves roughly 80% of theoretical peak performance when timed in cache (therefore, while this is not the fastest known kernel, it is quite good).

Each timer is tested for the totally cold-cache state (all operands cache flushed, labeled as **Flush** in figures) and with operands that have been allowed to preload the levels of cache by an unguarded operand initialization immediately prior to the timing call (labeled as **No Flush** in figures).

On the AMD platform, each *MMK* implementation is compiled using `gcc` 4.2.4 with optimization flags `-fomit-frame-pointer -mfpmath=387 -O2 -falign-loops=4 -m64` (note we use the x87 rather than the vector unit because on 2nd generation Opterons, both units have the same double precision theoretical peak, but the x87 unit has markedly decreased code size and increased precision). On the Intel platform, each *MMK* implementations is compiled using `gcc` 4.0.1 with optimization flags `-fomit-frame-pointer -mfpmath=sse -msse3 -O2 -m64`.

We selected the following routines from the SPEC2006 benchmark suite.

– Routine *mult_su3_mat_vec*, from 433.milc, performs an array-based matrix-vector multiplication and is data-insensitive (i.e., the performance does not depend on the content of the input arrays). We compare the performance of the POET-generated *default timer* using randomly generated arrays with timings obtained by profiling the whole application.

– Routine *mainGtU*, from 401.bzip2, is a variant of the *quicksort* algorithm using arrays. The computation depends on the content of the array being sorted and thus is data-sensitive. We compare the performance results obtained by the POET-generated *default timer* using randomly generated arrays, by the POET-generated *checkpoint timer*, and by profiling the application.
– *scan_for_patterns*, from 445.gobmk, is an extremely data-sensitive routine that operates on pointer-based linked-list data structures. We compare our POET-generated *checkpoint timer* with timings obtained by profiling the application.

Each SPEC2006 benchmark is compiled on both the AMD and the Intel platforms using a variety of optimization flags, including `-O0, -O1, -O2, -O3,` and `-Os`. All POET-generated timers themselves are compiled with the flag `-O2`. When using POET-generated *checkpoint timers*, we present results using both the immediate checkpointing and delayed checkpointing approaches (for details of these approaches, see Section 3.2).

5.2 Cost comparison of timing mechanisms

	Benchmark	**Delayed Checkpoint**	**Immediate Checkpoint**	**Default Timer**
mult_su3_mat_vec	877,430ms	3,502ms	3,510ms	5ms
mainGtU	45,765ms	2,019ms	1,975ms	4ms
scan_for_patterns	90,460ms	6,218ms	5,930ms	n/a

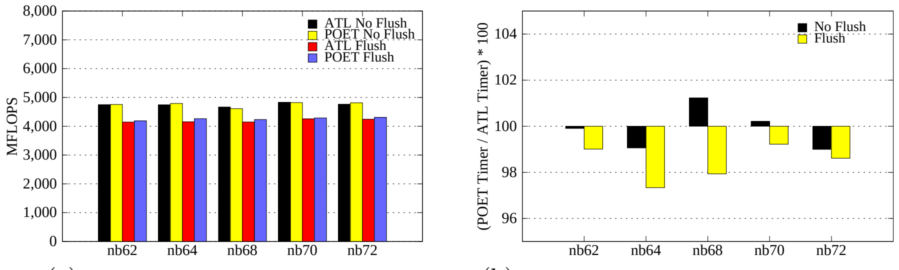
Table 1: Average runtimes for each SPEC2006 routine on AMD.

As shown in Table 1, the time required to collect performance feedback of a routine can be drastically reduced by using an independent timing driver instead of instrumenting the whole application. We see that our *default timer* is as much as 175,486 times faster than running the full benchmark, while even our *delayed checkpoint timer* is over 250 times faster. The timing results on the Intel machine are similar. These reductions in execution time are critically important for empirical tuning systems which may evaluate a routine hundreds or thousands of times.

For data-sensitive routines such as *mainGtU* and *scan_for_patterns*, checkpointing or running the full benchmark yields the most accurate results. However, these methods are also more expensive than the default timer approach. The *default timer* takes significantly less time to run than the other approaches and therefore should be used where possible. The *checkpoint timers* take significantly less time than running the entire benchmark and should be used when the values of the input arrays are critical or when pointer-chasing routines are involved.

5.3 Comparing with The ATLAS Timer

Figure 6 compares the timings reported by the ATLAS timer and POET *default timer* with and without cache flushing for a *MMK* kernel using five different

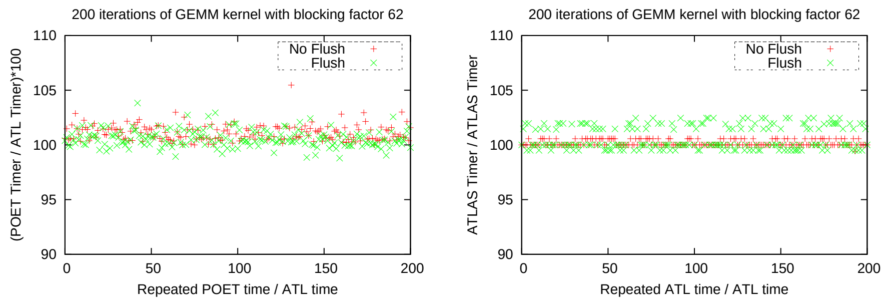


(a) ATLAS timer versus POET timer. (b) POET as a percentage of ATLAS timer

Fig. 6: ATLAS vs. POET timers with identical GEMM kernels on AMD.

cache blocking factors, where `nb62` denotes a blocking factor of 62*62 for the matrices. Figure 6(a) provides an overview of the timing results by comparing the MFLOPS measured by the ATLAS and the POET timers respectively. For each kernel implementation, the performance with cache flushing is slower than without flushing and the performance reported by ATLAS and POET are extremely close. This is easier to see in Figure 6(b), which reports the POET timer results as a percentage of the ATLAS timer results. For these kernel implementations, we see that the variation between the timers is less than 3%.

Timings taken on actual hardware running commodity OS are never precisely repeatable in detail. Since our POET timer is implemented independently of ATLAS, we expect some minor variance due to implementation details. The question is whether these relatively minor variances in timing represent errors, or if they are instead caused by the nature of empirical timings in the real world.



(a) ATLAS timer versus POET timer. (b) ATLAS timer versus ATLAS timer.

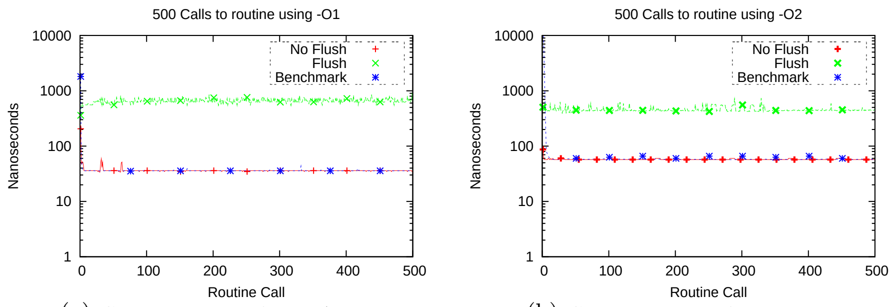
Fig. 7: Variance on a given kernel implementation between an initial ATLAS timing step with 200 subsequent results reported by POET or ATLAS timers on AMD.

Figure 7 sheds some light on this question: this figure shows the variation between timing runs for a single kernel implementation (with a predetermined blocking factor of 62), which we have timed 200 times. In Figure 7(a), we first run the ATLAS timer once for each cache state (+: no cache flushed, x: with cache flushing). We then time the same kernel 200 times using the POET *default timer*. For each cache state, we plot the POET timer's reported performance as a percentage of the original ATLAS measurement. We see some obvious jitter in the results, with variances that are mostly contained within a similar 3% range as we saw in Figure 6.

Figure 7(b) provides a strong indication that most of the observed variance is indeed due to the nature of empirical timings on actual machines. Here (using the same initial ATLAS timing as we compared against in Figure 7(a)), we do 200 more timings using the ATLAS timer itself. When we compare Figures 7 (a) and (b), we see that the variance between POET and ATLAS is only slightly greater than the variance between ATLAS and itself. Therefore, we conclude that our generated timer is able to adequately reproduce the behavior of ATLAS's hand-crafted timer for these hot and cold cache states.

5.4 Timing Results For SPEC2006 Routine *mult_su3_mat_vec*

This is a matrix-vector multiplication routine and is timed 1000 times using the POET-generated *default timer* (with arrays initialized with random values) both with and without cache flushing. The goal of this experiment is to verify that the *default timers* accurately replicates the measured performance from within the benchmark. Specifically, we expect that the *default timer* without cache flushing will accurately reproduce the benchmark's performance, since this routine is called with its operands in cache for all invocations except the initial call.

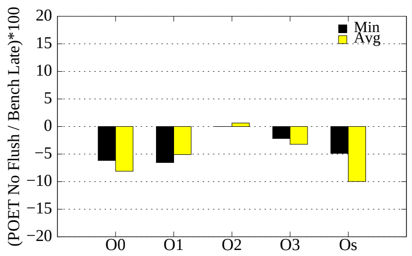


(a) Compiled with *-O1* on AMD. (b) Compiled with *-O2* on Intel.

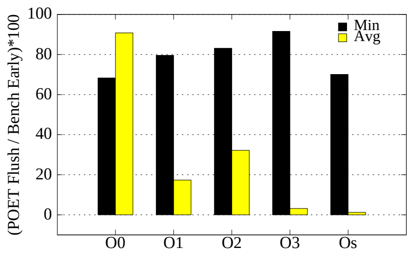
Fig. 8: Timings of 500 consecutive calls to *mult_su3_mat_vec*.

Figure 8 confirms our expectation by showing performance results of the routine when measured using the POET *default timers* (with and without cache flushing) and when measured from within the benchmark. As shown in Figure 8(a) and Figure 8(b), the default timings without flushing match extremely closely to the calls timed from within the benchmark, except when the routine is called the first couple of times (demonstrated by a lone aberrant $*$ along the 0th call), and a spike just past the 200th call of routine from within the application in (a). The spike is apparently due to unrelated activity affecting our wall times (this spike either disappears or shows up in other places if the timing is repeated). Therefore, we can classify this benchmark as using the kernel with warm caches (except the first few calls).

Since the routine uses statically initialized arrays, the input data are not in the cache on the first call, whose performance matches our **flush** timing results.



(a) No Flush / Late Group.



(b) Flush / Early Group.

Fig. 9: Minimum and Average timings of *mult_su3_vec* on AMD.

However, after calling the routine several times with the same workspace, the operands become cache contained, and so the overwhelming majority of calls closely match our **no flush** timings. On a cache with LRU cache line replacement, the second call would already have brought the data into any cache large enough to contain it. The machines we are using, however, have non-LRU replacement L2 caches, and this means that several passes over a piece of data are required before it is almost completely retained in the cache. Therefore, what we see is that the first call essentially runs at **flush** speed, and then the time for the next few calls decreases monotonically until the **no flush** time is reached. In such cases, it can be helpful to sort the usage contexts into several important groups, and tune them separately.

In Figure 9 we have sorted the first four routine calls into the **early** group, and all remaining calls into the **late** group. We see in Figure 9(a) that for both minimum and average times[1] that the **no flush** times are an extremely close match for the **late** group (verifying the general trend of Figure 8(a)). In Figure 9(b), we see that the **flush** group is not an exact match for our **early** group, which averages the first 4 calls, where only the first is fully out-of-cache. However, even this rough grouping would be adequate to tune this kernel for both contexts, assuming they were both important enough.

Our POET-generated timers can obviously capture the performance of routines when invoked either in-cache or fully out-of-cache, but for some applications the data may be only partially cache-contained. We can simulate these cases by partially flushing the cache in the *default timer*. Figure 10 demonstrates the ability of the *default timers* to capture the variation that arises as a result of the cache state. The *default timers* can reproduce a range of timings between the non-flushed and completely flushed state by using different flush sizes. This figure shows a progression of flushing sizes (from 512K to 2048K) in addition to no-flushing and complete flushing. Tracing through each flushing size (from none to full), a monotonic increase in the wall clock time can be observed such that an increase in the size of the flush results in an increase of time.

[1] maximum wall clock time can be extremely unstable and is therefore not used.

We can use the POET-generated *default timers* to reproduce timings that lie anywhere in between the **flush** and **no flush** lines. If the cache state of the application during a typical routine call is unknown (the usual case), profiling can be used to capture where the results lie, and the flush size can be adjusted so that the timer roughly reproduces the required cache state.

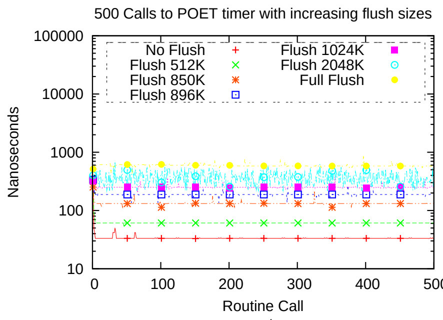
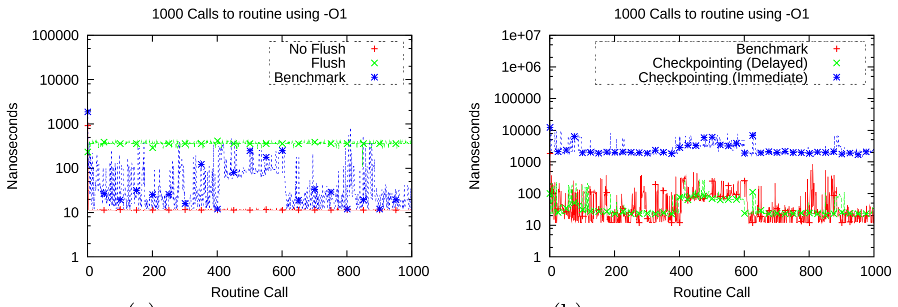


Fig. 10: POET Timers w/ increasing flush.

5.5 Timing Results For SPEC2006 Routine *mainGtU*

The routine *mainGtU* is a sorting algorithm whose performance is sensitive to the content of the input array. We have used both the POET-generated *default timer* and the POET-generated *checkpoint timer* to independently measure performance of this routine. The goal of this experiment is to determine how closely both the *default timer* and the *checkpoint timer* can reproduce the timings obtained from instrumenting the benchmark.



(a) POET timings. (b) Checkpointing timings.

Fig. 11: 1000 consecutive calls to *mainGtU* on AMD.

Figure 11(a) compares timings generated using the *default timer* with those taken from profiling the entire benchmark. Since the *default timer* uses randomly generated data, it will not capture cases where the data is in a particular order (eg., near sorted). Given these factors it is not expected that the POET-generated *default timer* should identically replicate each individual call; indeed unless the data distribution and order can be characterized, random data is probably as accurate as anything short of using the application's actual input.

If the precise behaviour of individual calls needs to be replicated, then checkpointing can yield accurate timings by exactly duplicating the input data. Figure 11(b)compares the results for both *immediate* (denoted by `*`) and *delayed* (`x`) checkpointing to those obtained from profiling the benchmark (`+`). Immediate checkpointing creates a checkpoint image of exactly one routine call. Delayed checkpointing creates a checkpoint image of several calls (three calls for this timing) to the routine, of which only the last call is timed. The methodology to determine how much to “delay” is discussed in section 3.2. Immediate checkpointing results in a cold memory hierarchy, and we see that these numbers are therefore even slower than our **flush** results from Figure 11(a). Delayed checkpointing allows for bringing the working set into the cache, thereby allowing for an extremely accurate reproduction of the timings taken from within the benchmark.

5.6 Timing Results For SPEC2006 Routine *scan_for_patterns*

The routine *scan_for_patterns* is a data-sensitive pattern matching algorithm that processes pointer-based linked-lists. Faithfully reproducing the necessary contexts for such a pointer-chasing algorithm is extremely difficult and often not feasible. To support the tuning of routines such as *scan_for_patterns* we use the POET-generated *checkpoint timer*. Here, the goal of our experiment is to demonstrate that the performance of complex pointer-chasing routines can be accurately replicated using the *checkpoint timer*.

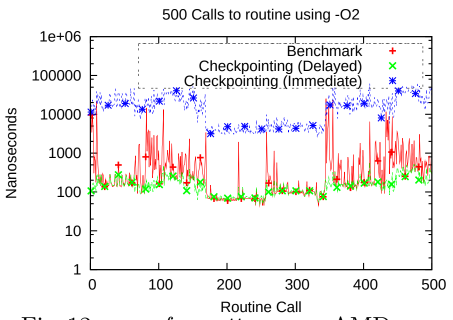


Fig. 12: *scan_for_patterns* on AMD.

Figure 12 compares the result of delayed and immediate checkpointing with timings obtained by profiling the benchmark. We can see that delayed checkpointing (designated by `x`) closely replicates the benchmark profiled timings (`+`) while immediate checkpointing follows the general trend of the benchmark (i.e., calls 200-300 are faster than 400-500) but with increased times due to the cold-cache state as discussed in Section 3.2.

6 Conclusion

This paper presented a general-purpose framework for automatically generating timing drivers that can accurately report the performance of computational routines in the context of automatic performance tuning. We have explored a variety of ways to accurately reproduce the common usage patterns of a routine so that when independently timing the routine, the reported performance results accurately reflect the expected performance of the routine when invoked directly within applications. We have demonstrated the importance of checkpointing several instructions or iterations before the routine being measured (delayed checkpointing) in obtaining accurate timings. Finally, we have shown that using the auto-generated timers can significantly reduce tuning time without compromising tuning accuracy.

References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006).*, New York, NY, 2006.
2. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.
3. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*, San Jose, CA, USA, March 2005.
4. M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.
5. Future Technologies Group. Berkeley lab checkpoint/restart (blcr), 2009. https://ftg.lbl.gov/CheckpointRestart.
6. L. McVoy and C. Staelin. lmbench: portable tools for performance. In *Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, pages 35–44, Berkeley, California, 1996. USENIX Association.
7. J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 2002. In press. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*
8. J. Moura, J. Johnson, R. Johnson, D. Padua, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Proceedings of the Conference on High-Performance Embedded Computing*, MIT Lincoln Laboratories, Boston, MA, 2000.
9. Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.
10. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
11. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2004.
12. ROSE Team. Rose-based end-to-end empirical tuning: Draft user tutorial (associate with rose version 0.9.4a), 2009. www.rosecompiler.org.
13. A. Saavedra and R. Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, Oct 1995.
14. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.
15. R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC*, San Francisco, California, 2005. Institute of Physics Publishing.
16. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
17. R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software—Practice and Experience*, 38(15):1621–1642, 2008.
18. R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.
19. R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *34th International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, 2005. IEEE Computer Society.
20. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Long Beach, California, Mar 2007.
21. Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.
22. K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *In Proceedings of the 2nd International Conference on Quantitative Evaluation of SysTems*, 2005.

Static Java Program Features for Intelligent Squash Prediction

Jeremy Singer, Paraskevas Yiapanis, Adam Pocock, Mikel Lujan, Gavin Brown, Nikolas Ioannou, and Marcelo Cintra

[1] University of Manchester, UK
jsinger@cs.man.ac.uk
[2] University of Edinburgh, UK

Abstract. The *thread-level speculation* paradigm parallelizes sequential applications at run-time, via optimistic execution of potentially independent threads. This enables unmodified sequential applications to exploit thread-level parallelism on modern multicore architectures. However a high frequency of data dependence violations between speculative threads can severely degrade the performance of thread-level speculation. Thus it is crucial to be able to schedule speculations to avoid excessive data dependence violations. Previous work in this area relies mainly on program profiling or simple heuristics to avoid thread squashes. In this paper, we investigate the use of *machine learning* to construct squash predictors based on static program features. On a set of standard Java benchmarks, with leave-one-out cross-validation, our approach significantly improves speculation performance for two benchmarks, but unfortunately degrades it for another two, in relation to a spawn-everywhere policy. We discuss how to advance research on squash prediction, directed by machine learning.

1 Introduction

With the emergence of multi-core architectures, it is inevitable that parallel programs are favored as they are able to take advantage of the available computing resource. However there is a huge amount of legacy sequential code. Additionally, parallel programs are difficult to write as they require advanced programming skills. Some state-of-the-art compilers can automatically parallelize sequential code in order to run on a multi-core system. However such compilers conservatively refuse to parallelize code where data dependencies are ambiguous. Thread-Level Speculation (TLS) has received a lot of attention in recent years as a means of facilitating aggressive auto-parallelization [1] [2] [3] [4] [5] [6] [7] [8]. TLS neglects any ambiguities in terms of dependencies and proceeds in parallel with future computation in a separate speculative state as if those dependencies were absent. The results are then checked for correctness. If they are correct, the speculative state can safely write its side effects back to memory (*i.e.* commit). If the results are wrong, all speculative state is discarded and the computation is re-executed serially. (*i.e.* squash).

A high number of squashes results in *performance degradation* as:

1. There is a relatively high overhead associated with thread-management (roll-back and re-execute).
2. Squashed threads waste processor cycles that could usefully be allocated to other non-violating parallel threads.

An optimal situation would be one that no cross-thread violation occurs and therefore all speculative threads can commit their state. The spawning policy (the speculation level) employed by a TLS system is an important factor here. However spawning policy alone cannot guarantee the absence of data dependence violations. Ideally we would like to have a mechanism that can detect conflicts ahead-of-time and thus ultimately decide whether to spawn a thread or not. In this paper we simulate method-level speculation or *Speculative Method-level Parallelism (SMLP)* in order to collect data about speculative threads that commit or squash. We then mine static characteristics of these Java methods using Machine Learning in order to relate general method properties to TLS behavior.

The main contributions of this paper are:

– a description of static program characteristics that may provide useful features for learning about Java methods (Section 3).
– a comparative evaluation of profile-based and learning-based policies for squash prediction, in the context of method-level speculation for Java programs (Section 4).
– an outline of future directions for investigation into learning-based squash prediction (Section 6).

2 Speculation Model

2.1 Speculative Method-Level Parallelism

Since the Java programming language is *object-oriented*, the natural unit of abstract behavior is the *method*. Thus we assume that distinct methods are likely to have independent behavior, so methods are suitable code segments for scheduling as parallel threads of execution [9] [10] [11] [12].

Figure 1 presents a graphical overview of how SMLP operates, given a method f that calls a method g. A speculative thread is spawned at the method call to g. The original non-speculative thread continues to execute the body of g, without any change in its speculative status. The new thread skips over the method call and starts execution at the point where g returns to the continuation of f. This new child thread is in a more speculative state than its parent spawner thread.

During the subsequent parallel execution of these two threads, if the parent writes to a memory location that has been read by the child, then we have a *data dependence violation*. The child speculation must be squashed, and the method continuation re-executed. On the other hand, if the parent thread completes the method call without causing any data dependence violations, then the spawnee can be committed. This means that its speculative actions can be confirmed to the whole system, and the spawnee is joined to the spawner. Execution resumes

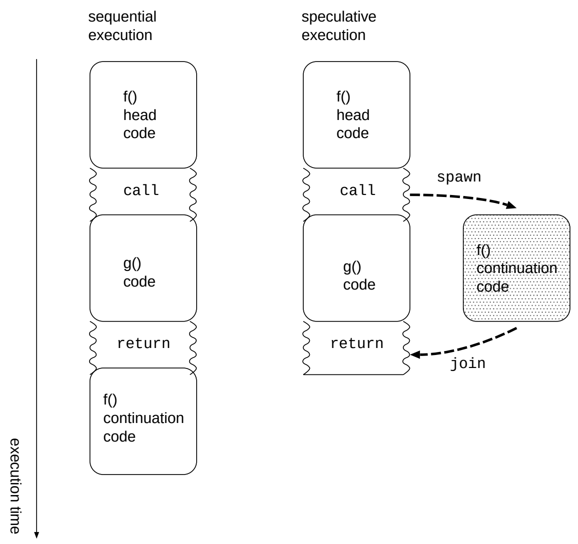


Fig. 1. Speculative execution model

from where the spawnee was at the join point, in the less speculative state of the spawner.

The SMLP model permits *in-order* nested speculation, which means that spawned threads can in turn spawn further threads at more speculative levels. However if a spawner thread itself has to be squashed, all its spawned threads must also be squashed.

Note that there are overheads for spawning new threads, committing speculative threads and squashing mis-speculations. Speculation must be carefully controlled to avoid excessive mis-speculation and the corresponding performance penalty. This motivates our interest in accurate *squash prediction* techniques.

In our architectural model, we make two idealized assumptions:

Write buffering: A speculative thread keeps its memory write actions private until the speculation is committed. This requires buffering of speculative state until the commit event. We assume buffers have *infinite size.* **Return value prediction:** If a method continuation (executing as a speculative thread) depends on the value of a method call (executing concurrently as a less speculative thread), then we assume that the return value may be predicted with *perfect accuracy.*

2.2 Benchmarks

This investigation uses Java applications from the SpecJVM98 [13] and DaCapo [14] benchmark suites, which are widely used in the research domain. Programs from different suites and genres are helpful to assess how our predictions generalize, for previously unseen data. An overview of the selected benchmarks is shown in Figure 2. We use `s1` inputs for SpecJVM98 and `small` inputs for DaCapo.

benchmark	*description*
_202_jess	AI problem solver
_205_raytrace	raytracing graphics
_213_javac	Java compiler
_222_mpegaudio	audio decoding
_228_jack	parser generator
antlr	parser generator
fop	PDF graphics renderer
pmd	Java bytecode analyser

Fig. 2. Benchmarks.

2.3 Trace-driven Simulation

All speculative execution is simulated using *trace-driven simulation.* Each sequential, single-threaded Java benchmark application executes with Jikes RVM v2.9.3 [15] in the Simics v3.0.31 full-system simulation environment [16]. Simics is configured for IA-32 Linux, using the supplied `tango` machine description, with a *perfect memory* model. The Jikes RVM compiler is instrumented to enable call-backs into Simics to record significant runtime events. These include method entry and exit, heap read and write, exception throw, etc. Each event is recorded with appropriate metadata, such as method identifier, memory address, and processor cycle count.

Thus we produce a sequential execution trace of events that may affect speculative execution. We feed the trace file to a custom TLS simulator. It uses method call information to drive speculative thread spawns, and heap memory access information to drive thread squashes based on data dependence violations. The timing information in the sequential trace enables the TLS simulator to determine method runlengths, in order to estimate an execution time based on parallel execution once it has determined which spawned threads commit or squash.

For the sake of simplicity, the TLS simulator only has two processors. Only two methods can execute in parallel. Methods are considered as candidates for spawning if their total sequential runlength is between 1000 and 10,000 cycles. If both available cores are occupied, then new speculations cannot be scheduled, i.e. the speculation is *in-order.* We impose a small, fixed 10 cycle overhead for each TLS spawn, squash and commit event during program execution.

All performance improvements in our simulated TLS system are due to *execution time overlap.* We do not model any secondary effects due to warming up caches and other architectural units. Other researchers quantify the benefit of this helper-thread effect of speculative execution, and find it to be a large component of the overall performance improvement [5] [17].

3 Intelligent Squash Prediction in TLS

3.1 Squash Prediction

Once we have fixed our TLS model so that spawns are only allowed to occur at method calls, the next task is to determine which potential spawn points we should ignore. Obviously we could spawn a new speculative thread at each method call; however many of these spawns will not lead to performance gain, either because the spawned method is too short for the parallelism to outweigh the overhead of thread creation, or because the spawned thread is guaranteed to cause a data dependence violation resulting in a squash.

It is sometimes possible to eliminate certain useless spawn points via static analysis. Other spawns are eliminated as a result of dynamic profiling that studies their behavior over a program run. In general, proposed TLS systems employ either or both of these techniques to eliminate poor spawn points. We aim to create a *hybrid* scheme, that can generate advice about spawn points based on features of program code (like *static* analysis) given prior knowledge of runtime behavior of that same, or similar, code (like *dynamic* analysis).

The relationship between static code features and dynamic squashing behavior is constructed using *machine learning* algorithms. Thus we are able to create general *squash predictors* that can provide advice for any code, whether previously seen or unseen. This is a key strength of learning-based techniques, which has not been exploited previously in the TLS domain.

Note that use of learning-based predictors does not prevent further runtime profiling to fine-tune spawn point advice dynamically. At present, we envisage offline learning for ahead-of-time predictions, as a drop-in replacement for static spawn point elimination. Previously such static elimination was based on ad-hoc compiler heuristics, whereas now we are proposing to apply well-understood learning techniques to enable more intelligent squash prediction.

3.2 Feature Collection

This study focuses entirely on *static* features, i.e. information about a program that can be gained from an inspection of its source or object code, without actually executing the program. We characterize each Java method by 45 features. 22 features are *fundamental nano-patterns*, described in [18] and [19]. These are binary properties of Java methods that can be easily extracted by trivial static analysis of the bytecode. The patterns denote a summary of the behavior and characteristics a method exhibits (such as array accesses and method calling relationships). Figure 3 outlines the patterns we collect.

We collect a further 23 integer-valued characteristics for each method, derived from the MILEPOST feature set [20]. These measurements are taken on the

Feature	Meaning
NoParams (N)	takes no arguments
NoReturn (V)	returns `void`
Recursive (R)	calls itself recursively
SameName (S)	calls another method with the same name
AbstractCaller (A)	issues calls via `abstract` methods
Leaf (L)	does not issue any method calls
ObjectCreator (OC)	creates `new` objects
ThisInstanceFieldReader (TFR)	reads field values from `this` object
ThisInstanceFieldWriter (TFW)	writes values to field of `this` object
OtherInstanceFieldReader (IFR)	reads field values from some object
OtherInstanceFieldWriter (IFW)	writes values to field of some object
StaticFieldReader (SFR)	reads static field values from a class
StaticFieldWriter (SFW)	writes values to static field of a class
TypeManipulator (TM)	uses type casts or instanceof operations
StraightLine (SL)	no branches in method body
Looping (LO)	one or more control flow loops in method body
Exceptions (E)	may throw an unhandled exception
LocalReader (LR)	reads values of local variables on stack frame
LocalWriter (LW)	writes values of local variables on stack frame
ArrayCreator (AC)	creates a new array
ArrayReader (AR)	reads values from an array
ArrayWriter (AW)	writes values to an array

Fig. 3. Java method-level features, based on fundamental nano-patterns

compiler intermediate form (in our case, Jikes RVM HIR code). They relate to the size and shape of the control flow graph, and the mixture of instruction kinds. Figure 4 outlines these features.

3.3 Learning Problem

We treat each method call in a dynamic execution trace as a potential TLS spawn point. At each spawn, two methods are involved: the caller and the callee. We characterize the spawn by a vector of 90 features: 45 from each method involved. The class to predict is a binary summary of the outcome of this speculation: commit or squash. We employ *supervised learning*, which means that we have a set of *labeled* samples to create the classification model. We use the C5.0 [21] algorithm to construct a rules-based classifier.

4 Evaluation

We use *leave-one-out cross-validation* (LOOCV) to evaluate our squash prediction classifiers. When we want to evaluate the squash predictor for benchmark b, we gather training data from the other benchmarks (excluding b) to generate

Feature	**Meaning**
bbNum	Number of basic blocks
bbOneSuc	Number of basic blocks with one successor
bbTwoSuc	Number of basic blocks with two successors
bbMtTwoSuc	Number of basic blocks with more than two successors
bbOnePred	Number of basic blocks with a single predecessor
bbTwoPred	Number of basic blocks with two predecessors
bbMtTwoPred	Number of basic blocks with more than two predecessors
bbOnePredOneSuc	Number of basic blocks with a single predecessor and a single successor
bbOnePredTwoSuc	Number of basic blocks with a single predecessor and two successors
bbTwoPredOneSuc	Number of basic blocks with two predecessors and a single successor
bbTwoPredTwoSuc	Number of basic blocks with two predecessors and two successors
bbMtTwoPredMtTwoSuc	Number of basic blocks with more than two predecessors and more than two successors
bbInstrLt15	Number of basic blocks with number of instructions less than 15
bbInstr15and500	Number of basic blocks with number of instructions in the interval [15,500]
bbInstrMt500	Number of basic blocks with number of instructions greater than 500
directCallNum	Number of direct calls in the method
cndBrnchNum	Number of conditional branches in the method
uncndBrnchNum	Number of unconditional branches in the method
methodInstrNum	Number of instructions in the method
bbInstrNum	Average number of instructions in basic blocks
loadNum	Number of memory load instructions in the method
storeNum	Number of memory store instructions in the method
allocNum	Number of memory-allocating instructions in the method

Fig. 4. Java method-level features obtained from the compiler intermediate form, based on the MILEPOST feature set

the classifier, which will then be applied to *b*. In this way, we test the generality of the squash prediction rules since they are always applied to previously unseen data.

The C5.0 algorithm generates a set of rules for predicting squashes and commits. Each rule has an associated *confidence* based on its accuracy in the training set. We select rules that predict squashes with a confidence above 90%, to apply in our testing rule set. Rules are incorporated directly into our TLS simulator via C header files. At each method call in the simulated execution, the rules are applied to check method properties and predict whether this potential spawn would result in a squash. If no squash is predicted and a free core is available, then the TLS spawn event occurs in the simulator. In an actual TLS system, such static rules would probably be encoded as static hints at compile time. A small amount of runtime support might be required for some dynamically dispatched methods.

Figure 5 compares three squash prediction policies for method-level speculation on a 2-core TLS system. For each benchmark, the left-most bar shows the speedup (over sequential execution) for no squash prediction. The middle bar shows the speedup for profile-based prediction, which involves determining the overall proportion of squashes at each callsite in the program, during a profiling run. Then during the test run, squashes are predicted for call sites that had more than 90% squashes on the profiling run. This is not machine learning—it is simple statistical analysis, training and testing on the same data. Finally, the right bar shows the speedup with learning-based prediction, using the squash prediction rules with at least 90% confidence, generated by C5.0 with LOOCV.

Note from Figure 5 that in the majority of cases, learning-based prediction is almost as good as, or better than, profile-based. However, for _205_raytrace and pmd, learning-based prediction is significantly worse. This may be because these benchmarks are unlike the others, e.g. _205_raytrace is significantly smaller than all the other execution traces. Perhaps a greater diversity in the training set would result in more generally applicable squash prediction rules.

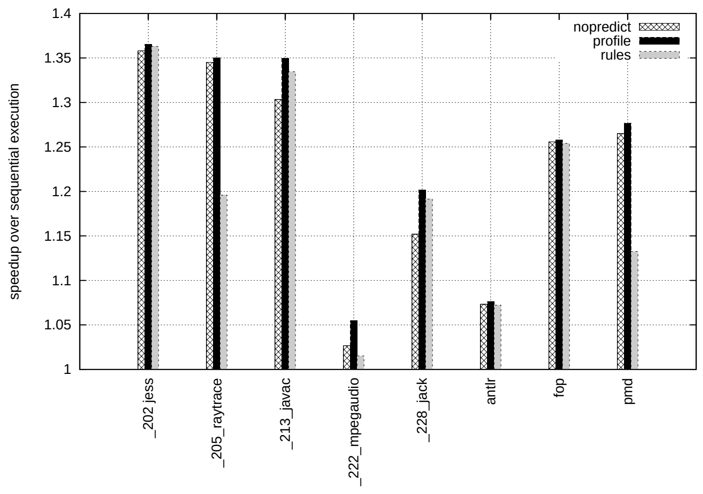


Fig. 5. TLS speedup for various squash prediction schemes

Figure 6 shows the frequencies of TLS spawn and squash events for each benchmark, with each squash prediction policy. For each benchmark, there are three bars: no prediction, profile-based squash prediction and rules-based squash prediction. For each prediction policy, the lower solid bar gives the number of squashes for a benchmark execution, and the upper shaded bar gives the number of thread spawns for a benchmark execution.

The desired impact of squash prediction is to reduce the number of squashes, thus eliminating wasted work. At the same time, the predictor must not have a high *false positive rate*, since this would suppress genuinely data-independent threads and decrease the parallel performance. Figure 6 shows that rules-based squash prediction reduces the number of squash events in relation to no prediction, in all cases. In the majority of cases, fewest squash events occur with profile-based prediction.

The detrimental effect of a high false positive rate can be seen with the pmd benchmark. The aggressive rules-based squash prediction suppresses many thread spawns, so much so that the number of spawned threads is around 50% of that for the other policies. This reduces the amount of available parallelism in the program execution, this reducing the speedup over the sequential version.

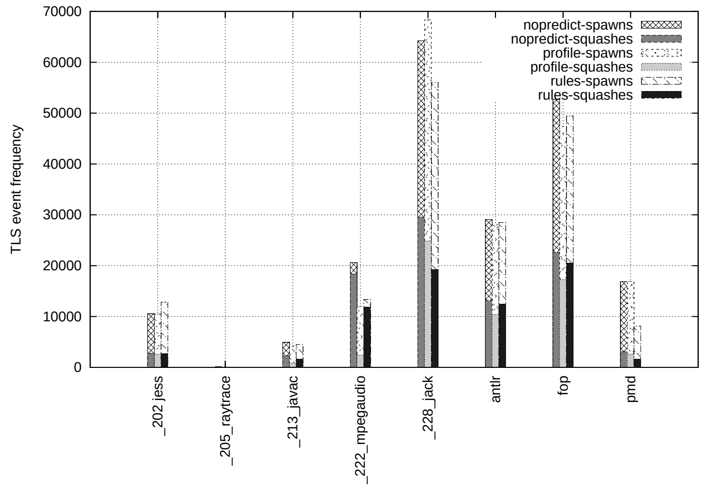


Fig. 6. Frequency of TLS thread spawn and squash events for each prediction policy, (solid bars are squashes, shaded bars are spawns)

5 Related Work

Whaley [4] presents a set of seven heuristics for controlling spawns in SMLP execution. His heuristics are manually generated from expert domain knowledge. The heuristics require dynamic information such as method runlengths and profile information. Thus programs require ahead-of-time profiling before heuristics can be applied. Our machine-learning based approach has the advantage that it can train on a set of programs, then be applied to a previously unseen program. Whaley gives oracle predictor speedup figures of around 1.5 on a set of Java benchmarks. His heuristics are able to achieve around 80% of this optimal speedup.

The POSH TLS compiler [5] uses program structure (loops and method calls) to identify potential spawn points. These spawn points are then *refined* by means of runtime profiling to eliminate useless spawns. The authors report a mean 1.3 speedup on standard benchmarks. They attribute 26% of the speedup to prefetching effects, which we do not consider.

Other researchers use similar profile feedback information to drive the compiler's decisions on spawn point insertion [22, 10]. In contrast, Dou and Cintra [23] present an entirely static cost model of TLS, which allows the compiler to estimate speedups in loop-level speculation with fairly high accuracy, and thus refine spawn insertions. They claim the framework gives a 25% speedup over a naive 'spawn everywhere' approach. There are also entirely *dynamic* squash prediction techniques. For instance, Cintra and Torrellas [24] present a hardware lookup table approach that learns which data dependencies are likely to cause thread squashes as the TLS program executes. Thus it is able to adapt dynamically the spawning policy for threads that eliminate many squashes. Our machine-learning based approach is currently intended as a static squash prediction technique, although we are investigating using learning at runtime for dynamic squash prediction.

Warg and Stenstrom [25] present another heuristic approach to improving SMLP. They observe that short methods should not be parallelized since the speculative overhead outweighs the benefit of parallel execution. They use a dynamic runlength predictor to identify short methods, and suppress spawns at these method calls. They use several different method runlength thresholds for spawn suppression, varying between 0 and 500 cycles. We use a small TLS event overhead of 10 cycles, which means the overhead is generally negligible in relation to the speculative runlength. (Other researchers adopt 10 cycles as a realistic cost - cites.) In fact, Warg and Stenstrom's problem is largely orthogonal to the squash prediction problem. We could apply similar learning techniques in both cases.

6 Conclusions

This paper has investigated the use of static Java program features for TLS squash prediction, using leave-one-out cross-validation to generate results for previously unseen programs. On the whole, the performance generated from rules-based squash prediction is not overly impressive. In several cases for our benchmarks, it would be better to spawn everywhere, rather than use the squash predictions. One possible inference is that our current set of static features may not *characterize the problem sufficiently* to enable accurate predictions. Alternatively, perhaps we have not selected enough programs to provide good *coverage in the training set* for LOOCV.

There are numerous parameters to tune in our TLS simulator. For instance, we have fixed overhead costs for squashes and thread sizes for consideration as spawn candidates. Additionally, there are parameters in our machine learning setup. We can vary the confidence threshold for selection of rules. We can select rules for both commit and squash events, and vote between them in the event of a disagreement. The irony is that our initial justification for machine learning was to avoid manually generated heuristics. However now we are intending *to tune parameters* for our customized learning algorithm, instead of for the underlying TLS squash prediction problem.

In future work, we should consider loop-level speculation too. We may be able to use features from existing machine learning studies on loop optimizations, e.g. [26]. We may also progress to consider dynamic features, such as read- and write-set sizes for speculative threads. This may tie in with the theme of *online learning*, integrated into the runtime system with low-overhead profiling and learning costs.

References

1. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium. (2002) 20–29
2. Cintra, M., Llanos, D.: Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In: Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming. (2003) 13–24
3. Quiñones, C., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.: Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In: Proceedings in Conference on Programming Language Design and Implementation. (2005) 269–279
4. Whaley, J., Kozyrakis, C.: Heuristics for Profile-Driven Method-Level Speculative Parallelization. In: Proceedings of the 34th International Conference on Parallel Processing. (2005) 147–156
5. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming. (2006) 158–167
6. Johnson, T., Eigenmann, R., Vijaykumar, T.: Speculative Thread Decomposition Through Empirical Optimization. In: Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming. (2007) 205–214
7. Luo, Y., Packirisamy, V., Hsu, W.C., Zhai, A., Mungre, N., Tarkas, A.: Dynamic Performance Tuning for Speculative Threads. In: Proceedings of the 36th Annual International Symposium on Computer Architecture. (2009) 462–473
8. Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., Sager, D., fook Ngai, T., Fang, J.: Dynamic Parallelization of Single-Threaded Binary Programs using Speculative Slicing. In: Proceedings of the 23rd International Conference on Supercomputing. (2009) 158–168
9. Hu, S., Bhargava, R., John, L.K.: The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism. Journal of Instruction-Level Parallelism **5** (2003) 1–21
10. Chen, M., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs. In: Proceedings of the 30th Annual International Symposium on Computer Architecture. (2003)
11. Warg, F., Stenström, P.: Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms. In: Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques. (2001) 221–230

12. Oplinger, J., Heine, D., Lam, M.: In Search of Speculative Thread-Level Parallelism. In: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques. (1999) 303–313
13. : Spec jvm98 http://www.spec.org/jvm98.
14. Blackburn, S., Garner, R., Hoffmann, C., Khang, A., McKinley, K., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, E., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., VonDincklage, D., Wiedermann, B.: The DaCapo Bbenchmarks: Java Benchmarking Development and Analysis. In: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. (2006) 169–190
15. Alpern, B., Attanasio, C., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., Hind, M., Hummel, S., Lieber, D., Litvinov, V., Mergen, M., Ngo, T., Russel, J., Sarkar, V., Serrano, M., Shepherd, J., Smith, A., Sreedhar, V., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM Systems Journal **39**(1) (2000) 211–238
16. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. IEEE Computer **35**(2) (2002) 50–58
17. Xekalakis, P., Ioannou, N., Cintra, M.: Combining thread level speculation helper threads and runahead execution. In: Proceedings of the 23rd International Conference on Supercomputing. (2009) 410–420
18. Singer, J., Pocock, A., Brown, G., Luján, M., Yiapanis, P.: Fundamental Nano-Patterns to Characterize and Classify Java Methods. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications. (2009) 204–218
19. Høst, E., Østvold, B.: The Programmer's Lexicon, Volume I: The Verbs. In: Proceedings of the 7th International Working Conference on Source Code Analysis and Manipulation. (2007) 193–202
20. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: MILEPOST GCC: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit. (2008)
21. Quinlan, R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc. (1993)
22. Wu, P., Kejariwal, A., Cascaval, C.: Compiler-Driven Dependence Profiling to Guide Program Parallelization. In: Proceedings of Workshop on Languages and Compilers for Parallel Computing. (2008) 232–248
23. Dou, J., Cintra, M.: A Compiler Cost Model for Speculative Parallelization. ACM Transactions on Architecture and Code Optimization **4**(2) (2007) 12
24. Cintra, M., Torrellas, J.: Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In: Proceedings of the 8th International Symposium on High Performance Computer Architecture. (2002) 43
25. Warg, F., Stenström, P.: Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. (2003) 12.2
26. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. (2009) 177–187

Smartlocks: Self-Aware Synchronization through Lock Acquisition Scheduling

Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal

Massachusetts Institute of Technology
{eastep, wingated, santambr, agarwal}@mit.edu

Abstract. As multicore processors become increasingly prevalent, system complexity is skyrocketing. The advent of the asymmetric multicore compounds this – it is no longer practical for an average programmer to balance the system constraints associated with today's multicores and worry about new problems like asymmetric partitioning and thread interference. Adaptive, or self-aware, computing has been proposed as one method to help application and system programmers confront this complexity. These systems take some of the burden off of programmers by monitoring themselves and optimizing or adapting to meet their goals.

This paper introduces an open-source self-aware synchronization library for multicores and asymmetric multicores called Smartlocks. Smartlocks is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning to optimize toward a user-defined goal, which may relate to performance, power, or other problem-specific criteria. Smartlocks builds upon adaptation techniques from prior work like *reactive locks* [1], but introduces a novel form of adaptation designed for asymmetric multicores that we term lock acquisition scheduling. Lock acquisition scheduling is optimizing which waiter will get the lock next for the best long-term effect when multiple threads (or processes) are spinning for a lock.

Our results demonstrate empirically that lock scheduling is important for asymmetric multicores and that Smartlocks significantly outperform conventional and reactive locks for asymmetries like dynamic variations in processor clock frequencies caused by thermal throttling events.

1 Introduction

As multicore processors become increasingly prevalent, system complexity is skyrocketing. It is no longer practical for an average programmer to balance all of the system constraints and produce an application or system service that performs well on a variety of machines, in a variety of situations. The advent of the asymmetric multicore is making things worse. Addressing the challenges of applying multicore to new domains and environments like cloud computing has proven difficult enough; programmers are not accustomed to reasoning about partitioning and thread interference in the context of performance asymmetry.

One increasingly prevalent approach to complexity management is the use of *self-aware* hardware and software. Self-aware systems take some of the burden off

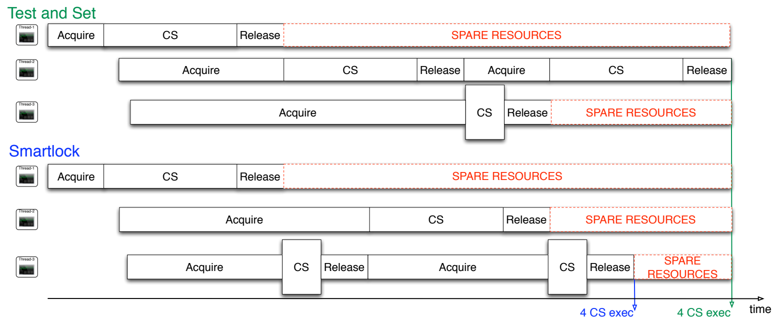


Fig. 1: The Impact of Lock Acquisition Scheduling on Asymmetric Multicores. Good scheduling finishes 4 critical sections sooner and creates more spare execution resources.

of programmers by monitoring themselves and optimizing or adapting to meet their goals. Interest in such systems has been increasingly recently. They have been variously called adaptive, self-tuning, self-optimizing, autonomic, and organic systems, and they have been applied to a broad range of systems, including embedded, real-time, desktop, server, and cloud systems.

This paper introduces an open-source[1] self-aware synchronization library for multicores and asymmetric multicores called Smartlocks. Smartlocks is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning. Smartlocks optimizes toward a user-defined goal (programmed using the Application Heartbeats framework [2]) which may relate to performance, power, problem-specific criteria, or combinations thereof.

Smartlock takes a different approach to adaptation than its closest predecessor, the *reactive lock* [1]. Reactive locks optimize performance by adapting to scale, i.e. selecting which lock algorithm to use based on how much lock contention there is. Smartlocks use this technique, but use an additional novel adaptation – designed explicitly for asymmetric multicores – that we term *lock acquisition scheduling*. When multiple threads or processes are spinning, lock acquisition scheduling is the procedure for determining who should get the lock next, to maximize long-term benefit.

One reason lock acquisition scheduling is an important power and performance opportunity is because performance asymmetries make cycles lost to spinning idly more expensive on faster cores. Figure 1 illustrates that, with significant asymmetries, the effect can be very pronounced. Using spin-locks can be broken up into three computation phases: acquiring the lock, executing the critical section (CS), then releasing the lock. The figure shows two scenarios where two slow threads and one fast thread contend for a spin-lock: the top uses a Test and Set lock not optimized for asymmetry, and the bottom shows what happens when Smartlocks is used. We assume that the memory system limits acquire and release but that the critical section executes faster on the faster core.

[1] The authors plan to release Smartlocks in January 2010.

In the top scenario, the lock naively picks the slower thread, causing the faster thread to idle in the acquire stage. In the bottom scenario, the Smartlock prioritizes the faster thread, minimizing its acquire time. The savings are put to use executing a critical section, and the threads complete 4 total critical sections sooner, giving a performance improvement. That improvement can be utilized as a latency improvement when a task requires executing some fixed number of critical sections, or it can be utilized as a throughput improvement (since the faster thread is able to execute another critical section sooner and more critical sections overall). The lock acquisition scheduler in the Smartlock scenario has the advantage that it can also (simultaneously or alternatively) optimize the total amount of spare execution cycles, which can be utilized for other computations or for saving power by sleeping cores.

We empirically evaluate Smartlocks on a related scenario in Section 4. We measure throughput for a benchmark based on a simple work-pool programming model (without work stealing) running on an asymmetric multicore where core clocks speeds vary due to two thermal throttling events. We compare Smartlocks to conventional locks and reactive locks, and show that Smartlock significantly outperforms other techniques, achieving near-optimal results.

The rest of this paper is organized as follows. Section 2 gives background about the historical development of various lock techniques and compares Smartlocks to related works. Section 3 describes the Smartlock implementation. Section 4 details the benchmarks and experimental setup referenced above. Finally, Section 5 concludes and identifies several additional domains in asymmetric multicore where Smartlock techniques may be applied.

2 Background and Related Work

This section begins with a basic description of spin-lock algorithms followed by the historical development of various algorithms and the challenges they solved. Then, this section compares Smartlocks to its most closely related works.

2.1 The anatomy of a Lock

Using spin-locks in applications can be thought of as executing a cycle of three computation phases: acquiring the lock, executing the application's critical section (CS), then releasing the lock. Depending on the algorithms used in its acquire and release phases, the spin-lock has three defining properties: its protocol, its waiting strategy, and its scheduling policy (summarized in Figure 2).

The protocol is the synchronization mechanism that the spin-lock uses to guarantee atomicity, or *mutual exclusion*, so that only one thread or process can hold a lock at any time. Typical mechanisms include global flags and counters and distributed queue data structures that locks manipulate using hardware-supported atomic instructions. The waiting strategy is the action that threads (or processes) take when they fail to acquire the lock. Typical wait strategies for spin-locks include spinning and backoff. Backoff is a spinning technique that

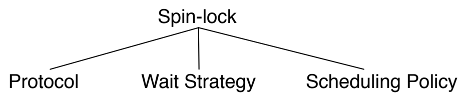


Fig. 2: Properties of a Spin-Lock Algorithm

systematically reduces the amount of polling. Locks other than spin-locks may use different strategies like blocking. Lastly, the scheduling policy is the strategy for determining which waiter should go next when threads are contending for a lock. Most lock algorithms have fixed scheduling policies that are intrinsic to their protocol mechanism. Typical policies include "Free-for-all" and "FIFO". Free-for-all refers to the unpredictability of which waiter will go next while FIFO refers to a first-come first-serve fair ordering.

The most well-known spin-lock algorithms include Test and Set (TAS), Test and Set with Exponential Backoff (TASEB), Ticket locks, MCS queue locks, and priority locks (PR Locks). Table 1 summarizes the protocol mechanisms and scheduling policy of these locks. In the next section, we will revisit the information in this table. The next section describes the historical evolution of various lock algorithms and the challenges that motivated them.

2.2 A Historical Perspective on Lock Algorithms

Because synchronization is such an important part of parallel programming, a wide variety of different algorithms exist. Two of the most basic are the Test and Set lock and the Ticket lock. Both algorithms have poor scaling performance when large numbers of threads or processes are contending for the lock. Inefficiency typically stems from degraded performance in the shared memory system when contention increases [3].

This led to an optimization on the Test and Set lock called Test and Set with Exponential Backoff that limits contention by systematically introducing wait periods between polls to the lock variable. Other efforts to improve performance scalability were based on distributed queues. Queue locks have the property that lock waiters spin on local variables which improves performance on cache-coherent shared memory systems [3]. Some popular queue-based spin-locks include the MCS lock [4] and MCS-variants such as the CLH lock [5]. Other works have since improved upon queue locks. The QOLB lock [6] improves performance by adding to the queue lock's local-spinning and queue-based techniques the techniques of collocation and synchronized prefetch.

One key deficiency of the queue-based algorithms is poor performance at small and medium scales due to the overhead of operations on the distributed queue. Smartlock gets better performance by dynamically switching to locks with low overhead when the scale is smaller. Table 1 summarizes the scalability of the various lock protocols. In a sense, research in scalable lock algorithms compliments the Smartlocks work because the Smartlock can use these base

Table 1: Summary of Lock Algorithms

Algorithms	Protocol Mechanism	Policy	Scalability	Target Scenario
TAS	Global Flag	Free-for-All	Not Scalable	Low Contention
TASEB	Global Flag	Randomizing Try-Retry	Not Scalable	Mid Contention
Ticket Lock	Two Global Counters	FIFO	Not Scalable	Mid Contention
MCS	Distributed Queue	FIFO	Scalable	High Contention
Priority Lock	Distributed Queue	Arbitrary	Scalable	Asymmetric Sharing Pattern
Reactive	Adaptive (not priority)	Adaptive (not arbitrary)	Scalable	Dynamic (not asymmetric)
Smartlock	Adaptive (w/ priority)	Adaptive (arbitrary)	Scalable	Dynamic (w/ asymmetry)

algorithms in its repertoire of dynamic implementations and leverage the benefits that any new algorithms bring.

Another key deficiency of the above algorithms is that they can perform poorly when the number of threads (or processes) exceeds the number of available cores, causing context switches. Problems arise when a running thread spins, waiting for action from another thread that is swapped out. As demonstrated in [7] these problems are especially bad for scalable distributed lock algorithms like queue locks. Lock strategies have been developed to optimize the interaction of locks and the OS Scheduler, including preemption-safe ticket locks and queue locks and scheduler-conscious queue locks [7]. These approaches could also be integrated into the Smartlock base.

Other orthogonal efforts to improve performance have developed special-purpose locks for some scenarios. The readers-writer lock is one example [8] that enables concurrent read access (with exclusive write access). Priority locks were developed for database applications where transactions have different importance [9]. Priority locks present many challenges such as priority inversion, starvation, and deadlock, and are a rich area of research [10]. NUCA-aware locks were developed to improve performance on NUCA memory systems [11].

These special-purpose locks are important predecessors to Smartlocks because they are some of the first examples of locks that hint at the benefits of lock acquisition scheduling. The write-biased reader-writer scheduling policy treats writers preferentially over readers. The priority lock explicitly supports prioritization of lock holders. The NUCA-aware lock uses a policy that prefers releasing locks to near neighbors for better memory system performance. Smartlock lock scheduling policies can emulate these policies (see Section 3.5 and Section 4.2).

The key advantage of Smartlock over these predecessors is that Smartlock is a one-size-fits-most solution that uses machine learning to automate the process of discovering good policies. Programmers can ignore the complexity of a.) identifying what a good scheduling policy would be and which special-purpose lock algorithm among dozens they should use or b.) figuring out how to program the priorities in a priority lock to make it do something useful for them while avoiding priority inversion, starvation, and deadlock.

The next section compares Smartlocks to other adaptive lock strategies.

2.3 Adaptive Locks

Various adaptive techniques have been proposed to design synchronization strategies that scale well across different systems, under a variety of dynamic conditions [1, 12, 13]. These are Smartlock's closest related works. A recent patch for Real-Time Linux modifies its kernel support for user-level locks so that locks adapt wait strategies between spinning and blocking: if locks are spinning for too long, the implementation switches to blocking. This is an adaptation that could be incorporated into Smartlocks. Other related works are orthogonal compiler-based techniques that build multiple versions of the code using different synchronization algorithms then periodically sample throughout execution, switching to the best as necessary [13]. Other techniques such as *reactive locks* [1] are library-based like Smartlocks. Library-based approaches have the advantage that they can be improved over time and have those improvements reflected in applications that dynamically link against them.

Like Smartlocks, reactive locks also perform better than the "scalable" lock algorithms at small and medium scales by dynamically adapting their internal algorithm to match the contention scale. The main difference is that Smartlock's strategies are optimized for asymmetric multicores: in addition to adapting for scale, Smartlocks uses a novel form of adaptation that we call lock acquisition scheduling. The scheduling policies of the various lock strategies are contrasted in Table 1. In Section 4.2, our results demonstrate empirically that lock acquisition scheduling is an important optimization for asymmetric multicores.

The Smartlock approach differs in another way: while prior work focused on performance optimization, Smartlock targets broader optimization goals which may also include power, latency, application-defined criteria, and combinations thereof. Whereas existing approaches attempt to infer performance indirectly from statistics internal to the lock library (such as the amount of lock contention), Smartlocks uses a direct measure provided by the programmer via the Heartbeats API[2] that more reliably captures application goals.

2.4 Machine Learning in Multicore

Recently, researchers have begun to realize that machine learning is a powerful tool for managing the complexity of multicore systems. So far, several important works have used machine learning to build a self-optimizing memory controller [14] and to coordinate management of interacting chip resources such as cache space, off-chip bandwidth, and the power budget [15]. Our insight is that machine learning can be applied to synchronization as well, and our results demonstrate that machine learning achieves near-optimal results for the benchmarks we study.

3 Smartlock

Smartlocks is a spin-lock library that adapts its internal implementation during application execution using heuristics and machine learning. Smartlocks opti-

[2] See Section 3.1 on annotating goals with the Heartbeats API.

Table 2: Smartlock API

Function Prototype	Description
`smartlock_helper::smartlock_helper()`	`Spawns Smartlock helper thread`
`smartlock_helper::~smartlock_helper()`	`Joins Smartlock helper thread`
`smartlock::smartlock(int max_lockers, hb* hb_ptr)`	`Instantiates a Smartlock`
`smartlock::~smartlock()`	`Destroys a Smartlock`
`void smartlock::acquire(int id)`	`Acquires the lock`
`void smartlock::release(int id)`	`Releases the lock`

mizes toward a user-defined goal (programmed using the Application Heartbeats framework [2]) which may relate to performance, power, problem-specific criteria, or combinations thereof. This section describes the Smartlocks API and how Smartlocks are integrated into applications, followed by an overview of the Smartlock design and details about each component. We conclude by describing the machine learning engine and its justification.

3.1 Programming Interface

Smartlocks is a C++ library for spin-lock synchronization and resource-sharing. The Smartlock API is similar to pthread mutexes, and applications link against Smartlocks just the way they do the pthreads library. Smartlocks supports off-the-shelf pthread apps, requiring trivial modifications to the source to use Smartlocks instead. While the underlying Smartlocks implementation is dynamic, the API abstracts the details and provides a consistent interface.

The API is summarized in Table 2. The first significant difference between the Smartlock and pthread API is that Smartlocks has a function to initialize the library.[3] This function spawns a thread for use by any Smartlocks that get instantiated. The next difference is that Smartlock initialization requires a pointer to a Heartbeats object so that the Smartlock can use the Heartbeats API [2] in the optimization process. The last difference is that Smartlock's acquire and release functions require a unique thread or process id (zero-indexed).[4]

Heartbeats is a generic, portable programming interface developed in [2] that applications use to indicate high-level goals and measure their performance or progress toward meeting them. The framework also enables external components such as system software or hardware to query an application's heartbeat performance, also called the *heart rate*. Goals may include throughput, power, latency, output quality, others, and combinations thereof.

Figure 3 part a) shows the interaction between Smartlocks, Heartbeats, and the application. The application instantiates a Heartbeat object and one or more Smartlocks. Each Smartlock is connected to the Heartbeat object which provides the reward signal that drives the lock's optimization process. The signal feeds into each lock's machine learning engine and heuristics which tune the lock's protocol, wait strategy, and lock scheduling policy to maximize the reward.

[3] This could be made unnecessary by auto-initializing upon the first Smartlock instantiation.
[4] This could be made unnecessary through the use of thread local storage or other techniques.

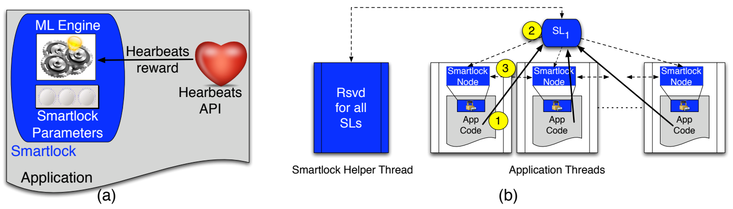


Fig. 3: a) Application-Smartlocks Interaction. Smartlock ML engine tunes Smartlock to maximize Heartbeat reward signal which encodes application's goals. b) Smartlock Implementation Architecture. Smartlock interface abstracts underlying distributed implementation and helper thread.

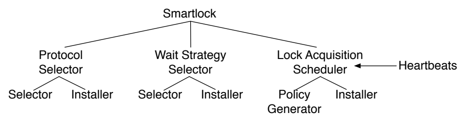


Fig. 4: Smartlock Functional Design. A component to optimize each aspect of a lock.

As illustrated in Figure 3 part b), Smartlocks are shared memory objects. All application threads acquire and release a Smartlock by going through a top-level wrapper that abstracts the internal distributed implementation of the Smartlock. Each application thread actually contains an internal Smartlock node that coordinates with other nodes for locking (and waiting and scheduling). Each Smartlock object has adaptation engines as well that run alongside application threads in a separate helper thread in a *smartlock_helper* object. Adaptation engines for each Smartlock get executed in a round-robin fashion.

3.2 Design Overview

As depicted in Figure 4, there are three major components to the Smartlock design: the Protocol Selector, the Wait Strategy Selector, and the Lock Acquisition Scheduler. Each corresponds to one of the general features of lock algorithms (see Section 2.1) and is responsible for the runtime adaptation of that feature.

3.3 Protocol Selector

The Protocol Selector is responsible for protocol adaptation within the Smartlock. Supported protocols include {TAS, TASEB, Ticket Lock, MCS, PR Lock} which each have different performance scaling characteristics. The performance of a given protocol depends upon its implementation and how much contention there is for the lock. The Protocol Selector tries to identify what scale of contention the Smartlock is experiencing and match the best protocol.

There are two major challenges to adapting protocols dynamically. The first is determining an algorithm for when to switch and what protocol to use. The second is ensuring correctness and good performance during protocol transitions.

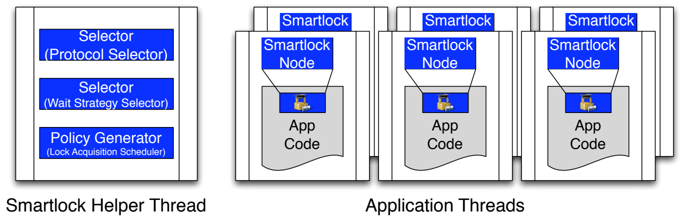


Fig. 5: Smartlock Helper Thread Architecture. Each Smartlock's adaptation components run decoupled from application threads in a separate helper thread.

As previously shown in Figure 4, the Protocol Selector has a component to address each problem: a Selector and an Installer. The Selector heuristically identifies what scale the Smartlock is experiencing and matches the best protocol, similar to the method in [1]: it measures lock contention and compares it against threshold regions empirically derived for the given host architecture and configuration. When contention deviates from the current region, the Selector initiates installation of a new protocol. Alternatively, the self-tuning approach in [16] could be used. As illustrated in Figure 5, the Selector executes in the Smartlock helper thread (described previously in Section 3.1).

The Installer installs a new protocol using a simple, standard technique called consensus objects. An explanation of consensus objects is outside the scope of this paper but is detailed in [1]. Protocol transitions have some built-in hysteresis to prevent thrashing.

3.4 Wait Strategy Selector

The Wait Strategy Selector adapts wait strategies. Supported strategies include {spinning, backoff}, and could be extended to include blocking or hybrid spinning-blocking. Designing the Wait Strategy Selector poses two challenges. The first is designing an algorithm to determine when to switch strategies. The second is ensuring correctness and reasonable performance during transitions. The Wait Strategy Selector has a component to address each of these problems: a Selector and an Installer.

Presently, the implementation of the Selector is null since each of Smartlock's currently supported protocols have fixed waiting algorithms. E.g. TASEB uses backoff and MCS and PR Lock use spinning. Eventually, the Wait Strategy Selector will run in the Smartlock's helper thread (described in Section 3.1) as illustrated in Figure 5. Careful handling of transitions in the wait strategy are still required and are again achieved through consensus objects.

3.5 Lock Acquisition Scheduler

The Lock Acquisition Scheduler is responsible for adapting the scheduling policy, which determines who should acquire a contended lock. As illustrated in Figure

4, the Lock Acquisition Scheduler consists of two components: the Policy Generator and the Installer. The Policy Generator uses machine learning to optimize the scheduling policy, and the Installer is responsible for smoothly coordinating transitions between policies.

Like the Protocol Selector and Wait Strategy Selector, designing a scheduler for lock acquisitions presents challenges. The first challenge is generating timely scheduling decisions, and the second is generating good scheduling decisions.

Since locks are typically held for less time than it takes to compute which waiter should go next, the scheduler adopts a decoupled architecture that does not pick every next lock holder. Rather, the scheduler works at a coarser granularity and enforces scheduling decisions through prioritization and priority locks. The scheduler works with the Smartlock's PR Lock protocol to schedule via continually adapting lock holder priority settings.

Figure 5 shows that the Lock Acquisition Scheduler's Policy Generator runs in the Smartlock's helper thread (see Section 3.1) and is decoupled from the Smartlock's lock protocols. The policy generator typically updates the policy every few lock acquisitions, independent of any Smartlock acquire and release operations driven by the application.

Smoothly handling decoupled policy updates requires no special efforts in the Installer. Transitions between policies are easy because only one of Smartlock's protocols, the PR Lock, supports non-fixed policies (see Table 1) and the PR Lock is implemented such that policies can be updated partially or incrementally (or even modified while a thread is in the wait pool) with no ill effects.

The Policy Generator addresses the second challenge (coming up with algorithms that yield good scheduling decisions) by using machine learning. While it is true that the asymmetric multicore optimizations this paper explores in Section 4 (adapting to dynamic changes in clock frequencies) could by achieved heuristically by reading off core clock frequencies and ranking processor priorities accordingly, Smartlock's machine learning approach has some advantages: heuristics are brittle and special-purpose whereas machine learning is a generic optimization strategy; good heuristics can be too difficult to develop when problems get really complex or dynamic; and finally, real-world problems often require co-optimizing for several problems simultaneously, and it is too difficult to come up with effective heuristics for multi-layered problems. Important recent work in coordinating management of interacting CMP chip resources has come to a similar conclusion [15].

3.6 Policy Generator Learning Engine

To adaptively prioritize contending threads, Smartlocks use a Reinforcement Learning (RL) algorithm which treats the heartbeat as a reward and attempts to maximize it. From the RL perspective, this presents a number of challenges: the state space is almost completely unobservable, state transitions are semi-Markov due to context switches, and the action space is exponentially large.

Because we need an algorithm that is a) fast enough for on-line use and b) can tolerate severe inobservability, we adopt an average reward optimality criterion

[17] and use policy gradients to learn a good policy [18]. Policy gradients are particularly well-suited for a number of reasons, one of which is action selection. At each update, the RL engine must select a priority ordering of the threads; n threads and k priority levels means k^n possible actions; other RL algorithms which require maximization over the action space are intractable.

The goal of policy gradients is to improve a policy by estimating the gradient of the average reward with respect to the policy parameters. The reward is taken to be the heartbeat rate, smoothed over a small window of time. Assume that we have access to some policy π, parameterized by θ. The average reward of $\pi(\cdot|\theta)$ is a function of θ: $\eta(\theta) \equiv \mathbb{E}\{\mathbb{R}\} = \lim_{t\to\infty} \frac{1}{t}\sum_{i=1}^{t} r_i$, where $\mathbb{R}$ is a random variable representing reward, and r_i is a particular reward at time i. The expectation is approximated with importance sampling, as follows (we omit the full derivation in the interests of space):

$$\nabla_\theta \eta(\theta) = \nabla_\theta \mathbb{E}\{\mathbb{R}\} \approx \frac{1}{N}\sum_{i=1}^{N} r_i \nabla_\theta \log \pi(a_i|\theta) \tag{1}$$

where the sequence of rewards r_i is obtained by executing the sequence of actions a_i sampled from $\pi(\cdot|\theta)$.

So far, we have said nothing about the particular form of the policy. We must address address the combinatorics of the naive action space, construct a stochastic policy which balances exploration and exploitation, and which can be smoothly parameterized to enable gradient-based learning.

We address all of these issues with a stochastic soft-max policy. We parameterize each thread i with a real valued weight θ_i, and then sample a complete priority ordering over threads – this relaxes a combinatorially large discrete action space into a continuous policy space. We first sample the highest-priority thread by sampling from $p(a_1) = \exp\{\theta_{a_1}\}/\sum_{j=1}^{n}\exp\{\theta_j\}$. We then sample the next-highest priority thread by removing the first thread and renormalizing the priorities. Let $\mathcal{S}$ be the set of all threads: $p(a_2|a_1) = \exp\{\theta_{a_2}\}/\sum_{j\in\mathcal{S}-\{a_1\}}\exp\{\theta_j\}$. We repeat this process $n-1$ times, until we have sampled a complete set of priorities. The overall likelihood of the policy is therefore:

$$\pi(a) = p(a_1)p(a_2|a_1)\cdots p(a_n|a_1,\cdots,a_{n-1}) = \prod_{i=1}^{n} \frac{\exp\{\theta_{a_i}\}}{\sum_{j\in\mathcal{S}-\{a_1,\cdots,a_{i-1}\}}\exp\{\theta_j\}}.$$

The gradient needed in Eq. 1 is easily computed. Let p_{ij} be the probability that thread i was selected to have priority j, with the convention that $p_{ij} = 0$ if the thread has already been selected to have a priority higher than j. Then the gradient for parameter i is simply $\nabla_{\theta_i} = 1 - \sum_j p_{ij}$.

When enough samples are collected (or some other gradient convergence test passes), we take a step in the gradient direction: $\theta = \theta + \alpha\nabla_\theta\eta(\theta)$, where α is a step-size parameter. Higher-order policy optimization methods, such as stochastic conjugate gradients [19] or Natural Actor Critic [20], would also be possible. Future work could explore these and other agents.

4 Experimental Results

We now illustrate how how Smartlocks can help solve heterogeneity problems by applying them to the problem of adapting to the variation in core clock frequencies caused by thermal throttling.

4.1 Experimental Setup

Our setup emulates an asymmetric multicore with six cores, where core frequencies are drawn from the set {3.16 GHz, 2.11 GHz}. The benchmark is synthetic, and represents a simple work-pile programming model (without work-stealing). The app uses pthreads for thread spawning and Smartlocks within the work-pile data structure. The app is compiled using `gcc` v.4.3.2. The benchmark uses 5 threads (reserving one for Smartlock use) consisting of the main thread and 4 workers. The main thread generates work while the workers pull work items from the queue and perform the work; each work item requires a constant number of cycles to complete. On the asymmetric multicore, workers will, in general, execute on cores running at different speeds; thus, x cycles on one core may take more wall-clock time to complete than on another core.

The experiment models an asymmetric multicore but runs on a homogeneous 8-core (dual quad core) Intel Xeon(r) X5460 CPU with 8 GB of DRAM running Debian Linux kernel version 2.6.26. In hardware, each core runs at its native 3.16 GHz frequency. Linux system tools like *cpufrequtils* could be used to dynamically manipulate hardware core frequencies, but our experiment instead models clock frequency asymmetry using a simpler yet powerful software method: adjusting the virtual performance of threads by manipulating the number of heartbeats. At each point where threads would ordinarily issue 1 beat, they instead issue 2 or 3, depending on whether they are emulating a 2.11 GHz or 3.16 GHz core. This artificially inflates the total heartbeats but preserves the pertinent asymmetry.

The experiment simulates a runtime environment by simulating two thermal-throttling events which change core speeds. No thread migration is assumed. Instead, the virtual performance of each thread is adjusted by adjusting heartbeats. The main thread always runs at 3.16 GHz. At any given time, 1 worker runs at 3.16 GHz and the others run at 2.11 GHz. The thermal throttling events change which worker is running at 3.16 GHz. The first event occurs at time 1.4s. The second occurs at time 2.7s and reverses the first event.

4.2 Results

Figure 6 shows several things. First, it shows the performance of the Smartlock against existing reactive lock techniques. At any given time, reactive lock performance is bounded by the performance of its highest performing internal algorithm; thus, this experiment models the reactive lock as a write-biased readers-writer lock (described in Section 2).[5] Smartlock is also compared against

[5] This is the highest performing algorithm for this problem known to the authors to be included in a reactive lock implementation.

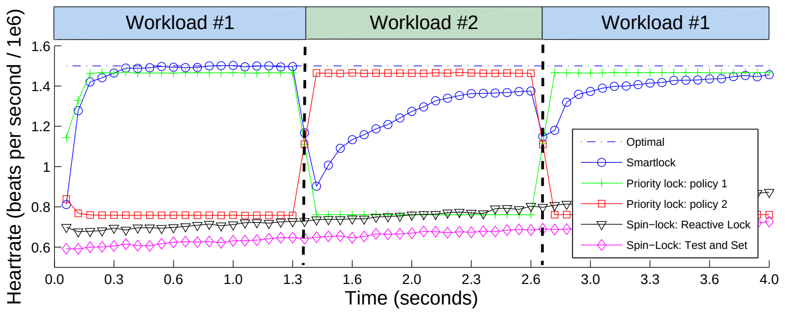


Fig. 6: Performance results on thermal throttling experiment. Smartlocks adapt to different workloads; no single static policy is optimal for all of the different conditions.

a baseline Test and Set spin-lock. The number of cycles required to perform each unit of work has been chosen so that the difference in acquire and release overheads between lock algorithms is negligible but so that lock contention is high; what *is* important is the policy intrinsic to the lock algorithm (and the adaptivity of the policy in the case of the Smartlock). Smartlock always outperforms the write-biased readers-writer lock and the Test and Set lock. This implies that reactive locks do not achieve optimal performance for this scenario.

The second thing that Figure 6 shows is that there is a gap between reactive lock performance and optimal theoretical performance. One lock algorithm / policy that can outperform standard techniques is the priority lock and prioritized access. The graph compares two priority locks with hand-coded priority settings against reactive locks. The first priority lock is optimal for two regions of the graph: from the beginning to the first throttling event and from the second throttling event to the end. Its policy sets the main thread and worker 0 to a high priority value and all other threads to a low priority value (e.g. high = 2.0, low = 1.0). The second priority lock is optimal for the region of the graph between the two throttling events; its policy sets the main thread and worker 3 to a high priority value and all other threads to a low priority value. Within the appropriate regions, the priority locks outperform the reactive lock, clearly demonstrating the gap between reactive lock performance and optimal theoretical performance.

The final thing that Figure 6 illustrates is that Smartlock approaches optimal performance and readily adapts to the two thermal throttling events. Within each region of the graph, Smartlock approaches the performance of the two hand-coded priority lock policies. Performance dips after the throttling events (time=1.4s and time=2.7s) but improves quickly.

Figure 7 shows the time-evolution of the Smartlock's internal weights $\wp_i$. Initially, threads all have the same weight, implying equal probability of being selected as high-priority threads. Between time 0 and the first event, Smartlock learns that the main thread and worker 0 should have higher priority, and uses a policy very similar to optimal hand-coded one. After the first event, the Smart-

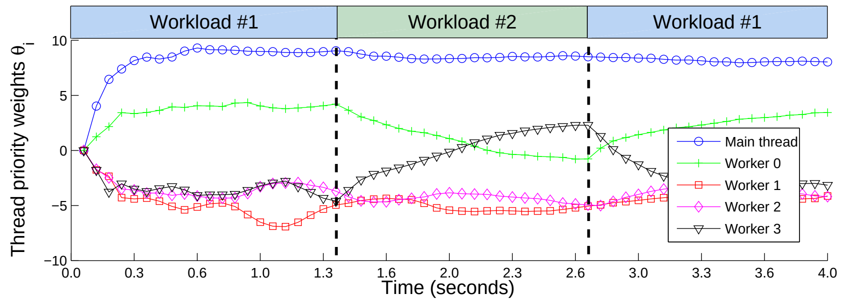


Fig. 7: Time evolution of weights in the lock acquisition scheduling policy.

lock learns that the priority of worker 0 should be decreased and the priority of worker 3 increased, again learning a policy similar to the optimal hand-coded one. After the second event, Smartlock learns a third optimal policy.

5 Conclusion

Smartlocks is a novel open-source synchronization library designed to remove some of the complexity of writing applications for multicores and asymmetric multicores. Smartlocks is a self-aware computing technology that adapts itself at runtime to help applications and system software meet their goals. This paper introduces a novel adaptation strategy ideal for asymmetric multicores that we term lock acquisition scheduling and demonstrates empirically that it can be applied to dynamic frequency variation. The results show that Smartlocks, owing to lock acquisition scheduling, can significantly outperform conventional locks and reactive locks, Smartlock's closest predecessor, on asymmetric multicores.

In the same way that atomic instructions act as building blocks to construct higher-level synchronization objects, Smartlocks can serve as an *adaptive* building block in many contexts, such as operating systems, libraries, system software, DB / webservers, and managed runtimes. Smartlocks can be applied to problems such as load-balancing and mitigating thread interference. For example, placing a Smartlock between applications and each DRAM port could learn an optimal partitioning of DRAM bandwidth; Smartlocks guarding disk accesses could learn adaptive read/write policies. In general, Smartlocks could mitigate many thread interference problems by giving applications a share of each resource and intelligently managing access.

Smartlocks may also be a key component in developing asymmetry-optimized parallel programming models. One strong candidate is a modification of the work-pool with work-stealing programming model that maintains the self scheduling property of work-stealing but optimizes by adapting the work-stealing heuristic; this can be accomplished by replacing the lock around the work-pool data structure and prioritizing access to each thread's work pool.

Smartlocks are, of course, not a silver bullet, but they do provide a foundation for many researchers to further investigate possible synergy between multicore programming and the power of adaptation and machine learning.

Bibliography

[1] Lim, B.H., Agarwal, A.: Reactive synchronization algorithms for multiprocessors. SIGOPS Oper. Syst. Rev. **28**(5) (1994) 25–35

[2] Hoffmann, H., Eastep, J., Santambrogio, M., Miller, J., Agarwal, A.: Application heartbeats for software performance and health. Technical Report MIT-CSAIL-TR-2009-035, MIT (Aug 2009)

[3] Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1) (1991) 21–65

[4] Mellor-Crummey, J.M., Scott, M.L.: Synchronization without contention. SIGARCH Comput. Archit. News **19**(2) (1991) 269–278

[5] Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. (Apr 1994) 165–171

[6] Kägi, A., Burger, D., Goodman, J.R.: Efficient synchronization: let them eat qolb. In: Proceedings of the 24th annual international symposium on Computer architecture, New York, NY, USA, ACM (1997) 170–180

[7] Kontothanassis, L.I., Wisniewski, R.W., Scott, M.L.: Scheduler-conscious synchronization. ACM Trans. Comput. Syst. **15**(1) (1997) 3–40

[8] Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proceedings of the 3rd ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (1991) 106–113

[9] Johnson, T., Harathi, K.: A prioritized multiprocessor spin lock. IEEE Trans. Parallel Distrib. Syst. **8**(9) (1997) 926–933

[10] Wang, C.D., Takada, H., Sakamura, K.: Priority inheritance spin locks for multiprocessor real-time systems. Parallel Architectures, Algorithms, and Networks, International Symposium on **0** (1996) 70

[11] Radović, Z., Hagersten, E.: Efficient synchronization for nonuniform communication architectures. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–13

[12] Karlin, A.R., Li, K., Manasse, M.S., Owicki, S.: Empirical studies of competitve spinning for a shared-memory multiprocessor. In: Proceedings of the thirteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (1991) 41–55

[13] Diniz, P.C., Rinard, M.C.: Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. ACM Trans. Comput. Syst. **17**(2) (1999) 89–132

[14] Ipek, E., Mutlu, O., Martínez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. In: Proc of the 35th International Symposium on Computer Architecture. (2008) 39–50

[15] Bitirgen, R., Ipek, E., Martinez, J.F.: Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2008) 318–329

[16] Hoai Ha, P., Papatriantafilou, M., Tsigas, P.: Reactive spin-locks: A self-tuning approach. In: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks, Washington, DC, USA, IEEE Computer Society (2005) 33–39

[17] Mahadevan, S.: Average reward reinforcement learning: Foundations, algorithms, and empirical results. Machine Learning **22** (1996) 159–196

[18] Williams, R.J.: Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, Northeastern University (1988)

[19] Schraudolph, N.N., Graepel, T.: Towards stochastic conjugate gradient methods. In: Proc. 9th Intl. Conf. Neural Information Processing. (2002) 853–856

[20] Peters, J., Vijayakumar, S., Schaal, S.: Natural actor-critic. In: European Conference on Machine Learning (ECML). (2005) 280–291