Combining Model and Iterative Compilation for Program Performance Optimization

Pingjing Lu
National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha 410073, China
Email: pingjinglu@gmail.com

Yonggang Che and Zhenghua Wang
National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha 410073, China
Email: {ygchecn, zhhwang}@nudt.edu.cn

***Abstract*—The performance gap for high performance applications has been widening over time. High level program transformations are critical to improve applications' performance, many of which concern the determination of optimal values for transformation parameters, such as loop unrolling and blocking. Static approaches achieve these values based on analytical models that are hard to achieve because of increasing architecture complexity and code structures. Recent iterative compilation approaches achieve it by executing different versions of the program on actual platforms and select the one that renders best performance, outperforming static compilation approaches significantly. But the expensive compilation cost has limited their application scope to embedded applications and a small group of math kernels. This paper proposes a combinative approach--Combining Model and Iterative Compilation for Program Performance Optimization (CMIC). Such an approach first constructs a program optimization transformation model based on hardware performance counters to decide how and when to apply transformations, and then selects the optimal transformation parameters using Nelder-Mead simplex algorithm. Experimental results show that our approach can effectively improve programs' floating-point performance, reducing programs' runtime, therefore, lessening the performance gap for high-performance applications.**

***Index Terms*—Performance, Languages, Measurement, Experimentation, Algorithms**

I. INTRODUCTION

With the rapid development of processors following Moore's Law, the performance gap between memory and processors is widening, which has influenced single processors' performance greatly. High-level transformations such as loop unrolling, array tiling, and array padding are effective ways to improve programs' performance. Many of these transformations have numerical parameters whose values must be carefully selected to achieve optimal performance. There are mainly two kinds of optimization methods to compute optimal optimization parameters—model-driven optimization and empirical optimization. A model-driven method uses a simple model of the program and the machine to select parameters, which might be not precise due to the increasing complexity of architectures and programs. Automated empirical optimization method generates multiple versions of the program, runs them on the actual machine, and selects the one that renders the best performance [1]. With this empirical optimization approach, ATLAS [1, 2], PHiPAC[3], and FFTW[4] successively generate highly optimized libraries for dense, sparse linear algebra kernels and FFT respectively. It has been shown that empirical method is more effective than model-driven method [5]. However, because the optimization spaces (set of all possible program transformations) are large, non-linear with many local minima, finding a good solution may be long and non-trivial, making iterative method quite time consuming. To best optimize programs' memory performance, it would be a better choice to combine model-driven and empirical optimization methods, which utilizes apriori information to narrow the optimization space and uses some search method to empirically search good optimization parameters.

This paper proposes the approach of Combining Model and Iterative Compilation for Program Performance Optimization (CMIC), which first uses the performance counter values collected from a few runs of the program to determine the programs optimization model which decides when and how to transform a loop to most effectively optimize programs' performance, and then selects the optimal transformation parameters that renders the least runtime using Nelder-Mead simplex algorithm.

This paper is organized as follows. The next section describes the framework of CMIC. Section 3 presents the Program Optimization Transformation Model based on Hardware Performance Counters (POTraM), including the performance metrics it used and how they can characterize program behavior. Section 4 provides Iterative Optimization of Transformation Parameters based on Nelder-Mead Simplex Algorithm (ItOTraP)

used in CMIC. Section 5 describes the experimental setup and the experimental results and analysis. This is followed by concluding remarks and future work.

II. Framework of CMIC

CMIC includes two parts. 1) Generation of parameterized optimization code based on analytical models. In this part programmers transform and parameterize the program based on POTraM which will be described in Section 3. 2) Iterative optimization of transformation parameters based on Nelder-Mead Simplex algorithm. Iterative optimization selects the optimal parameters for every optimized program. The search engine selects performance data using performance monitoring tools. In our approach we evaluate the parameters using actual program execution time, and generate new optimization parameters using Nelder-Mead Simplex algorithm. Therefore, the program using the selected optimal parameter will render least runtime. In the next two sections, we will discuss the two parts in detail. The framework of CMIC is illustrated in Fig. 1.

III. Program Optimization Transformation Model based on Hardware Performance Counters

This section first looks at the performance counters used in this paper, then illustrates how they can be used to characterize well-known properties of programs, and POTraM is finally presented.

A. Performance Counters

Modern processors are often equipped with a special set of registers that allow for measuring performance counter events with no disruption to the running program. The information obtained from performance counters is a compact summary of a program's dynamic behavior. In particular, they summarize important aspects of a program's performance, e.g., cache misses or floating point unit utilization. Performance counters have been extensively used for performance analysis in explaining program behavior [6, 7]. One of the first papers to investigate how they could be used systematically to select optimizations [8] showed impressive performance gains.

We use hardware performance monitoring tool PAPI (Performance Application Programming Interface) [9] to access these hardware performance counters, which is developed at the University of Tennessee's Innovative Computing Laboratory in the Computer Science Department. The performance counters used in this study are shown in Table I. The first column lists the performance counter acronyms, and the second column gives a description.

TABLE I.
Performance counters used

Name	Meaning
PAPI_L1_DCA	L1 data cache accesses
PAPI_L1_DCM	L1 data cache misses
PAPI_L1_TCM	L1 total cache misses
PAPI_L2_TCA	L2 total cache accesses
PAPI_L2_TCM	L2 total cache misses
PAPI_FP_INS	Floating-point instructions

B. Dynamic characterization of program behavior using performance counters

This section lays the foundation for the application of transformations that improve programs' performance. In this paper we optimize programs on Intel Pentium D platform; therefore, we only consider L1 cache and L2 cache.

B.1. Cache miss rate

L1 data cache miss rate:

$$MSR_{L1D} = PAPI\_L1\_DCM \big/ PAPI\_L1\_DCA \quad (1)$$

L2 cache miss rate:

$$MSR_{L2} = PAPI\_L2\_TCM \big/ PAPI\_L2\_TCA \quad (2)$$

It is generally thought that below 5% is a fairly good cache miss rate, and if certain level's cache miss rate of the program is smaller than 5%, it means that the program has good cache access locality in this cache level.

B.2. Performance influence ratio

In modern processors, the memory and instruction pipeline utilization of programs influences performance most. Mo, et al [10, 11] present the concepts of Influence Ratio for Memory Reference (η_m) to quantify the impact of memory and pipeline operations. η_m represents the impact degree of cache miss to performance, and the higher the cache hit rate, the less η_m will be. η_{pl} represents the impact degree of instruction level parallelization to performance, and it reflects whether applications have fully utilized multi-function units and super-scalar instruction pipeline architectures of microprocessors. The higher reuse ratio of the operators in the register, the less η_{pl} will be. η_{fp} describes applications' utilization ratio of single processor peak floating point performance, which relies on η_m and η_{pl}.

Let C_1 and C_2 be L1 cache and L2 cache miss cost (in cycles), F be processors' frequency (in Hz), G be machines' peak floating-point performance (in Mflops), T be program running time (in seconds), and T_m be the total time cost of cache miss (in seconds), then we can compute η_m, η_{pl} and η_{fp} as follows:

$$T_{m1} = PAPI\_L1\_TCM * C_1 \big/ F \quad (3)$$

$$T_{m2} = PAPI\_L2\_TCM * C_2 \big/ F \quad (4)$$

$$T_m = T_{m1} + T_{m2} \quad (5)$$

$$\eta_m = T_m \big/ T \quad (6)$$

$$\eta_{pl} = 1 - PAPI\_FP\_INS / (G*(T - T_m)) \quad (7)$$

$$\eta_{fp} = PAPI\_FP\_INS / (G*T) \quad (8)$$

C. POTraM

Our approach obtains above-mentioned metrics using information collected by hardware performance counters, and achieves a program optimization transformation model--POTraM based on these metrics computed. The model we have presented can use them to provide source-code level feedback to a programmer about what transformations to concentrate on those most likely to result in memory performance boost. POTraM determines whether and how to apply various loop transformations, and guides the restructure of program loops to achieve high performance on specific target architectures. The flowchart of POTraM is described in Fig. 2.

1. Run the program. Collect and compute memory dependent metrics in Section B;
2. if (MSR_{L1D}>5%)
   {Apply loop exchange transformation.}
3. if (η_m>30%)
   { 3.1 Apply loop tiling transformation;
   3.2 Apply array tiling transformation.}
   else if (η_m>20%)
   {Apply loop tiling transformation.}
4. if (η_{pl}>80%)
   {Unroll the innermost loop.}
5. Terminate, resulting in the optimal transformations.

Figure 2. Model of POTraM

Ⅳ. Iterative Optimization of Transformation Parameters based on Nelder-Mead Simplex Algorithm

A. Theoretical Analysis

Iterative compilation optimization parameter selection problem can be formalized as follows:

$$\begin{aligned} &\min f(x = (p_1, p_2, \cdots, p_n)) \\ &Subject\ to \\ &\begin{cases} low_i \le p_i \le up_i,\ i = 1, 2, \cdots, n \\ p_i \in Z \\ f \in R \end{cases} \end{aligned} \quad (9)$$

Where p_i is one program transformation parameter; x is the compositional parameter vector of all transformation parameters, which is the search target of search algorithm; $f(x)$ is program execution time with parameter x, which is the objective function to minimize. Therefore, the optimization space includes $\prod_{1\le i\le n} (up_i - low_i)$ compositional parameter vectors. It's very huge, and highly non-linear.

Many researches focus on exploring search heuristics in iterative compilation method; however, there has been no suitable search strategy for exploring the large and complex search space. Previous researches show that genetic algorithm (GA) is successful to find the best sequence of compiler passes [12], however, Kulkarni et al. [13] find that simple techniques, such as local hill climbing, when allowed running over multiple iterations, can often outperform complex techniques such as GA. Kisuki et al. [14] also show that in finding transformation parameters, random search performs as well as other sophisticated techniques such as GA and simulated annealing. Recent works by Apan Qasem et al. [15] find that direct search can be an effective technique for finding good values for transformation parameters in a reasonable time. Haihang You et al. [16] apply simplex method to replace the ATLAS (Automatically Tuned Linear Algebra Software) [1] search heuristics, and find that simplex search scheme can produce parameters with better performance.

The Nelder-Mead simplex method [17, 18] is a classical and powerful direct search method for optimization. It appears to be a good fit to the problem of finding optimization parameters to minimize program runtime. First, the space of possible sequences is quite large when several optimizations are applied. This space is too large to search completely. Second, we have a very good evaluation function to assess the quality of a solution. We simply perform the optimizations and test programs runtime. Our objective function is discrete and nonlinear, making the problem difficult to address with more classical combinatorial optimization techniques. Third, The Nelder-Mead simplex method is useful for training parameters, especially for searching minima of multi-dimensional functions when dimension is less than 20. It is mainly used to solve the minimization problem: $\min f(x)$, where $f: R^n \to R$, and the gradient information is not available. Finally, the amount of time spent by Nelder-Mead simplex algorithm is flexible. More computation time may result in better solutions, but the algorithm can be interrupted at any time to return the best solution found. Therefore, we design the Nelder-Mead simplex algorithm based optimization parameter selection algorithm to solve iterative compilation optimization parameters selection problem. Our experiments show that Nelder-Mead simplex algorithm is well suited to the problem of finding good

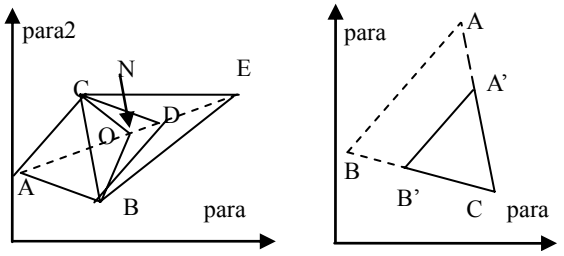

Figure 3. The advance procedure of two parameters Nelder-Mead simplex algorithm. (left) Reflection, contraction and expansion operation (right) Shrink operation.

optimization parameters.

B. Introduction to Nelder-Mead Simplex Algorithm

Spendley, Hext, and Himsworth[17] introduced the simplex method, which is a non-derivative based direct search method. Nelder and Mead improved the method by adding more moves and making the search more robust and faster [18]. A "simplex" is a geometrical figure consisting of n+1 point in n-dimensions, e.g. a 2-dimension simplex is a triangle. Through a sequence of elementary geometric transformations, the initial simplex moves towards minimum, and away from maximum. It is probably the most widely used optimization method. The advance procedure of two parameters Nelder-Mead simplex algorithm is demonstrated in Fig. 3. The original simplex consists of A, B, C, and $A>B>C$. Therefore, A is reflected through O, the centroid of B and C, generating D. If D is better than B and C, then expansion point E is generated. If E is better than D, A is replaced by E; else A is replaced by D. If D is worse than A, contraction point ND is generated. If it is better than A, A will be replaced by it; else the simplex ABC will be shrunk as a smaller simplex $A'B'C$ which is demonstrated in Fig. 3 (right).

Considering that if one parameter vector performs badly, it may be because the transformation parameter is too big (or too small) and makes full use of cache, register, pipeline, and etc. Through the reflection operation in Nelder-Mead simplex method, a much smaller (or much bigger) parameter will be generated, and it is apt to perform well. If reflection results in new minimum, we will consider that if it moves further along the minimization direction would it perform even better? So we do so through expansion operation. If the reflection point is still the worst point, we will try a smaller step with a contraction operation. If a simple contraction doesn't improve things, then try moving all points towards the current minimum through shrink operation. We think the performance of new generated parameter -- through a series of geometric transformation in Nelder-Mead simplex algorithm -- is more likely better than that of randomly generated parameters and that of GA through mutation, and crossover operation. Therefore, we design the Nelder-Mead simplex based optimization parameter selection algorithm to search the optimal optimization parameters.

C. Nelder-Mead Simplex Based Optimization Parameter Selection Algorithm

For the simplicity of algorithm description, we first list some notations in the Nelder-Mead simplex based optimization parameter selection algorithm.

Notation:

$S^{(k)}$: The simplex in the k^{th} iteration, and $S^{(k)} = (x_0^k, x_1^k, \cdots, x_n^k)$;

x_h^k : The highest (worst) point, i.e. $f(x_h^k) = \max\{f(x_i^k) \mid i = 0,1,\cdots,n\}$;

$x_{\inf h}^k$: The second highest point, i.e. $f(x_{\inf h}^k) = \max\{f(x_i^k) \mid i = 0,1,\cdots,n, \text{ and } i \neq h\}$;

x_l^k : The lowest (best) point, i.e. $f(x_h^k) = \min\{f(x_i^k) \mid i = 0,1,\cdots,n\}$;

$\overline{x}^k$: Average of all points, excluding the worst (highest) point;

$\max iter$: The maximum iteration number;

ε : The precision requirement;

$\alpha, \beta, \gamma, \omega$: The coefficient of reflection, contraction, expansion, and shrink.

The Nelder-Mead simplex based optimization parameter selection algorithm is described as follows.

1.*Initialization*. For each parameter, we define an empirical range Ω. Randomly generate a non-degenerate initial simplex $S^{(1)}$ that belongs to R on Z_n, then do a measurement at each point. Set parameters: $\max iter, \varepsilon, \alpha, \beta, \gamma, \omega$, and set k=1.

2. Find $x_h^k, x_{\inf h}^k, x_l^k$, and calculate $\overline{x}^k$.

3. *Reflection*. $x_r^k = (1+\alpha)\overline{x}^k - \alpha x_n^k$, where $\alpha > 0$; if any parameter of x_r^k outreaches the range Ω, then randomly regenerate a new one that belongs to it.

3.1 *if* $f(x_r^k) \le f(x_l^k)$

{ go to 4}

3.2 *else if* $f(x_l^k) \le f(x_r^k) \le f(x_{\inf h}^k)$

{replace x_n^k with x_r^k, and go to 7}

3.3 *else if* $f(x_r^k) \ge f(x_{\inf h}^k)$

{ go to 7}

4. *Expansion*. $x_e^k = \gamma x_r^k + (1-\gamma)\overline{x}^k$, where $\gamma > 1$;

if any parameter of x_e^k outreaches the range Ω, then randomly regenerate a new one that belongs to it.

4.1 *if* $f(x_e^k) \le f(x_r^k)$

{ replace x_n^k with x_e^k, and go to 7;}

4.2 *else*

{ replace x_n^k with x_r^k, and go to 7.}

5. *Contraction*.

5.1 *if* $f(x_r^k) < f(x_n^k)$,

{ $x_c^k = \beta x_r^k + (1-\beta)\overline{x}^k$, where $0 < \beta < 1$;

if any parameter of x_c^k outreaches the range Ω, then randomly regenerate a new one that belongs to it. }

5.1.1 *if* $f(x_c^k) < f(x_r^k)$

{ replace x_n^k with x_c^k, and go to 7; }

5.1.2 *else*

{go to 6. }

5.2 *else*

{ $x_c^k = \beta x_n^k + (1-\beta)\bar{x}^k$, where $0 < \beta < 1$;

if any parameter of x_c^k outreaches the range Ω, then randomly regenerate a new one that belongs to it.}

5.2.1 *if* $f(x_c^k) < f(x_n^k)$

{replace x_n^k with x_c^k, and go to 7; }

5.2.2 *else*

{ go to 6.}

6. *Shrink*. All the points in the simplex except the lowest point are shrunk**,** i.e. $x_i^k = x_0^k + \omega(x_i^k - x_0^k)$, where $0 < \omega < 1,\ i = 0,1,\cdots n,\ i \neq l$.

7. *Stop criterion check.*

7.1 *if k*> max *iter* or $\{\frac{1}{n+1}\sum_{i=0}^{n}[f(x_i^k) - f(\bar{x}^k)]^2\}^{\frac{1}{2}} \le \varepsilon$

{ stop. x_l is the final solution, and $f(x_l)$ is the optimal object function. }

7.2 *else*

{ *k*=*k*+1, go to 2. }

Note that the most time-consuming part of the simplex method is the measuring of the execution time. Also notice that at least one set of parameters (the one with the minimum execution time) remains unchanged from one generation to the next. Therefore, there is no need to recalculate the execution time for that set of parameters. In the experiments, we found that more than half of parameters already been executed previously. To improve the running performance of the algorithm, we keep record of the parameters and execution time of previously executed points in a list. When measuring execution time, we check the list to see if the execution time for that set of parameters is already available. If so, there is no need to recalculate it and we move on to the next set of parameters.

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance of CMIC by comparing the program behavior of cache miss rate, program balance Performance influence ratio performance before and after optimization. Using performance model POTraM, the optimization space has been narrowed down which can be seen from Table III. Experiments demonstrate the feasibility of our approach by showing that CMIC method can produce parameter values with excellent performance in reduced optimization space.

TABLE II.
SALIENT ARCHITECTURAL PARAMETERS OF THE EXPERIMENTAL PLATFORM

Frequency	2.8 GHz
L1Data/ Instruction	2×16/2×12 (KB)
L2 cache	2×1024KB
Memory	DDR2 1G
OS	Ubuntu kernel 2.6.15-23-386
Compiler	Intel Fortran Compiler 9.0 -O3
Peak performance	2.8Gflops
BM1 /BM2 /BM3	16 / 32 /2.3
C1 / C2	4 / 31 (cycles)
Bus speed of L1 cache /L2 cache/Memory	44.8 / 89.6 / 6.4 (GB/s)

A. Experimental Setup

(1) Platforms Our experiments are performed on platform Intel Pentium D 820. Table II lists its salient architectural parameters. Peak performance, BM_i (i=1,2,3), C_i (i=1,2) and bus speed of L1 cache / L2 cache / Memory are calculated using the method presented by Jack Dongarra in [21].

(2) Benchmarks we test two different scale of three typical numerical compute kernels, the matrix multiplication program (mm) with scale 512 and 1024, Successive Overrelaxation (sor) with scale 512 and 1024, and Red-Black Successive Overrelaxation (rbSor) with scale 192 and 512. In the next figures, we name the program with small scale as the name of program followed by 1, and that with large scale are named as the name of program followed by 2. For example, program mm with scale 512 and 1024 are named as mm1 and mm2 respectively.

(3) Transformations We have tested program's behavior listed in Section III.B, and the experimental results and responding transformations based on POTraM are listed in Table III. T, P and U stand for loop tiling, array padding and innermost loop unrolling respectively.

(4) Parameter Settings Let max *iter* be 150, ε be 0.0001. And set $\alpha = 1.0,\ \beta = 0.5, \gamma = 2.0,\ \omega = 0.5$ [19]**.**

B. Experimental Results

We compare the performance of original programs and

TABLE III.
DYNAMIC CHARACTERIZATION PERFORMANCE DATA OF ORIGINAL PROGRAMS

	MSR_{L1D} (%)	MSR_{L2} (%)	BL_1	BL_2	BL_3	η_m (%)	η_{pl} (%)	η_{fp} (%)	transfo -mation
mm1	10.2	0.2	205	88	3.2	29.5	90.9	6.4	(T,U)
mm2	31.5	4.7	109	94	3.8	34.7	89.6	6.8	(T,P,U)
rbSor1	30.8	8.4	69	65	4.8	42.9	85.7	8.2	(T,P,U)
rbSor2	42.7	29.5	75	87	5.8	45.6	90.3	5.3	(T,P,U)
sor1	21.8	4.6	49	37	1.4	47.5	75.9	12.7	(T,P)
sor2	22.9	5.6	38	31	2.3	49.2	65.7	17.4	(T,P)

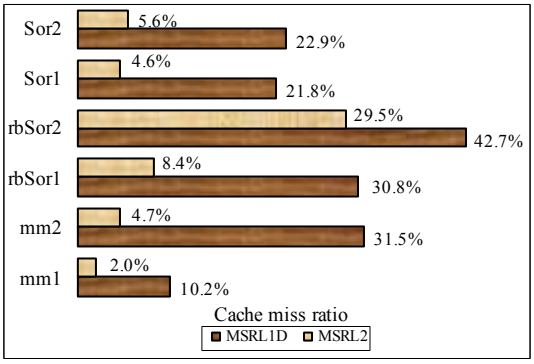

Figure 4. Cache miss rate before optimization

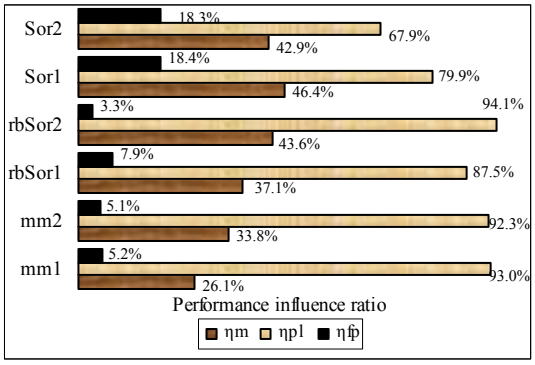

Figure 6. Performance influence ratio before optimization

optimized programs after transformations implemented in POTraM as shown in Table III.

Fig. 4 and Fig. 5 show the dynamic characteristics performance of cache. We can see that the original L1 data cache miss rate of all the programs are greater than 5%, therefore, it is necessary to optimize programs' memory performance to fully utilize cache. After loop exchange, cache miss rate of both L1 data and L2 cache decrease, especially after loop transformation implemented in POTraM.

Fig. 6 and Fig. 7 show the performance influence ratio for memory reference, pipeline and floating-point operation of the programs. The original η_m of all test programs except mm_1 are greater than 30%, demonstrating that the memory utilization of programs is very low, which will influence programs performance, therefore, loop tiling and array tiling transformation are carried out. The original η_m of mm_1 is greater than 20%, but less than 30%, thus we just carried out loop tiling transformation. After optimization, η_m reduces to less than 10%, therefore, programs' memory performance has increased greatly, thus narrowing the gap between memory and processors. η_{pl} of mm and rbSor are greater than 80%, demonstrating that the pipeline and registers utilization of programs is very low, therefore, loop unrolling transformation is carried out on them. Meanwhile, η_{pl} of sor are lower than 80% (75.9 and 65.7), therefore, loop unrolling transformation is not carried out on them. The improvement of memory performance and pipeline will result in the boost of program floating-point performance as seen in Figure 9. The original η_{fp} of all test programs except sor are less than 10%, and that of sor are less than 20%, after optimization, η_{fp} of programs improved up to 58.5%.

Fig. 8 illustrates the benefits of CMIC—the reduction of program execution time. Because the execution time of different programs vary greatly and it's hard to integrate all these programs' information in one graph, we normalize them as the speedup relative to original program, where speedup equals the running time of original program divided by that of optimized program.

The results show that CMIC can effectively improve programs' floating-point performance, reducing programs' runtime, therefore, lessening the performance gap for high-performance applications. Meanwhile the optimization space has been reduced using performance model POTraM. Experiments validate that CMIC approach presented in this paper can be a practical and portable means to implement architecture-aware optimizations for high-performance applications.

VI. Conclusion and Future Work

Present compilers fail to model the complex interplay between different optimizations and their effect on code on all the different processor architecture components. Meanwhile the high cost of iterative compilation approaches has limited their application scope to

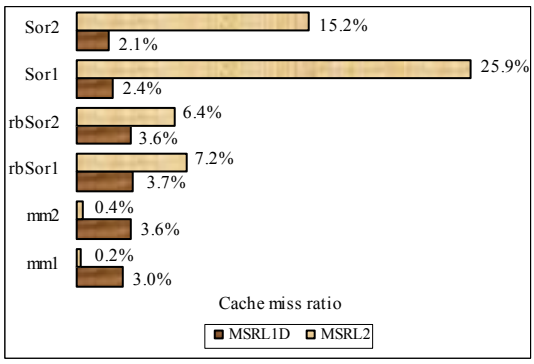

Figure 5. Cache miss rate after optimization

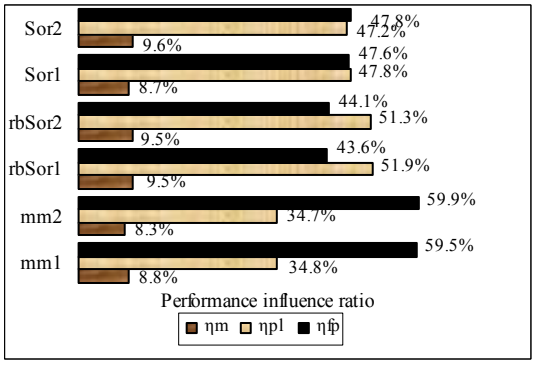

Figure 7. Performance influence ratio after optimization

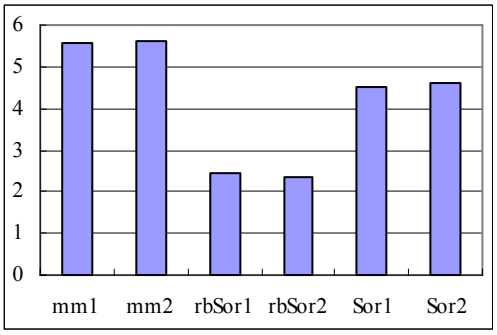

Figure 8. Speedup of tested programs

embedded applications and a small group of math kernels. CMIC method presented in this paper combines program optimization transformation model based on hardware performance counters and iterative optimization to improve the performance of programs. The transformation model can effectively narrow down the transformation parameter space, and the ability of the Nelder-Mead simplex algorithm to discover good solutions in relatively less iterations makes it a good choice for an iterative compilation search strategy. Experimental results show that our approach can greatly reduce cache miss rate, program balance of all levels, and the influence ratio for memory reference, effectively improve programs' floating-point performance, reducing programs' runtime, therefore, lessening the performance gap for high-performance applications, which makes it a practical and portable means to implement architecture-aware optimizations for high-performance applications.

In future, we plan to improve our strategy by adding more architectural and program information to the model, so as to better guide the application of program transformations, and use training data sets during the tuning process to cut down the program execution time.

ACKNOWLEDGMENT

This work was partially supported by the National Natural Science Foundation of China under Grant No.60603055 and the National High Technology Development 863 Program of China under Grant No. 2007AA01Z116.

REFERENCES

[1] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project", *Parallel Computing*, Vol. 27, No, 1--2, pp. 3--35, Jan. 2001.

[2] Jim Demmel, Jack Dongarra, et al, "Self adapting linear algebra algorithms and software", *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Adaptation*", Vol. 93, No. 2, 2005.

[3] Jeff Bilmes, Krste Asanovic, et al, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology", *In International Conference on Supercomputing*, pp. 340–347, 1997.

[4] Matteo Frigo and Steven G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT", *In Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, Vol. 3, pp. 1381–1384, 1998.

[5] Kamen Yotov, Xiaoming Li, Gang Ren, et al, "A Comparison of Empirical and Model-driven Optimization", In PLDI '03: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 63–76, 2003.

[6] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos, "Profileme: Hardware support for instruction-level profiling on out-of-order processors", *In International Symposium on Microarchitecture*, pp. 292–302, 1997.

[7] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters", *In ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pp. 101–110, 2005.

[8] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun, "Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors", *In SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pp. 15, 2004.

[9] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors", *International Journal of High Performance Computing Applications*, Vol. 14, No. 3, pp. 189–204, Aug 2000.

[10] Mo Zeyao, Liu Xingping, Liao Zhenmin, "Research on key techniques for parallelization and optimization of applied codes", *Proceedings of 6th International Parallel Computing conference*, National University of Defense Technology Press, China, 2000.

[11] Mo Zeyao, "Realistic performance analysis methods for parallel codes", *Journal of Numerical Computing and Computer Applications*, Vol. 21, No. 4, pp. 266-275, 2000.

[12] P. Knijnenburg, T. Kisuki, and M. O. Boyle, "Iterative compilation", *In Embedded Processor Design Challenges System Architecture, Modeling and Simulation (SAMOS)*, Lecture Notes in Computer Science 2268, pages 171–187. Springer Verlag, 2002.

[13] Peter M. W. Knijnenburg, Toru Kisuki, Kyle Gallivan, Michael F. P. O'Boyle, "The effect of cache models on iterative compilation for combined tiling and unrolling", *Concurrency and Computation: Practice and Experience 16*(2-3): 247-270, 2004.

[14] Kamen Yotov, Keshav Pingali, Paul Stodghill, "Think Globally, Search Locally", *In ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, Boston, MA, USA, 2005.

[15] Apan Qasem, Ken Kennedy, John Mellor-Crummey, "Automatic Tuning of Whole Applications Using Direct Search and a Performance-based Transformation System", *Proceedings of the LACSI Symposium*, pp: 183-194, 2004.

[16] H. You, K. Seymour and J. Dongarra: An Effective Empirical Search Method for Automatic Software Tuning. *UTK CS Technical Report*, ICL-UT-05-02, 2005.

[17] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization", *The Computer Journal*, No.8, pp. 308-313, 1965.

[18] W. Spendley, G.R. Hext, and F.R. Himsworth, "Sequential Application of Simplex Designs in Optimization and Evolutionary Operation". *Technometrics*, No. 4, pp. 441-461, 1962.

[19] Jack Dongarra, "Performance Optimization for Cluster Computing", *in Proceedings of the Myrinet Users Group Conference*, Vienna, Austria, 2002.

Pingjing Lu was born in Anhui Province of China in 1984. She received the B.A's and M.A' s. degree in computer architecture from the National University of Defense Technology, Hunan, China, in 2004 and 2006 respectively. Since 2006, she has been a Ph.D. student in computer science from the National University of Defense Technology. Her current research interests include computer systems performance evaluation and compiler optimization.

Yong-Gang Che, born in Yunnan Province of China in 1973. He received the B.A's, M.A' s and Ph.D.'s. degree in computer architecture from the National University of Defense Technology, Hunan, China, in 1997, 2000 and 2004 respectively. He has been an associate professor of computer science at the National University of Defense Technology since 2006. His current research interests include computer architecture and compiler optimization.

Zheng-Hua Wang, born in Hunan Province of China in 1962. He received the B.A's, M.A' s and Ph.D.'s. degree in aerodynamics from the National University of Defense Technology, Hunan, China, in 1983, 1986 and 1991 respectively. He has been a professor of computer science at the National University of Defense Technology since 1999. His research interests are in computer systems performance evaluation and parallel processing.