Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation

Grigori Fursin Cupertino Miranda Sebastian Pop Albert Cohen
Olivier Temam

Alchemy group, INRIA Futurs and LRI, Paris-Sud 11 University, Orsay, France
HiPEAC members
firstname.lastname@inria.fr

Abstract

Iterative feedback-directed optimization is now a popular technique to obtain better performance and code size improvements for statically compiled programs over the default settings in a compiler. The offline evaluation of multiple optimization strategies for a given program is a potentially costly operation. The number of iterations typically grows with the complexity of the program transformation search space, and with the number of input datasets used for performance assessment. In addition, as the behavior of a program can vary considerably across different datasets, it is often preferable to generate different optimization versions, covering the full spectrum of the program's representative datasets.

Continuous and collective optimization are targeted at these issues. Continuous optimization searches for the best program transformation at run-time, taking advantages of the phase behavior of programs to evaluate multiple optimization versions within a single run, and dynamically adapting to changing execution contexts. Collective optimization interleaves optimization iterations with program executions along the lifetime of the program. In both cases, the user expects the optimization process to learn from the past execution contexts and program behavior. The user also assumes the system will be fully transparent, take negligible overhead for the incremental profiling, learning, decision and code generation steps, while bringing significant performance benefits over the lifetime of the program.

In order to explore multiple optimization options, we propose a simple and practical solution based on cloning of all procedures, applying any complex optimizations to these clones and randomly selecting either original or transformed procedures at run-time. Obtaining execution time distribution among original and cloned procedures, we can statistically determine the influence of compiler optimizations on the code in a single run. The simplicity of the implementation makes this technique reliable, secure and easy to debug. Yet it enables practical transparent low-overhead continuous optimizations for programs statically compiled with GCC while avoiding complex dynamic recompilation frameworks. In addition, our framework can enable program self-adaptation at fine-grain level for different environments such as parallel heterogeneous and reconfigurable systems with different ISA, and for different constraints such as performance, code size and power consumption.

1 Introduction

Static compilation often fails to deliver fastest program on modern architectures due to a large number of possible optimizations, simplistic models of rapidly evolving hardware, lack of run-time information and inability to dynamically adapt to changes in program and/or system behavior. Recently, iterative compilation became a popular approach to search for the best selection of compiler transformations that optimizes a program for different objective functions such as program performance, code size and power consumption on a particular architecture and for a particular workload (dataset) [10, 26, 15, 16, 14, 28, 22, 43, 41, 18, 37, 21, 24, 34, 17, 35, 36, 25, 9, 20, 27, 11]. Several configurable tools are already provided for the compiler suites to search for the best combination of compiler flags [7, 1, 3]. Iterative optimization has also been employed in well-known library generators in such systems

as ATLAS [46], FFTW [32] and SPIRAL [38] to tune parameters of various transformations to get best performance on a targeted platform. However, in most of the cases, practical applicability of iterative compilation is limited due to often intolerable excessive overhead of compiling and running program multiple times to evaluate every optimization setting and inability to reuse optimization knowledge between different programs, architectures and workloads.

Machine learning has been recently introduced to tackle the problem of growing complexity of the systems, compilers and optimizations, speed up the search for the best optimizations, and enable optimization knowledge reuse among different programs [33, 41, 40, 12, 47, 9, 11]. However, current techniques are still limited by the large amount of off-line runs needed to train the model and inability to dynamically adapt to changing environment for statically compiled programs. Some of these issues can be solved in dynamic compilation environments but often with a cost of shipping complex, resource-hungry or limited in their abilities run-time recompilation frameworks [45, 44, 13, 31, 29, 42].

To solve these issues and make iterative compilation practical for statically compiled programs, we started developing a Continuous Collective Compilation framework [2]. This framework is based on previous research on dynamic selection of complex compiler optimizations during stable program phases using static multi-versioning [21, 30, 20] and on analysis of the sensitivity of compiler optimizations towards different datasets [19, 6]. To enable transparent collection of statistics without the need to run programs a number of times with the same dataset for detecting the base line performance, we clone and optimize all or most time-consuming subroutines during compilation. For assessing the average influence of program optimizations in one run, we randomly select at run-time the version of the function to be executed. We continuously collect the distribution of time spent in both the original and the cloned and optimized parts of the program. This will allow various users to participate in the collection of the global optimizations statistics for a large number of statically compiled programs transparently and continuously. We implemented this technique in GCC which is well suited for this task—it is free software and has a large user base.

Our technique can now be used to collect optimization statistics from all users based on their applications and workloads instead of selecting some representative benchmarks, datasets and running them multiple numbers of times for each individual user and architecture. This data will be used later to build optimization models and enable global optimization knowledge reuse to improve both programs and compiler optimization heuristic (pool of best default optimization flags for example). It is based on our research on machine learning and statistical techniques to narrow down the optimization search space and quickly find best compiler flags using program static features and dynamic characteristics (hardware counters) [17, 9, 11]. We believe that this framework can considerably improve the program optimization process and automatic tuning of a compiler heuristic for new architectures using machine learning and statistical techniques.

2 Cloning and run-time version selection

Finding best compiler settings for a program, building an optimization model using machine learning or statistical techniques and improving compiler optimization heuristic are tedious and time consuming tasks that require a large number of runs of the program. It could be speeded up considerably if the information from multiple users about program optimizations could be shared transparently, however it is currently problematic for statically compiled programs since all users have to execute the same code with the same dataset multiple times to be able to compare the effect of different optimizations on the code as shown in Figure 1(a,b). Therefore, to enable transparent Continuous Collective Compilation, we developed a technique within GCC that clones all or most time-consuming code sections (procedures or functions) of the program, applying different compiler optimizations for the clones, and randomly selecting at run-time, for each invocation, either the original code section with the baseline optimization or the clone, as shown in Figure 1(c).

We use `gprof` to collect the informations about the total time `tt` spent in a function, and the number of calls `nc` to that function. We normalize the time spent in the function by computing the average time $\texttt{avt} = \frac{\texttt{tt}}{\texttt{nc}}$ spent in the function.

In order to determine the effect of the tested transformations, we compute the speedup of the cloned function with respect to the original function, as: $\texttt{s} = \frac{\texttt{avt}_{\texttt{original}}}{\texttt{avt}_{\texttt{cloned}}}$.

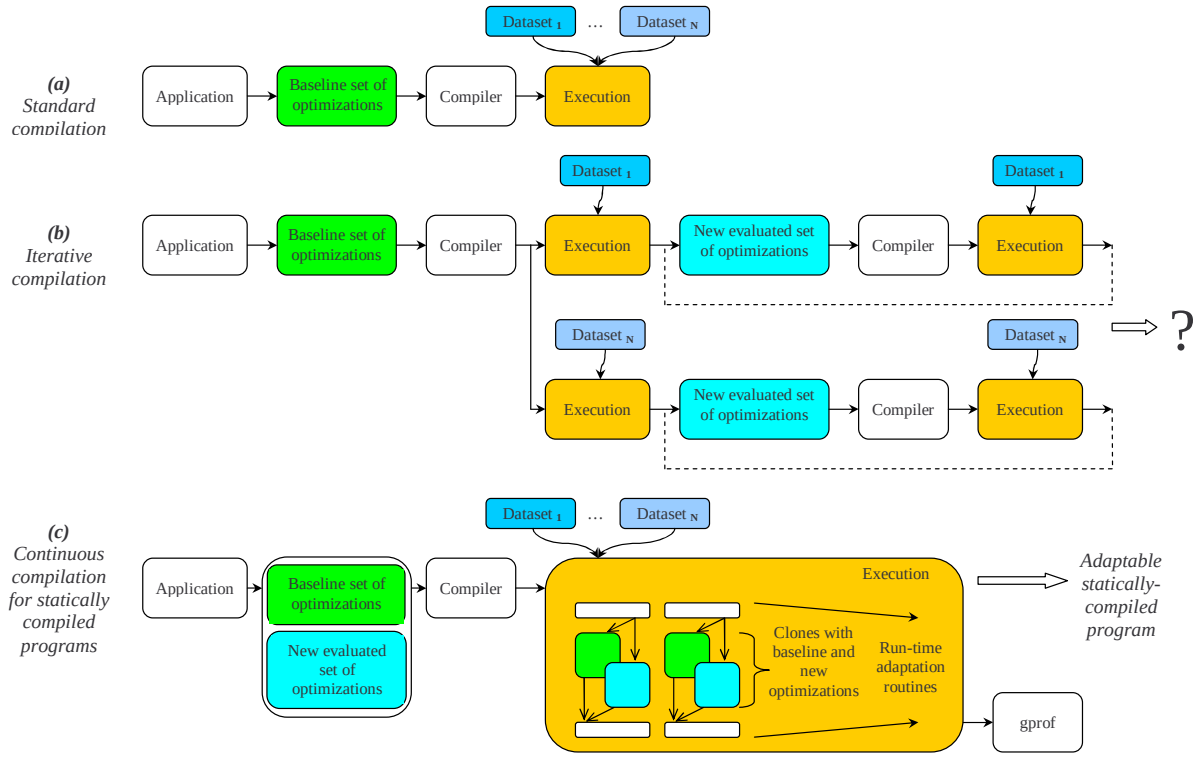

Figure 1: Comparison of standard, iterative and continuous compilation techniques

In the statistical evaluation framework, several runs of the function lead to a sequence of evaluations of the speedup $s_1, \ldots, s_n$. It is possible to compute from this information the expected speedup value $e = \sum_{i=1}^{n} s_i/n$, the variance $v = \sum_{i=1}^{n} (s_i - e)^2$, and infer from this the dependence of the code transformation on the processed data.

Naturally, since the input data vary across calls to the original or cloned subroutines, we continuously monitor the variance v to detect the convergence across executions. If there is no convergence, this data is currently skipped unless more complex techniques are used (such as in [21] to detect stable program phases, for example). Despite this limitation, obtaining data from a large number of users should be already sufficient to use machine learning or statistical techniques to search for the best optimizations or improve default compiler heuristic.

3 Implementation

We describe the minimal implementation that we realized for prototyping this system, then we propose improvements to this prototype to make the user interface more flexible for a potential integration in GCC.

3.1 Prototype

The prototype implementation in GCC contains two parsers for the set of functions to be cloned, and for the set of options to be applied to the cloned functions. These two sets are provided by the programmer in environment variables `GCC_ADAPT_OPT` and `GCC_ADAPT_FUNCS`.

The function cloning is based on the infrastructure for function versioning also used in the interprocedural constant propagation. The functions created by the function versioning are automatically named, making the task of recognizing the link between the original and the cloned functions difficult. We rename the newly created functions based on the original function name and appending a suffix `.cloned`, such that the programmer can simply recognize the original functions and their cloned versions in the output of function level profiling tools, such as `gprof`.

After having created the cloned version, we transform the original function as illustrated in Figure 2. At the beginning of the original function we insert a call to a selection function that determines which version of the

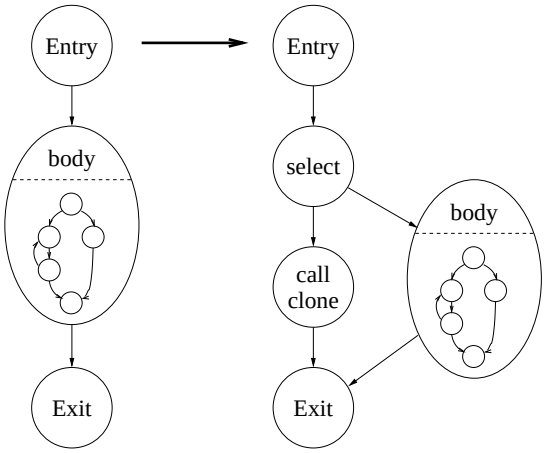

Figure 2: Transforming a function body by inserting a condition expression "select," the call to the cloned function "call clone," and the original code contained in the function body.

code should be executed. In the minimal implementation of this library, that we used for the experiments, we setup a random number generator and the selection function produces a random boolean result.

In the advanced implementation, the programmer can provide its own selection function in an external library that has to be linked in with the program to enable various customizable run-time adaptation techniques. This external library contains two other functions for the initialization and finalization of the structures used in the selection function.

```
unsigned int gcc_adapt_select (void);
void gcc_adapt_init (void);
void gcc_adapt_fini (void);
```

The advanced implementation will also make it possible to specify different optimization flags for different functions by using an external configuration file. This configuration file will have a syntax close to the declarative syntax of makefiles:

```
<rules> ::= <rule>
          | <rule> <newline> <rules>
<rule> ::= <fns> : <flags>
<fns> ::= <fn_name>
            | <fn_name> <fns>
<flags> ::= <flag>
          | <flag> <flags>
```

4 Experiments and Usage Scenarios

We implemented our technique in the GCC 4 series and performed experiments on the Dell Precision 390 N-Series Server with Intel Core 2 Duo processor E6300 (1.86 Ghz/ 2 MB Cache / 1066Mhz FSB) and Mandriva Linux 2006. To demonstrate the use of our technique we selected two well-known programs: `mgrid` from the SPEC2006 benchmark suite [39] and `jpeg_encoder` from MiBench benchmark suite [23] that has several most time-consuming subroutines with varying run-time input data.

First, we ran unmodified `mgrid` with ref and train datasets, and `jpeg_encoder` with small and large inputs. We selected three optimization levels (-O1,-O2,-O3) and show their respective speedups with respect to the default GCC optimization level in Table 1.

Later, we select two most time-consuming procedures for each benchmark, clone them and compile with different optimization levels. After executing each benchmark 10 times (5 with train dataset and 5 with ref dataset), we obtain the time distribution among original and cloned procedures and the number of calls to each procedure using `gprof`. Table 2 shows the speedups of the cloned procedures with respect to the original ones (normalized with the number of procedure calls), expected speedup values and the variance.

Finally, Figure 3 compares the speedups of the unmodified programs with the speedups of the cloned procedures for three optimization levels. These experiments

Application, dataset	speedup (-O1)	speedup (-O2)	speedup (-O3)
`mgrid, train dataset`	2.75	3.64	3.62
`mgrid, ref dataset`	2.64	3.42	3.40
`jpeg_encoder, small dataset`	1.66	1.70	1.71
`jpeg_encoder, large dataset`	1.56	1.54	1.60

Table 1: Speedups for three optimization levels with respect to the default compiler optimization level for the selected unmodified benchmarks

Application, procedure	expected speedup (-O1),v	expected speedup (-O2),v	expected speedup (-O3),v
`mgrid, proc1`	2.38, 0.21	3.27, 0.32	3.35. 0.33
`mgrid, proc2`	2.25, 0.30	3.20, 0.39	3.24, 0.37
`jpeg_encoder, proc1`	1.92, 0.16	1.98, 0.14	1.95, 0.17
`jpeg_encoder, proc2`	1.12, 0.09	1.25, 0.11	1.27, 0.09

Table 2: Expected speedups for three optimization levels with respect to the default compiler optimization level for the selected benchmarks with cloned procedures and their variance

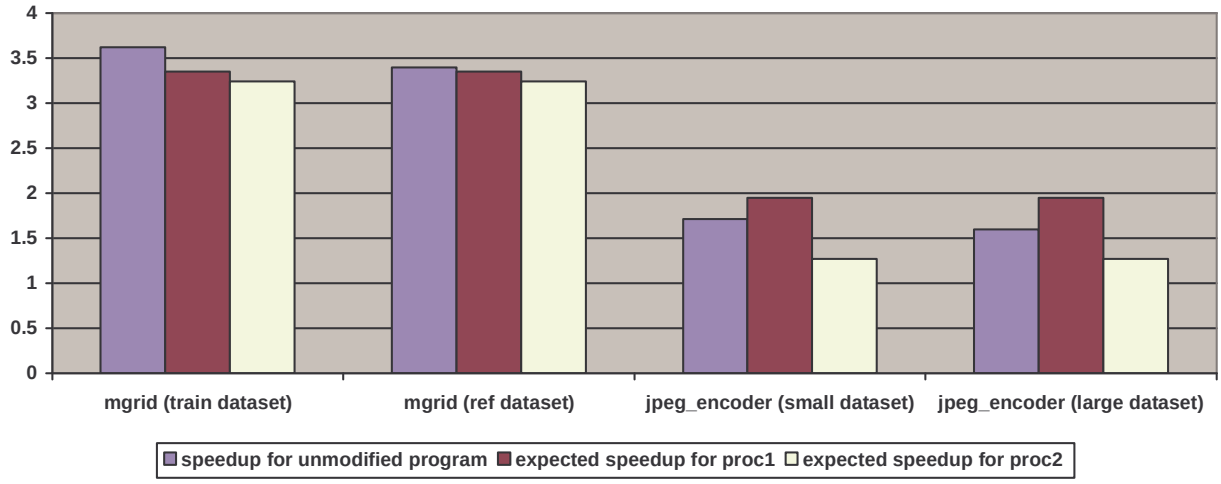

Figure 3: Comparison of the speedups of unmodified programs with the expected speedups of the cloned procedures for -O3 optimization level

show that we can effectively evaluate the influence of different compiler optimizations on various code sections of statically compiled programs at run-time and use that information to improve performance. As mentioned earlier, this framework can be used for multiple purposes and one of them is Continuous Collective Compilation. This optimization approach is similar to the common local iterative feedback-directed compilation except that it will exchange information at each optimization step with the global server that keeps and reuses optimization information collected transparently from multiple users. Based on the server data, it will suggest the best optimizations for a specific program using machine learning and both static and dynamic (hardware counters) features [9, 19] as shown in Figure 4.

5 Conclusions and Future Work

In this article we present the technique we implemented in GCC that allows to determine the influence of compiler optimizations on statically compiled programs during continuous runs with any datasets and improve performance. This technique eliminates the need for additional program runs with the same inputs only to obtain the baseline performance. It is achieved through cloning of all or most time-consuming code sections statically and applying evaluated optimizations on these clones.

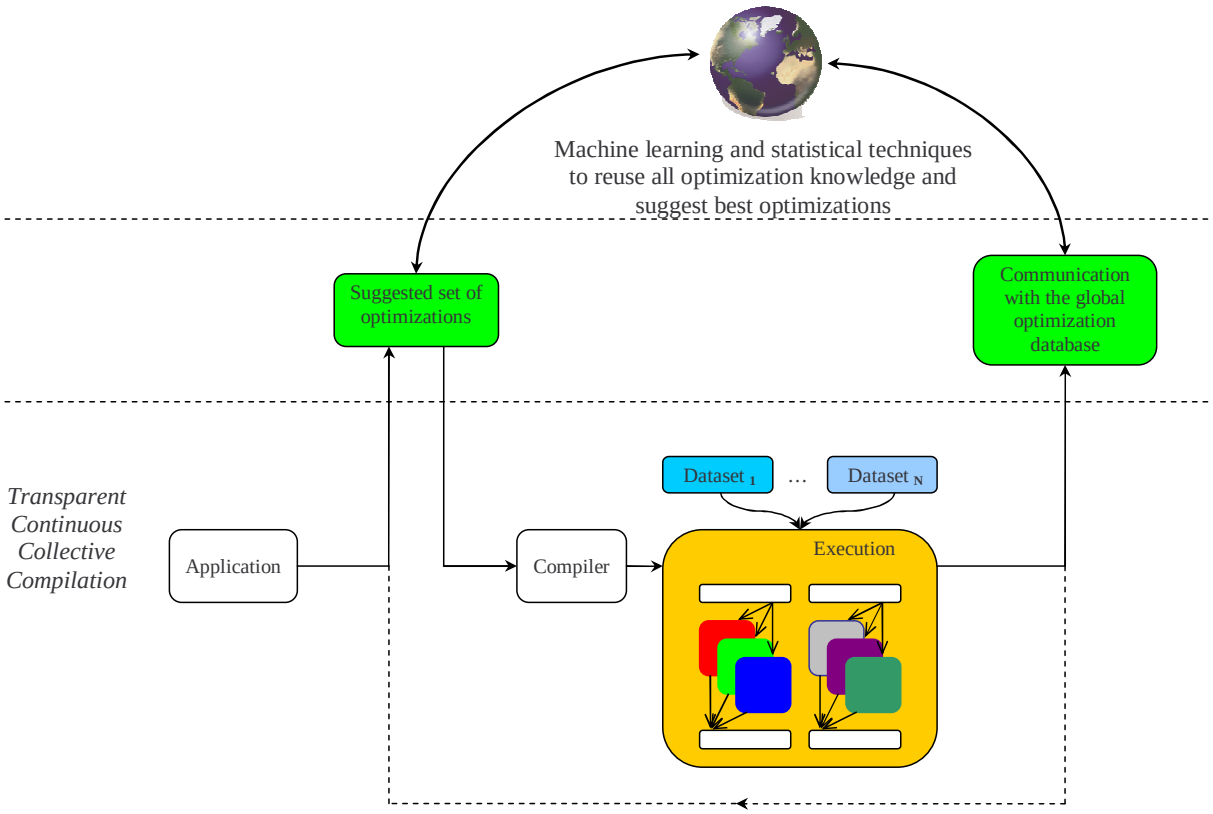

Figure 4: Using our technique to enable Continuous Collective Compilation

Later at run-time we randomly select either original version with the baseline optimizations or the clone during execution at each invocation of this code part and obtain the execution time distribution among original and cloned versions with `gprof` to determine the average speedup when there is a convergence during continuous runs. Though simple, this technique enables continuous collective compilation for statically compiled programs when multiple users transparently share information about behavior of their programs to collectively improve program performance or default compiler optimization heuristic.

We plan to improve our technique to provide a low-overhead execution time coverage instead of `gprof` and improve adaptation technique by monitoring speedup convergence at run-time per selected time slots and using hardware counters to analyze system or program behavior. We use this technique in our Continuous Collective Compilation framework that continuously reuses the collected optimization knowledge from multiple users to speed up the search for the best optimizations for different constraints (performance, power, code size) and continuously improve compiler optimization heuristic using machine learning and statistical techniques [2, 8]. The same technique will also be used to enable program self-adaptation at fine-grain level for multi-ISA platforms such as parallel heterogeneous and reconfigurable systems (CPU/GPU architectures, CELL-like architectures, specialized accelerators, etc), and for varying run-time program or system behavior.

6 Acknowledgments

We would like to thank our colleagues from the University of Edinburgh and IBM for the fruitful discussions on this topic. This work is partially supported by the HiPEAC network of excellence [5] and the MilePost project [4].

References

[1] ACOVEA: Using Natural Selection to Investigate Software Complexities. `http://www.coyotegulch.com/products/acovea`.

[2] Continuous Collective Compilation framework. `http://gcc-ccc.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] European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). http://www.hipeac.net.

[6] MiDataSets SourceForge development site. http://midatasets.sourceforge.net.

[7] QLogic PathScale EKOPath Compilers. http://www.pathscale.com.

[8] GCC Interactive Compilation Interface. http://sourceforge.net/projects/gcc-ici, 2006.

[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] 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.

[12] 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.

[13] H. Chen, J. Lu, W.-C. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2004)*, pages 241–255, 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] 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.

[18] G. Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.

[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 2007)*, January 2007.

[20] G. Fursin and A. Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007.

[21] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.

[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] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[24] M. Haneda, P. Knijnenburg, and H. Wijshoff. Generating new general compiler optimization settings. In *Proceedings of the 19th annual international conference on Supercomputing (ICS'05)*, pages 161–168, New York, NY, USA, 2005.

[25] M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2006.

[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, D. Whalley, G. Tyson, and J. Davidson. Evaluating heuristic optimization phase order search algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169, March 2007.

[28] 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.

[29] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[30] 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'06)*, 2006.

[31] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, volume 6, 2004.

[32] 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.

[33] 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.

[34] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the International Conference on Supercomputing*, 2004.

[35] 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.

[36] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, September 2006. IEEE Computer Society.

[37] R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff. Statistical selection of compiler options. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 494–501, 2004.

[38] B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.

[39] The Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[40] 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.

[41] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 77–90, June 2003.

[42] M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, MIT, USA, 2006.

[43] 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.

[44] M. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the Symposium on Principles and practices of parallel programming*, 2001.

[45] M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Proc. ICPP*, 2000.

[46] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.

[47] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.