Optimizing Chip Multiprocessor Work Distribution Using Dynamic Compilation

Jisheng Zhao, Matthew Horsnell, Ian Rogers, Andrew Dinn, Chris Kirkham, and Ian Watson

University of Manchester, UK
{jishengz,horsnell,irogers,adinn,chris,watson}@cs.man.ac.uk

Abstract. How can sequential applications benefit from the ubiquitous next generation of chip multiprocessors (CMP)? Part of the answer may be a dynamic execution environment that automatically parallelizes programs and adaptively tunes the work distribution. Experiments using the Jamaica CMP show how a runtime environment is capable of parallelizing standard benchmarks and achieving performance improvements over traditional work distributions.

Keywords: Automatic parallelization, feedback-directed optimization, dynamic execution.

1 Introduction

In most traditional optimizing compilers, each optimization has a corresponding performance prediction. These predictions are often based on abstract metrics, with the assumption that there is a direct correlation between the metric and the runtime performance. However, a given program may have markedly different characteristics when run with different input data or on different architectures, significantly impacting the performance of an optimization.

A runtime compilation environment has the potential to take into consideration the most promising optimizations and pick a good choice based on runtime profiling data to maximize performance, and avoid reapplying optimizations shown to incur performance degradation.A compiler-enabled virtual machine framework is presented capable of collecting runtime performance information and automatically reconfiguring the executing code. Using this *Online Tuning Framework* (OTF), a loop-based program can be parallelized and tuned at runtime, with acceptable overheads, increasing the performance when compared to traditional parallelization schemes.

This paper is organized as follows. Section 2 introduces the online tuning framework and its interaction with the Jikes Research Virtual Machine. Section 3 describes the experimental methodology. Section 4 presents and discusses the results from experimental evaluation of the OTF, Section 5 describes how this work compares to related research, and finally Section 6 concludes this paper.

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 258–267, 2007.

2 Online Tuning Framework

The *Online Tuning Framework* (OTF) consists of three distinct elements: the loop parallelizing compiler (see Section 2.1), the adaptive optimization component (see Section 2.2), and the runtime profiler (see Section 2.3).

The OTF is embedded within the adaptive optimization system (AOS) of the Jikes Research Virtual Machine (RVM) [1,2]. The Jikes RVM captures runtime information by instrumenting the running code at the method-level. Once the instrumentation indicates that a given method is *hot* (i.e. number of times the method is executed is above a threshold), the AOS decides whether to compile it using an optimizing compiler[3]. The OTF hijacks this decision, so that any hot method is also considered for parallelizing optimizations. The following sections describe in detail the internal elements of the OTF and how they interact with the AOS.

2.1 Loop Parallelizing Compiler

The Loop Parallelizing Compiler (LPC) searches for fine-grain parallel code within amenable loop structures. The LPC works within two phases of the Jikes RVM optimizing compiler's workflow:

Loop Analysis and Annotation. Loop analysis and annotation occurs in the high-level optimization phase. In this phase the LPC detects loop structures, analyses the data dependencies within them, creates parallel loops where these dependencies can be maintained, and annotates the loops with high-level pseudo code. By performing the analysis of loops at this high-level compiler phase the LPC benefits from Java's strong typing and single static assignment (SSA) form [4].

In order to determine whether array accesses within the loops are amenable to parallelization the Banerjee Test [5] is performed. This allows *do-all* and *do-across* loops to be created when presented with loop carried dependencies on arrays with affine indices. The code for the parallel loop body is placed at the end of the method containing the parallel loop. All parallel loop bodies have a prologue to set up their state and load loop-constant values, and an epilogue to join them back with the parent thread.

Parallel Thread Creation. Parallel thread creation occurs in the machine-level optimization phase. In this phase the previously inserted pseudo-code is replaced by machine specific code, enabling the code to fork new threads on idle processor contexts[1], as well as applying different adaptively optimizing distribution policies.

[1] A processor context is defined as a hardware supported context within a chip multiprocessor architecture.

2.2 Adaptive Optimization Component

The Adaptive Optimization Component (AOC) inserts one or more optimizations deemed to be appropriate for optimizing a given loop, identified by the LPC, into the code. The AOC is invoked by the LPC to place the optimizations around the identified parallel loops. Presently the AOC supports three adaptive optimizations for parallelizable code (see Optimizations 1–3 below). These three optimizations vary either the number of loop iterations inside a block[2] or tile, the number of threads created, or the manner in which the blocks or tiles are distributed. By varying these factors the OTF is able to find strategies that best balance the costs associated with threading, the cache performance, and the system load.

Optimization 1 – Adaptive Block Division (ABD). For a given loop the total number of iterations is divided into blocks. Each block is then distributed through the creation of a parallel thread. The parallel threads can be run on any available processor context. In all optimizations if a thread cannot be invoked on a remote processor context the generator thread must consume its work before continuing to distribute subsequent threads. This optimization uses two simple hill-climbing like algorithms [6] to adaptively divide the total number of iterations in a loop into blocks. The first algorithm, searches for an optimal divisor in the range $1 \leq \text{optimal} \leq \text{number of processor contexts}$. An extension to this algorithm is used to increase the search to the range $1 \leq \text{optimal} \leq m$, where m is a multiple of the number of processor contexts.

Optimization 2 – Adaptive Tile Division (ATD). As loops can be tiled to take advantage of data resuse [7], selecting a suitable tile size is a common technique for improving performance. This optimization is applied when a perfect nested loop is identified by the LPC. The 2-dimensional loop traversal of the iteration space is divided into tiles which are then distributed by the creation of parallel threads. Each tile has a corresponding divisor pair. Given a divisor pair (D_i, D_j), D_i is the divisor corresponding to the outer loop iterator and D_j corresponds to the inner loop iterator. Adaptive searching, again using simple hill-climbing, starts from the divisor pair $(M, 1)$, where M is the total number of processor contexts, this is equivalent to naïve ABD. D_i is incrementally decreased and D_j is increased. Two algorithms are used for ATD, one adaptively optimizes regular 2-dimensional loops and another optimizes triangular 2-dimensional loops. The division of both dimensions of the iteration space is configurable, but currently, the total number of tiles created is restricted to the total number of processor contexts. For regular 2D loops: $D_i \times D_j = M$, and for triangular 2D loops: $D_i \times \frac{D_j+1}{2} = M$.

Optimization 3 – Adaptive Version Selection (AVS). This optimization performs runtime selection between block/tile based loop distribution, as described in optimizations 1 and 2, and distribution using a cyclic recursive distribution (CRD), shown by Figure 1(b). CRD divides the tiles/blocks, using

[2] The term *block* is used to mean a contiguous sequence of loop iterations.

divisor pairs generated using ABD/ATD, such that half of them remain with the generator thread and half are distributed by the creation of a parallel generator thread to another processor context. This happens recursively until all the blocks/tiles are distributed. A variable cyclic stride pattern is also applied to shuffle the division of the blocks.

2.3 Runtime Profiler

To evaluate the performance of the selected adaptive optimizations, the OTF needs to be able to calculate each optimizations runtime profile. This is achieved by inserting two additional code stubs at the start and end of the parallelized loop being profiled. The first stub extracts from the architecture the cycle count[3] prior to the loops execution and the second stub extracts the cycle count after the loop has executed. The second stub is also responsible for reporting the total execution time, and the number of loop iterations, back to the AOS, running in a parallel thread, as shown in Figure 1(a).

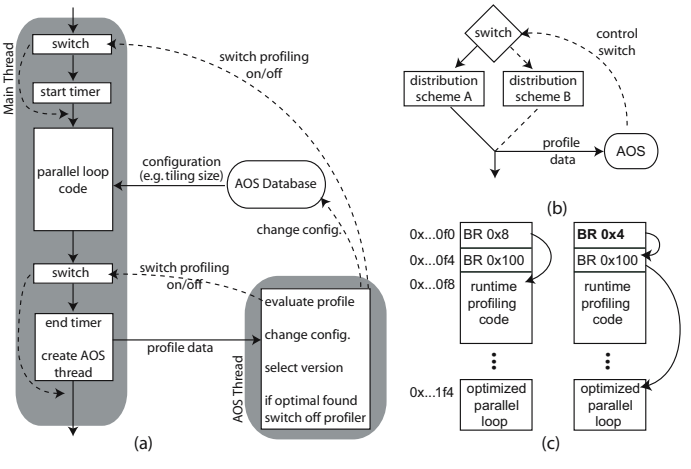


Fig. 1. Runtime profiling and adaptive version switching

The OTF is then able to calculate the execution time per iteration of each invocation of the loop and can make a decision about the comparative performance with other invocations of the loop under different optimizations and divisors. Once an optimal divisor is found for a given optimization the AOS stops profiling it and either switches to a different distribution scheme using AVS, as shown in Figure 1(b), or having assessed all optimizations the AOS switches off the runtime profiler and runs any subsequent executions of the loop using the

[3] Although this mechanism is machine specific; instructions exist in the main architectures: RDTSC (x86), mftb (PPC), TICK register (SPARC).

best optimization found. The code stub that previously invoked the runtime profiling is modified, so that future execution of the code no longer needs to execute any code inside the profiling phase, Figure 1(c). It should also be noted that the AOS instrumentation code for loop back-edges is removed from the parallel code sections which prevents their interruption at runtime.

The precision of the execution time metric is a major factor in getting good results from the presented optimizations, and there are two issues that affect the precision. The first is that not all loops are of static length or duration, it is possible that both the number of iterations and the loops content will vary per invocation. The second issue is that the execution timings are affected by system noise, for example cold caches and other unrelated thread activity. To overcome these issues, the execution time for a given optimization on a parallelized loop is calculated, as an arithmetic mean of the cycles per iteration for four invocations of that loop. Loops that exhibit large profile deviations, defined as having a *coefficient of variation*[4] (CV) greater than a configurable threshold, for this work set at 0.1, are deemed unstable, the profiling code is switched off and the current best optimization is used.

3 Experimental Methodology

The experiments are performed on the JAMAICA architecture [8] using the OTF as part of the adaptive optimization system of the Jikes RVM. The Jikes RVM has been ported to the JAMAICA architecture and runs without an underlying operating system. The JAMAICA architecture is implemented within a highly configurable cycle-accurate processor and memory simulation platform. The main decision behind using a simulated architecture was the ability to evaluate the online tuning framework on a wide range of simulated hardware configurations all using the same instruction set. To evaluate the OTF seven benchmarks have been selected from standard suites, FourierTest from jBYTEmark [9], Euler from JavaGrande [10], MatrixMul, LU, Zchol from JaMa [11], Java Linpack [12] and a Java version of Swim adapted from SpecCPU2000 [13]. Each benchmark is executed to completion and validation on each simulated architecture configuration. The configurations assess the performance of the OTF in the presence of varying cache sizes, the number of cores and the number of contexts per core.

4 Results and Discussion

Figure 2 shows the OTF searching for an optimal divisor in the inner loop for the matrix multiply benchmark using ABD. By the third invocation of the parallelized loop the initial overhead of the runtime profiling code is amortized by the optimized performance, and by iteration 6 a local optimal divisor for this loop has been found. It should be noted that by the very nature of the

[4] Coefficient of variation (CV) is the ratio of the standard deviation to the arithmetic mean.

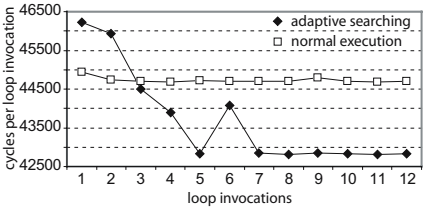
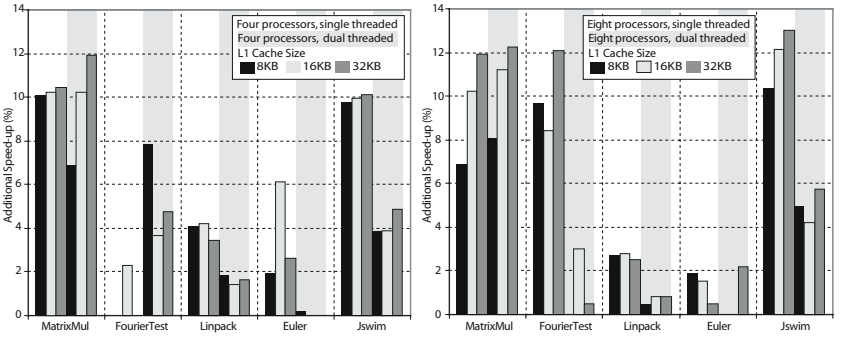


Fig. 2. Searching profile using ABD for the matrix multiply benchmark

hill-climbing algorithms used, the adaptive searching finishes after finding local-optimal solutions.

Figure 3 presents the results of optimizing the benchmarks using the ABD optimization. The results show the speedup achieved using the adaptively found local-optimal divisor, listed in the table, compared to dividing the loop iterations equally to a fixed number of threads, in this case equal to the total number of processor contexts.

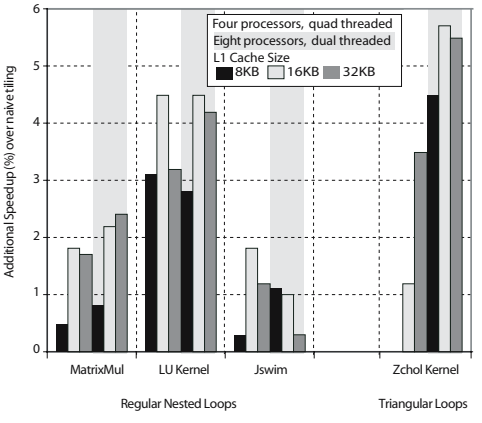


	4 processors, single-threaded			8 processors, single-threaded			4 processors, dual-threaded			8 processors, dual-threaded		
Naïve Divisor	4			8			8			16		
L1 Cache Size	8KB	16KB	32KB	8KB	16KB	32KB	8KB	16KB	32KB	8KB	16KB	32KB
MatrixMul	3.75	3.75	3.75	7.875	7.75	7.75	15.75	7.75	7.75	15.75	15.75	15.75
FourierTest	4	3.75	4	7.75	7.75	7.625	7.875	7.875	7.875	16	15.5	15.75
Linpack	3.75	3.75	3.75	7.625	7.5625	7.625	7.5625	7.625	7.625	15.5	15.5	15.5
Euler	3.875	3.875	3.875	7.625	7.75	7.625	7.75	8	8	16	16	14.25
Jswim	7.875	7.875	3.75	15.75	7.75	7.75	15.875	15.75	7.75	15.5	15.5	15.5

Fig. 3. Speedup of ABD compared to naïve division

For the majority of cases shown in figure 3, the optimal divisor is a value less than the naïve divisor. This is due to the nature of the distribution scheme. The processor context responsible for distributing the parallel threads, the generator, is always the last available for processing any of the loop iterations. In the case of a smaller divisor, 3.75 as opposed to 4, a loop with 100 iterations will be distributed such that the first three distributed threads contain a block of 26 iterations, and the fourth contains only 22. This scheme is therfore able to trade-off the overhead on the generator thread. In the cases where the optimal divisor is

larger than the naïve divisor, the optimization overcomes the context contention overhead. As mentioned previously, in Section 2.2, if the generator thread is not able to distribute a parallel thread to a remote context the work must be done by the generator which prevents subsequent threads being distributed. As the size of the block decreases with larger divisors, each parallel thread contains less work, which reduces the amount of serialization caused by context contention. For this reason in some cases a divisor greater than the number of processor contexts performs better.



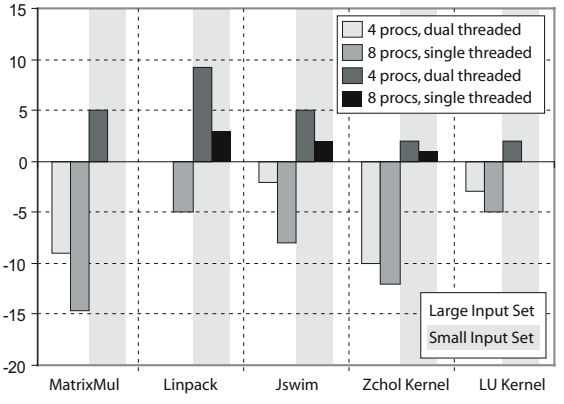
	Loop	Naïve Divisor	4 processors, quad threaded			8 processors, dual threaded		
L1 Cache Size			8KB	16KB	32KB	8KB	16KB	32KB
MatrixMul	Regular	(4.0, 4.0)	(4.0, 3.875)	(4.0, 3.75)	(8.0, 1.875)	(4.0, 3.75)	(8.0, 1.875)	(8.0, 1.875)
LU Kernel	Regular	(4.0, 4.0)	(4.0, 6.5)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)
Jswim	Regular	(4.0, 4.0)	(4.0, 3.75)	(8.0, 1.875)	(8.0, 1.875)	(8.0, 1.875)	(8.0, 1.875)	(8.0, 1.875)
Zchol Kernel	Tringular	(4.0, 7.0)	(4.0, 6.0)	(8.0, 3.0)	(8.0, 2.875)	(4.0, 6.0)	(8.0, 3.0)	(8.0, 2.875)

Fig. 4. Speed-up ATD compared to naïve tile division

Figure 4 presents the results of both regular, for matrix multiply, LU kernel and jswim benchmarks, and triangular, for the Zchol[5] kernel benchmark, ATD optimizations. The table shows the pairs of optimal divisors that achieved the speedups shown, compared to the performance using naïve tile divisor pairs. It should be noted that naïve tile divisor pairs performed better than ABD on the benchmarks shown. Both the regular and the triangular tile division algorithms, optimize the performance of nested loops to achieve more efficient cache use. The table illustrates the variation amongst the optimal divisor pairs given variations in the architecture and cache size. The OTF is able to increase the performance for almost all configurations presented. The ATD optimizations search for an optimal tile size which can take advantage of data resuse and therefore improve cache behaviour. Each architecture used to assess the ATD optimizations contained 16 processor contexts, and the results for the optimal

[5] Zchol implements the Cholesky decomposition of a positive definite matrix.

divisors are normalized against the results gathered using the divisor pair (4.0, 4.0) for regular 2D loops, and (4.0, 7.0) for triangular 2D loops. Both of these divisor pairs generate equal sized square tiles. In the Java programming language, operations on multi-dimensional arrays can be optimized. For example, a loop iteration that loads a value from a 2D array $A[i][j]$, where j is the inner loop iterator, needs two memory load instructions: load $A[i]$ to ref_i and load $ref_i[j]$ into the target register operand. The load operation for $A[i]$ can be moved outside of the loop, reducing the total load operations. For this reason the divisor pair $(8.0, 2.0)$ achieves performance benefits over the simple $(4.0, 4.0)$ divisor pair.



Benchmark	Small Input Set	Large Input Set
MatrixMul	20 x 20	64 x 64
Linpack	90 x 90	150 x 150
Jswim	32 x 32	64 x 64
Zchol Kernel	64 x 64	128 x 128
LU Kernel	64 x 64	128 x 128

Fig. 5. Speed-up from CRD on top of initial ABD/ATD gains

By using the AVS optimization, the OTF can select between a traditional and a cyclic recursive distribution (CRD) policy. To show the affect of AVS the performance of traditional distribution, using optimal divisors located using the ABD/ATD optimizations, and CRD, using the same divisors, was compared. Figure 5, shows the additional performance speedups gained by using CRD with the ABD/ATD optimizations. For each benchmark both a large and small data set was evaluated. Clearly for larger data sets the CRD scheme degrades the performance of the best ABD/ATD using traditional distribution, however, for smaller datasets additional performance increases are achieved for most of the benchmarks. CRD gains performance for smaller data sets as it uses a tree-like distribution policy to create parallel threads. This reduces the overhead of thread creation in the initial generator thread. It also employs a cyclic block distribution, which has been shown to improve the cache performance on multi-threading processors [14], especially for loops working on contiguous memory segments. Furthermore, when CRD is used on triangular loops, this cyclic distribution leads to less variation in the total amount of work received by each processor context. The drawback of CRD, however, is that it increases the number of cache misses when used in a multi-processor environment. This is because a contiguous memory segment will be mapped to different processors' caches.

5 Related Work

Voss and Eigenmann [15] established an adaptive optimization framework named ADAPT which performs dynamic optimization on hot spots through empirical search. The ADAPT uses dynamic recompilation to evaluate different optimizations and a domain-specific language to drive the search on the optimization space for a specific optimization (e.g. for loop unrolling, each level of unrolling will be compiled, run and timed, and the fastest version will be kept and used for the hot spot). The compiler used for recompilation was run on a parallel processor which reduced the recompilation overhead at runtime. *Fursin et al.* [16] explored online empirical searches for scientific benchmarks. To reduce runtime code generation overheads, a set of optimized versions of code were created prior to the execution of a program. These versions were then evaluated at runtime with the best performing version chosen for subsequent execution. They employed predictive phase detection to identify the periods of stable repetitive behavior of a program and used these phases to improve the evaluation of alternative optimized versions. Similarly *Lau, Arnold et al.* [17] investigate an online framework for evaluating the effectiveness of optimizations. They present a virtual machine based online system that automatically identifies the optimal parameters for optimizations, and give an example for selecting method inlinining policy by utilizing the framework. By deploying optimizations at the method-level, more runtime noise is present in the system, and they use a large number of iterations to assess the effectiveness of optimizations. *Diniz and Rinard* [18] use a simple version selection mechanism which reacts to runtime inputs and loop parameters. Their dynamic optimization system also generates code to provide dynamic feedback, allowing automated selection of the best synchronization policy for parallel execution.

In contrast with the above work, the method presented in this paper combines a loop-level parallel compiler and an adaptive optimization framework within a Java virtual machine that works on a CMP architecture. By employing the Jamaica CMP's capability of distributing fine-grain parallelization, the adaptive optimization system can perform online adaptive tuning to improve the performance of parallelized code for smaller data sets with acceptably low overheads.

6 Conclusion and Futher Work

This paper has presented an Online Tuning Framework, capable of locating and parallelizing loops. The framework is able through runtime profiling, to search for an optimal division of a parallelizable loop into blocks and an optimal distribution of the loop blocks across the parallel resources of a chip multiprocessor. This system realises additional performance speedups, up to 12% on the benchmarks assessed, over a traditional parallelization system, including the initial overheads involved with the profiling system.

The OTF is currently directed to optimize work distribution within parallel code sections which exhibit stability during profiling. As part of further studies the framework will be extended to *time-out*, whereupon previously unstable

code sections, and those previously optimized but performing poorly, will be re-evaluated. Additionally the framework is being extended to automatically optimize work distribution across larger and more scalable CMP architectures.

References

1. IBM: Jikes^{TM} Research Virtual Machine (2005), `http://jikesrvm.sourceforge.net/`
2. Arnold, M., Fink, S.: Adaptive optimization in the Jalapeño JVM. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 47–65 (2000)
3. Burke, M., et al.: The Jalapeeño dynamic optimizing compiler for Java. In: Proceedings ACM 1999 Java Grande Conference, San Francisco, CA, United States, pp. 129–141. ACM Press, New York (1999)
4. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Symposium on Principles of Programming Languages, pp. 107–120 (1998)
5. Banerjee, U.: Loop Transformations for Restructuring Compilers: The Foundations. Springer, Heidelberg (1993)
6. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1996)
7. Wolfe, M., Shanklin, C., Ortega, L.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1995)
8. Wright, G.: A single-chip multiprocessor architecture with hardware thread support. PhD thesis, The University of Manchester (2001)
9. Grehan, R., Rowell, D.: jBYTEMark Benchmark (1998)
10. Bull, J., et al.: A benchmark suite for high performance Java. Concurrency Practice and Experience 12(6), 375–388 (2000)
11. Hicklin, J., Moler, C.: **Ja**va **Ma**trix Package Benchmarks (July 2005)
12. Dongarra, J., Wade, R.: Linpack Benchmark - Java Version
13. Henning, J.: SPEC CPU2000: measuring CPU performance in the New Millennium. Computer 33(7), 28–35 (2000)
14. Lo, J., Eggers, S.: Tuning compiler optimizations for simultaneous multithreading. In: International Symposium on Microarchitecture, pp. 114–124 (1997)
15. Voss, M., Eigenmann, R.: High-level adaptive program optimization with ADAPT. In: PPOPP, pp. 93–102 (2001)
16. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: A practical method for quickly evaluating program optimizations. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 29–46. Springer, Heidelberg (2005)
17. Lau, J., Arnold, M.: Online performance auditing: using hot optimizations without getting burned. In: PLDI, pp. 239–251 (2006)
18. Diniz, P., Rinard, M.: Dynamic feedback: an effective technique for adaptive computing. In: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp. 71–84 (1997)