Finding representative sets of optimizations for adaptive multiversioning applications

Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, Grigori Fursin

Finding representative sets of optimizations for adaptive multiversioning applications*

Lianjie Luo[1,2], Yang Chen[1,2], Chengyong Wu[1], Shun Long[3], Grigori Fursin[4]

[1] Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate School of the Chinese Academy of Sciences, Beijing, China
{luolianjie, chenyang, cwu}@ict.ac.cn
[3] Department of Computer Science, JiNan University, Guangzhou, China
{tlongshun}@jnu.edu.cn
[4] INRIA Saclay, Orsay, France (HiPEAC Member)
{grigori.fursin}@inria.fr

Abstract. Iterative compilation is a widely adopted technique to optimize programs for different constraints such as performance, code size and power consumption in rapidly evolving hardware and software environments. However, in case of statically compiled programs, it is often restricted to optimizations for a specific dataset and may not be applicable to applications that exhibit different run-time behavior across program phases, multiple datasets or when executed in heterogeneous, reconfigurable and virtual environments. Several frameworks have been recently introduced to tackle these problems and enable run-time optimization and adaptation for statically compiled programs based on static function multiversioning and monitoring of online program behavior. In this article, we present a novel technique to select a minimal set of representative optimization variants (function versions) for such frameworks while avoiding performance loss across available datasets and code-size explosion. We developed a novel mapping mechanism using popular decision tree or rule induction based machine learning techniques to rapidly select best code versions at run-time based on dataset features and minimize selection overhead. These techniques enable creation of self-tuning static binaries or libraries adaptable to changing behavior and environments at run-time using staged compilation that do not require complex recompilation frameworks while effectively outperforming traditional single-version non-adaptable code.

1 Introduction

The past two decades have seen a rapid evolution of architectural designs and growth of their complexity. Modern compilers employ many advanced optimizations to achieve better performance on such architectures. However, they often

* This work was supported by a grant from the National Natural Science Foundation of China (No.60873057). This work was also partially supported by the MILEPOST project [4].

fail due to simplified hardware models used for static analysis and a lack of run-time information. Iterative compilation became a widely adopted technique to optimize programs for different constraints such as performance and code size without a priori knowledge of the underlying hardware [15, 10, 22, 37, 14, 27, 33, 24, 25, 3]. However, it is often used to optimize programs for a specific dataset which may not be practical as shown in [19] where an influence of multiple datasets on iterative compilation has been studied using a number of programs from MiBench benchmark suite.

Hybrid static/dynamic optimization approaches have been introduced to tackle those problems and allow compilers make better optimization decisions at run-time. Search-based methods have been adopted in several well-known library generators such as ATLAS [39], FFTW [30] and SPIRAL [34] to identify different optimization variants for different inputs that fit the computer architecture best at run-time. Some more general approaches have also been introduced in [11, 17, 38, 28] to make static programs adaptable at run-time by generating different code versions statically or dynamically and selecting them based on a given context, performance prediction or according to the changing run-time behavior. However, most of these frameworks are limited to only a few optimizations and do not have mechanisms to select a representative set of optimization variants. [20] presents a framework which creates adaptive binaries and statically enables run-time adaptation based on function multiversioning, iterative compilation and low-overhead hardware counters monitoring routines. It searches for complex combinations of optimizations in an off-line iterative manner. However, it is based on a reactive model and provides no pruning mechanism in order to avoid code size explosion.

This paper presents a novel approach to generate only a limited number of representative optimization variants across all datasets without performance loss or code-size explosion. It is based on finding good optimizations for hot program or library functions with different datasets using traditional off-line random iterative compilation in large optimization spaces and then iteratively pruning those variants while controlling overall performance and code size. When representative set of optimizations is found, we utilize several standard classification algorithms (decision trees or rule induction) to correlate some characteristics of the datasets with the best optimized function version. The learned decision trees or rules are then converted into executable code for runtime version selection. We evaluated our techniques using Open64 research compiler and plan to implement this framework inside GCC. However, hand-written optimization versions, libraries or versions generated using other optimization techniques or even compiled for different ISA (in virtual or heterogeneous environments) can be easily plugged into our framework.

The paper is organized as follows. Section 2 provides a motivation example for multiversioning and outlines the proposed framework. Section 3 describes a heuristic to find representative set of optimization versions while maximizing performance and minimizing code size. Section 4 evaluates different machine learning models to map some dataset characteristics to the selected versions

to maximize overall performance and minimize overheads. Section 5 summarizes related work in this area, before concluding remarks and future work in Section 6.

2 Static Multiversioning Framework to Enable Run-time Adaptation

2.1 Motivation

Some prior works show that different optimization combinations are needed for kernels or programs with multiple datasets [39, 30, 19]. We decided to confirm these findings using FFT benchmark with 15 different input sizes on a recent architecture such as dual-core AMD Opteron 2.6GHz with RedHat Linux AS4. We use Open64 4.0 compiler with the Interactive Compilation Interface [6] to apply combinations of fine-grain transformations such as loop tiling and unrolling with random parameters to the most time consuming loops in the kernel. We found 8 best optimization variants across those datasets. Figure 1 shows the speedups over -O3 optimization level of Open64 for these optimization variants, and how they vary across different datasets. It shows that the relative performance of a version can vary significantly on different input data, and no one single version can outperform all the other versions across all datasets. This motivates us to develop an automatic adaptive multiversioning technique which can select proper version based on a given runtime context. Moreover, having all 8 versions may not be practical due to considerable code-size increase and hence pruning technique is needed to select a small representative set of such optimizations, when given thresholds for tolerable code size increase and performance loss across all datasets.

2.2 Framework

The aim of our work can be formulated as a multi-objective problem as follows: given a set of semantically equivalent but differently optimized versions of a given program, kernel or library function (or compiled for different architecture in heterogeneous and virtual environments) for multiple input datasets, find the smallest subset of versions while maximizing overall performance (or reduce power consumption for example) and minimizing code size. When such set is found, use machine learning to build the mapping between dataset features (or run-time context such as hardware counters) and all versions in this representative set while minimizing the decision tree and hence run-time overhead using machine learning.

The framework implementing this approach is presented in Figure 2. It takes three steps to achieve these goals. First, we evaluate a large number of combinations of optimizations for a given program with multiple datasets using traditional iterative compilation techniques. This step can be considerably accelerated using techniques such as collective optimization [23, 2]. Then, we use a heuristic presented in Section 3 to select the representative versions. Finally, we

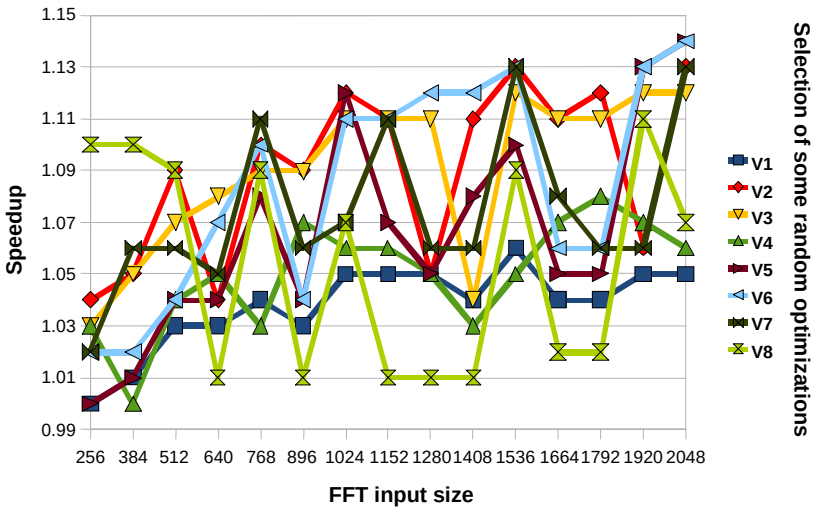

Fig. 1. *Speedup variation of 8 different optimization variants of FFT kernel for different input sizes on Opteron 2.6GHz machine using Open64 compiler V4.0 with ICI and with -O3 default optimization level*

build a model to map features of a program input to the representative versions using traditional machine learning techniques as described in Section 4. All representative versions with the resulted selection mechanism are statically linked into the final executable or library.

2.3 Adaptive Binaries and Libraries

When the representative set of versions for different run-time optimization cases is selected and the mapping function is prepared, we produce the final adaptive binary or library as shown in Figure 3. Such binaries or libraries include run-time routines for dataset/program/environment feature extraction and programs runtime behavior monitoring in order to select appropriate versions to improve performance, reduce power consumption or improve reliability, etc. Though compiled statically, this code is now adaptable to different datasets, run-time program and system behavior or even different heterogeneous, reconfigurable and virtual architectures.

3 Selection of Representative Optimization Versions

3.1 Experimental Setup and Iterative Compilation

Iterative compilation is traditionally performed using global compiler flags or source-to-source transformation tools which is not always satisfactory, particularly for function-level optimizations. Interactive Compilation Interface (ICI) has

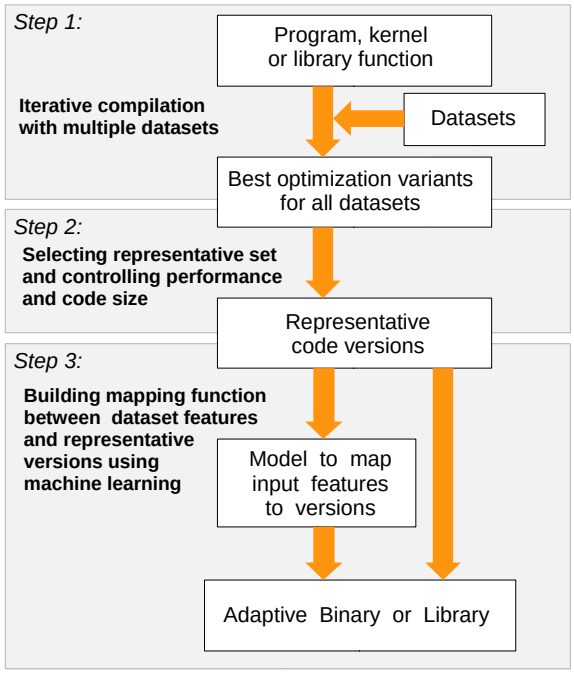

Fig. 2. *Static multiversioning framework to enable run-time optimization and adaptation while pruning number of versions, monitoring overall performance, reducing code size and finding mapping between dataset characteristics and representative versions*

been recently introduced [21, 6, 5] to enable fine-grain optimizations in production compilers with the ability to select different combinations, phase orders and parameters of available transformations. We decided to use Open64 4.0 compiler with ICI enabled [6] since it is a well-known research compiler with multiple aggressive optimizations available. We evaluated the following transformations using hill-climbing search similar to [22].

- loop tiling (2..512)
- register tiling (2..8)
- loop unrolling (2..16)
- loop vectorization
- loop interchange
- loop fusion
- array prefetching (8..128)

To validate our framework, we selected two widely used kernels: DGEMM from level-3 BLAS [1] library of NetLib, and FFT from UTDSP [7]. We evaluated 100 different combinations of optimizations on DGEMM and 280 on FFT (this

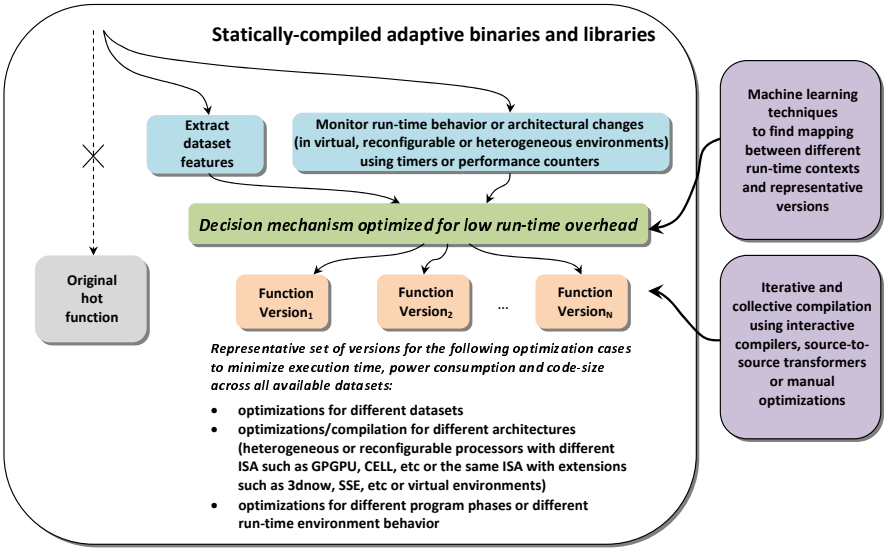

Fig. 3. *Adaptive binaries or libraries with a representative set of multiple function versions optimized or compiled for different run-time cases and with the decision tree or rule induction to map them with different run-time contexts*

number is program dependent). We randomly generated 1000 distinct datasets for BLAS and 280 for FFT with different input sizes and data values. All experiments are performed on a dual-core AMD Opteron at 2.6GHz, with 64KB L1 cache and 1MB L2 cache for each core, and 16GB memory, running RedHat AS4 (kernel 2.6.9-42).

3.2 Heuristic to Select the Representative Set of Optimizations

Given a potentially large number of combinations of optimizations (code versions) for a set of sample inputs, we would like to select only a minimal set of representative ones that obtain best performance on a maximum set of inputs. A heuristic algorithm for this is presented in Figure 4. Depending on the user's optimization priority (overall achieved performance vs maximum number of allowed versions to control code-size explosion), this algorithm tries to prune these versions and leaves only the representative ones that outperform the single version code for as many datasets as possible. To achieve this, it adopts a greedy strategy that is explained in Figure 4. And we plan to use multi-objective Pareto optimization for optimal performance/code size ratio as described in [24, 25] in the future work. Further more, we plan to extend the presented heuristic to take into account the mapping mechanism (described later in this paper) to be sure that all the representative versions can be effectively mapped with the dataset characteristics.

Input:

C = $\{ c^1, c^2, \ldots, c^O \}$ = a set of code versions (combinations of optimizations) to evaluate for a given program where O is the total number of versions to evaluate for a given program using iterative compilation

D = $\{ d^1, d^2, \ldots, d^X \}$ = a set of inputs (datasets) available for a given program where X is the total number of inputs available

P = threshold for possible performance loss for representative versions (percent)

V = threshold for the maximum allowed number of representative versions

Output:

M = minimal number of representative versions

C^R = $\{ c^{R1}, c^{R2}, \ldots, c^{RM} \}$ = a final set of representative versions to minimize performance loss

Heuristic:

1) Save a copy of D into D^{all}, because we need to change the content of D in the following steps for version selection but all the inputs are still needed for representative version set performance evaluation.
   Evaluate all versions from C across all datasets from D^{all} for a given program:
   $S = \{ s^{11}, s^{21}, \ldots, s^{X1}, s^{12}, s^{22}, s^{X2}, \ldots, s^{XO} \}$
   , where s is the speedup obtained over -O3.
   Calculate s^{max} which is the geometric mean of the best achievable speedups for each dataset from D^{all}. It is used as a reference for achievable performance across all datasets.

2) Find a code version c^i from C with a maximum geometric mean of speedups s^{Xi} across all datasets from D. Note, that if $s^{ij} < 1$, we set it to 1 to favor those optimizations that perform the best on a maximum number of datasets even if it may have a low performance on some others since we continue the search for optimization variants to obtain speedups across all datasets iteratively in this algorithm. We use geometric mean since speedups distribution is usually unknown.

3) Add c^i to the representative set of versions C^R and remove it from C.

4) Calculate s^{Rmax} which is a geometric mean of the best speedups for each dataset from D^{all} using versions from the representative set C^R. Remove all the datasets from D where c^i (from C^R) achieves the best speedup (or within some predefined threshold which can be a monitoring system routine precision, etc).

5) If the number of representative versions $M < V$ or $1 - (S^{Rmax} / S^{max}) > P / 100$ (note that it can be a Pareto optimal for two conditions at the same time or some other optimization scenario of a user) then iteratively continue from the step 2.

Fig. 4. *Algorithm to find a minimal representative set of versions that minimizes performance loss across all datasets*

Using our algorithm in Figure 4 we obtained 3 representative optimization versions for DGEMM and 4 for FFT. Evaluation of overall obtained performance and different overheads for these programs is presented in Sections 4.2 and 4.3 respectively.

4 Run-time Version Mapping Mechanism

4.1 Objective

To be able to statically create adaptive applications and libraries, we need an effective mechanism to select the representative optimization versions at run-time based on dataset characteristics or other run-time context such as dynamic feature vector of performance counters, information about process migration in multi-core heterogeneous architectures and virtual environments. Machine learning [31] has been effectively used to learn and build such mappings automatically. We evaluated some commonly used classification algorithms available in

the popular WEKA [8] machine learning suite that supporting multiple standard techniques such as clustering, classification, and regression.

All these algorithms vary in applicability and complexity depending on the problem encountered. In order to find one suitable for our version mapping mechanism, we decided to evaluate two widely adopted methods with several variants: direct classification (DC) and performance prediction model (PPM). Given a test case of a dataset, DC returns the most similar case from its prior experience (the training set), i.e. the optimization for another dataset most similar to the given one, expecting that the speedup will also be similar. On the contrary, PPM usually uses a probabilistic approach to correlate dataset features with available optimizations and speedups, uses probability distribution to suggest a set of good optimizations for a dataset before selecting the best out of these optimizations. Generally speaking, PPM performs better than DC, but at the extra cost in performance estimation, prediction and comparison. In our case, though the training cost can be tolerated , it is critical to link an optimized run-time decision tree to the adaptive binary or library in order to select appropriate versions online without considerable overhead. Six most commonly used heuristics for DC and PPM are evaluated, among which the best is selected.

It is vitally important for all machine learning techniques to find the suitable characterization of datasets in order to correlate dataset attributes with influential optimizations. It is a challenging task and beyond the scope of this paper. As a first step, we decided to use only the dimensions of input arrays since they are known to influence most of the transformations evaluated in this paper. There are other attributes that we plan to use in the future, such as the values of the entire array. However, it inevitably leads to a larger number of attributes to consider and may result in overfitting, while Li et al [29] suggested that the characteristics of input array elements may not be as important as the distribution of the values of them. Implicit attributes such as pointer type could also describe programs and library functions. However, if they point to an array, it may not be enough to learn from the value of the pointer itself. Therefore, we should consider high-level information about loops and array dimensions. We plan to combine all these characteristics with dynamic attributes such as performance counters, available hardware and software resources, system workload in the extension of this work. We can find or even generate as many features as possible and then automatically find the important ones using standard machine learning techniques such as Principle Component Analysis in order to keep the number of attributes low while maintaining the accuracy of learning and prediction.

4.2 Evaluation of Direct Classification vs Performance Prediction Model

Six different learning methods are adopted in DC [40, 8]:

- SMO - Support Vector Machine based
- J48 - decision tree based
- REPTree - decision tree based

– JRip - rule based
– PART - rule based
– Ridor - rule based

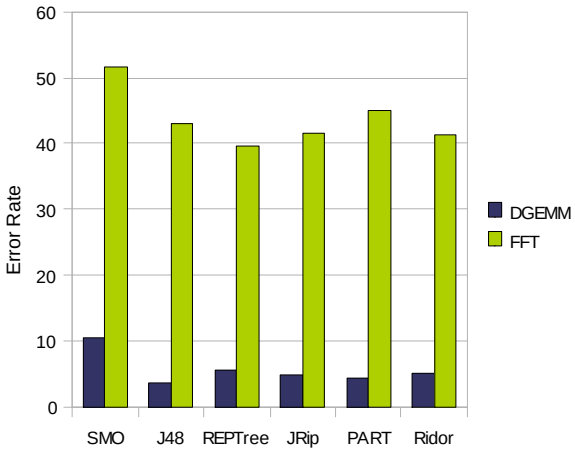

Fig. 5. Error rate of Direct Classification

We used a representative set of optimizations, a large number of datasets in our experiments, which were carried out in a ten-fold cross validation. We decided to use array dimension as the dataset characterization as mentioned earlier. Error rate is used as the performance evaluation metric for DC, and root relative squared error [40] for PPM which are standard metrics for these algorithms in WEKA.

Figure 5 shows that the classification accuracy depends on the given program and a machine learning method. J48 achieves the lowest error rate for DGEMM, while REPTree minimizes it for FFT. It is worth noting that the error rate of most of the classification algorithms have a very high error rate for FFT, more than 40% in most cases. This could be caused by a poor dataset characterization which needs further study in the future.

Six different learning methods are available for PPM:

– LeastMedSq - linear regression based
– LinearRegression - linear regression based
– PaceRegression - linear regression based
– SMOreg - Support Vector Machine based
– REPTree - decision tree based
– M5Rules - rule based

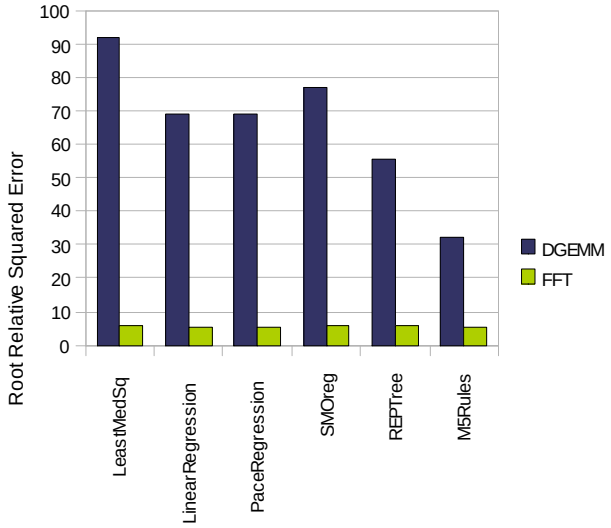

Fig. 6. Root relative squared error of Performance Prediction Model

Figure 6 demonstrates that M5Rules outperforms all other methods for both DGEMM and FFT. It is interesting to note that the best performing algorithms from DC and PPM are either decision tree or rule based which suggests that these methods suit our mapping objective best. We leave the detailed comparison of different algorithms for the future work.

Performance Evaluation of the Multiversioning Approach Once the best performing mapping algorithm is found, we can evaluate the mapping in a realistic environment by creating an adaptive binary or library linked with the selection function and representative optimization versions (3 versions for DGEMM, 4 for FFT). Then the produced binaries were executed with randomly generated test inputs (990 distinct inputs for DGEMM, 82 for FFT, none of them identical to the training data) on our experiment platform specified in section 3.1. Figure 7 summaries the performance results which include the dynamic version selection time (except for the estimated "ideal" case). The "ideal" numbers in Figure 7 for DGEMM and FFT refers to the estimated average speedup which could be achieved when the predication accuracy of the machine learned model for runtime version selection were 100%. It demonstrates by combining static multiversioning with dynamic version selection from 3 4 representative optimizations based on J48 and REPTree mapping mechanism using simple dataset features we can gain 98% of the available speedup.

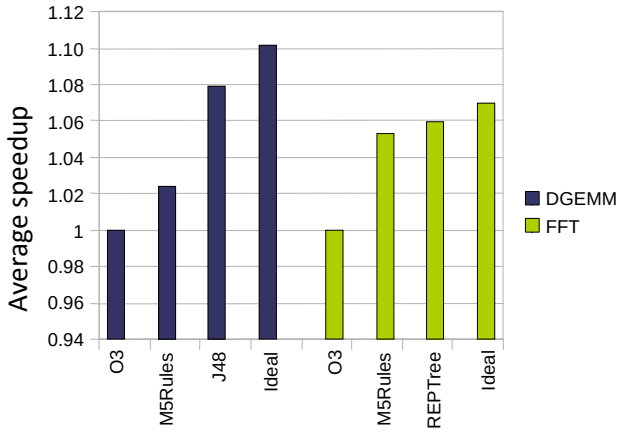

Fig. 7. Performance of DGEMM and FFT with multiversioning

4.3 Overheads for Code Size and Run-time Selection Mechanism

The introduction of multiple code versions in the binary or library as well as the run-time version selection inevitably results in code growth and run-time overhead. Table 1 presents the number of representative versions found for DGEMM and FFT. It demonstrates that the run-time overhead and the code growth for the version selection can be negligible whilst the overall code growth is not, due to multiversioning. However, depending on the user optimization scenarios, this overhead could be tolerable or reduced during the multi-objective tuning of the performance, representative optimization set and the overall code size. We believe that such an approach can, without a complex error-prone dynamic recompilation framework, automatically create static binaries and libraries which are adaptable to different behavior and environments at run-time.

	No of representative versions	Run-time selection overhead	Code size growth due to selection mechanism	Code size growth due to multiversioning
DGEMM	3	0.7%	1.8%	4.2%
FFT	4	0.5%	8.8%	76.5%

Table 1. Overheads of the static multiversioning approach

5 Related Work

Iterative compilation is an effective technique to optimize programs on a wide range of different architectures without a priori knowledge of the hardware/software

environment. It is performed in a feedback directed manner, i.e. the compiler's static optimization heuristics are replaced with an exploration of an optimization space , each step of which consists of program compilation, execution and search decision revision.

Iterative compilation has been widely used to optimize both kernels and larger programs on a given architecture [10, 26, 15, 16, 22, 27, 3, 14, 37, 18, 20, 33, 24, 25, 23] . Various optimization search spaces (composed of various parametric transformations and of different phase orders) are considered in order to minimize the execution time or code size. However it is a very time-consuming process which is unacceptable in many practical scenarios. To accelerate the iterative search process, most of the above works use some heuristics to focus optimizations. Machine learning techniques have been used recently to enable optimization knowledge reuse, predict good optimizations and speedup iterative compilation [32, 36, 35, 13, 41, 9, 12, 21], such techniques include genetic programming, supervised learning, decision trees, predictive modeling etc.

Iterative compilation is usually used to optimize program with one dataset which is not be practical. This is demonstrated in [19] where the influence of multiple datasets on iterative compilation has been studied using a number of programs from MiBench benchmark. Hybrid static/dynamic approaches have been introduced to tackle such problems. They are used in a well-known library generators such as ATLAS [39], FFTW [30] and SPIRAL [34] to identify different optimization variants for different inputs to improve overall execution time. Some general approaches have also been introduced in [11, 17, 38, 28] to make static programs adaptable to changes in run-time behavior by generating different code versions for different contexts. However, most of these frameworks are limited to simple optimizations or need complex run-time recompilation frameworks. None of them provide techniques to select a representative set of optimization variants.

Another hybrid static/dynamic framework has been introduced in [20, 23] to create self-tuning binaries. Run-time adaptation is achieved by first using offline iterative search for arbitrary combinations of available optimizations and then inserting into the static binary multiple versions of hot functions as well as low-overhead hardware counters monitoring routines.

None of the above techniques addresses the issue of automatic selection of a minimal representative set of optimizations for kernels or programs with multiple datasets in order to maximize overall performance and minimize code size explosion. The version selection mechanisms should be based on program input characteristics. This paper attempts to address these issues.We believe that this is an important practical step forward toward automatic creation of static self-tuning programs or libraries adaptable to different run-time behavior and environments automatically and without the help of a complex dynamic recompilation frameworks.

6 Conclusions and Future Work

This paper presented a static multiversioning approach with dynamic version selection which enables run-time optimizations based on iterative compilation, dataset characteristics and machine learning. It is capable of generating static binaries adaptive to different environments at run-time. We demonstrate that it is possible to effectively prune a large number of versions optimized for different datasets in order to build a representative set across available datasets. This is achieved without considerable performance loss nor code size explosion, which makes this approach practical. We also demonstrate how to use popular decision tree and rule induction classification algorithms to build an effective and low-overhead run-time mapping mechanism in order to correlate different datasets and optimized versions from the representative set.

Experimental results on several kernels demonstrate that our techniques can improve the overall performance of static programs or libraries with low run-time overhead. We plan to extend our algorithm to select representative set of optimizations not only based on performance but also taking into account both dataset characterization and possible run-time mapping at the same time. We will investigate the performance of different machine learning algorithms for run-time version mapping in detail and evaluate them for different multi-objective optimization scenarios. We plan to automate dataset and run-time feature generation in order to improve our version mapping technique. We believe that using staged compilation and self-tuning binaries can simplify automatic adaptation and optimization of the migrated code in virtual heterogeneous environments. Furthermore, we plan to combine our technique with collective optimization method [23, 2] and performance counters monitoring routines [20, 12] to evaluate it in a large number of heterogeneous, reconfigurable and virtual environments.

7 Acknowledgments

We would like to thank Yuanjie Huang, Mingjie Xing, Lujie Zhong, and Liang Peng for their help on the writing of the paper and implementation of the framework. We would like to thank Olivier Temam and Albert Cohen for fruitful discussions.

References

1. BLAS: Basic Linear Algebra Subprograms. `http://www.netlib.org/blas/index.html`.
2. CCC: Continuous Collective Compilation Framework. `http://cccpf.sourceforge.net`.
3. ESTO: Expert System for Tuning Optimizations. `http://www.haifa.ibm.com/projects/systems/cot/esto/index.html`.
4. EU Milepost project (MachIne Learning for Embedded PrOgramS opTimization). `http://www.milepost.eu`.

5. GCC ICI: Interactive Compilation Interface. http://gcc-ici.sourceforge.net.
6. Open64 ICI: Interactive Compilation Interface for Open64 research compiler. http://open64-ici.sourceforge.net.
7. UTDSP Benchmark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.
8. WEKA 3: The Waikato Environment for Knowledge Analysis. http://weka.wiki.sourceforge.net.
9. 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 *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.
10. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.
11. M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *In International Conf. on Parallel Processing*, pages pages 312–318, 1987.
12. J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2007.
13. J. Cavazos and J. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.
14. K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
15. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
16. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
17. P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, 1997.
18. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
19. G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, January 2007.
20. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.
21. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

22. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
23. G. Fursin and O. Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.
24. K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.
25. K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2008.
26. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.
27. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.
28. J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI)*, 2006.
29. X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 111, Washington, DC, USA, 2004. IEEE Computer Society.
30. F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.
31. T. Mitchel. *Machine Learning*. McGraw Hill, 1997.
32. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.
33. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.
34. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.
35. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.
36. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.
37. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
38. M. J. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on*

Principles and practices of parallel programming, pages 93–102, New York, NY, USA, 2001. ACM.

39. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.
40. I. Witten. *Data Mining*. Elsevier Science Inc., New York, NY, USA, 2005.
41. M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Proceedings of the Interational Conference on Code Generation and Optimization (CGO)*, pages 317–327, 2005.