Computing and Informatics, Vol. 29, 2010, 1159–1179

UMDA/S: AN EFFECTIVE ITERATIVE COMPILATION ALGORITHM FOR PARAMETER SEARCH

Pingjing Lu, Yonggang Che, Zhenghua Wang

National Laboratory for Parallel and Distributed Processing
School of Computer
National University of Defense Technology
Changsha 410073, China
e-mail: pingjinglu@gmail.com

Manuscript received 29 September 2009; revised 26 January 2010
Communicated by Vladimír Kvasnička

Abstract. The search process is critical for iterative compilation because the large size of the search space and the cost of evaluating the candidate implementations make it infeasible to find the true optimal value of the optimization parameter by brute force. Considering it as a nonlinear global optimization problem, this paper introduces a new hybrid algorithm – UMDA/S: Univariate Marginal Distribution Algorithm with Nelder-Mead Simplex Search, which utilizes the optimization space structure and parameter dependency to find the near optimal parameter. Elitist preservation, weighted estimation and mutation are proposed to improve the performance of UMDA/S. Experimental results show the ability of UMDA/S to locate more excellent parameters, as compared to existing static methods and search algorithms.

Keywords: Iterative compilation, optimization parameter, Nelder-Mead simplex algorithm, estimation of distribution algorithms, univariate marginal distribution algorithm

Mathematics Subject Classification 2000: 68N20, 68T05, 62H10

1 INTRODUCTION

Over the last several decades we have witnessed tremendous change in the field of computer architecture. New architectures have emerged at a rapid pace with computing capabilities that often exceed our expectations. However, the rapid rate of architectural innovations is becoming a major concern for the high performance computing community. Each new architecture or even a new model of a given architecture has brought with it new features that add to the complexity of the target platform. As a result, it has become increasingly difficult to exploit the full potential of modern architectures for complex scientific applications [1]. The gap between the theoretical peak and the actual achievable performance is widening over time, due to the tremendous complexity increase in microprocessor and memory architectures, and to the rising level of abstraction of popular programming languages and styles [2]. High level program transformations are critical in optimizing the performance of compiled code. Many of these transformations need numerical parameters, which should be carefully selected. Good parameters could bring great performance boost, while unsuitable parameters would bring less performance improvement, or even degrade performance. Most compilers estimate these parameters using static architectural models that are hard to achieve because of the increasing architecture complexity. Recent researches show that iterative compilation approach is a practical means to implement architecture-aware optimizations for high-performance applications, outperforming static compilation approaches significantly [3, 4]. It generates different program versions, and selects the one with the best performance by actually running them on target hardware and using certain search strategies [5].

Iterative compilation is promising in determining the optimal parameter values. It is essentially target neutral, and could adapt to different program behavior and different platforms. Therefore, it has been a hot research topic in the high performance computing community. However, because the optimization space (set of all possible program transformations) is large and non-linear with many local minima, finding a good solution using iterative method may be too time-consuming. We also notice that program optimization is a task where near optimal solutions would be satisfactory. Therefore, this paper presents a novel search algorithm UMDA/S–Univariate marginal distribution algorithm with Nelder-Mead simplex search, which combines Univariate Marginal Distribution Algorithm (UMDA) and Nelder-Mead simplex method (which will be called simplex method for short in this paper) to find the near optimal parameter values.

The rest of this paper is organized as follows. Related work is elaborated in Section 2. Formal description of the parameter search problem of iterative compilation is given in Section 3. UMDA/S algorithm is presented in Section 4. Experimental results are given in Section 5. Finally, we present a short conclusion along with a discussion about future work in Section 6.

2 RELATED WORK

The research efforts in iterative compilation search algorithms can be broadly classified into two categories according to the search space they operate on. Several ongoing research projects try to find the best compilation sequence using iterative compilation, known as phase order problem; others concentrate on finding the best parameter values for transformations that use numerical parameters. Although both of these approaches deal with very large search space, characteristics of these two search spaces can be quite different. Thus strategies used in exploring these two types of search space are also somewhat different. This paper focuses on the latter, i.e. tuning of numerical parameters.

Many studies on the search strategies for iterative compilation optimization parameters are unidimensional in nature. That is, they search for the best parameters of one transformation at a time. Fursin et al. [6] utilize random search algorithm to explore the search space of tiling, unrolling and array padding factors. Their strategy, however, does not use any intelligent search methods in the search space exploration and is unidimensional. Experiments with three SPEC benchmarks show that their strategy significantly outperforms native compilers on a variety of platforms. Chen et al. [7] combine analytical models with empirical search to automatically tune dense matrix computations to two different architectures. When performing search in one dimension, reference values are used for other dimensions. Although this strategy works reasonably well for ATLAS and some of the other empirical tuning systems, it has one major limitation. That is, the search strategy does not account for interaction between transformations and their search strategy – except for the tiling search space – is unidimensional. It is well established that many transformations interact with each other in complex ways and this complex interaction can have significant impact on program performance, especially when loop transformations are targeting the memory hierarchy. Thus when searching for the best parameter values for multiple transformations, it is imperative that the search is multi-dimensional in nature.

However, search the optimization parameters multi-dimensionally is NP-hard in nature [8]; therefore, many researchers prefer to solve this kind of problems based on genetic algorithm (GA), which is a non-deterministic method, providing the chance to efficiently address the NP-hard problems. Experimental results have shown that it is effective in finding good optimization sequences [5]. However, its applicability in finding suitable numerical parameters is somewhat limited. For numerical parameters, using a GA is not too different from using a random search. Kisuki et al. [4] use a variety of search techniques including genetic algorithm, simulated annealing, pyramid search and random search to explore the combined search space of tile sizes and unroll factors. A somewhat interesting and surprising finding of this work is that none of the search strategies used in their system has a clear advantage over the others. In fact, in most cases, random search performs just as well as some of the other more sophisticated search techniques. The main problem with the search for iterative compilation numerical parameters is that we still know relatively little about the nature of the search space, and thus there are still no models that can accurately describe the search space. Since many search strategies require some modelling of the search space to work efficiently, as yet we have not found a suitable search strategy for exploring the search space of transformation parameters.

One recent principled alternative to traditional evolutionary algorithms is the so-called estimation of distribution algorithms (EDAs), which use a probabilistic model of promising solutions to guide the exploration of the search space [9, 10]. EDAs evolve a population of potential solutions to the given optimization problem using a combination of evolutionary computation and machine learning. There is no traditional crossover or mutation in EDAs. Instead, they explicitly extract global statistical information from the parent population set and build a posterior probability distribution model of promising solutions, based on the extracted information. New solutions are sampled from the model thus built, and fully or in part replace the parent population to form the new population. EDAs outperform other types of evolutionary algorithms on broad classes of challenging problems. Meanwhile, recent works by Apan Qasem et al. [11, 12] find that simplex search scheme can be an effective technique for finding good values for transformation parameters in reasonable time. Therefore, this paper explores the hybrid of EDAs and simplex method to find near optimal parameters.

3 PROBLEM FORMULATION

In this paper we formalize iterative compilation parameter selection problem as a global optimization problem, then we apply UMDA/S to solve this problem.

3.1 Formulation of Iterative Compilation Parameter Selection Problem

Let $x_1, x_2, \dots, x_n$ be the performance-critical program transformation parameters, where n is the number of parameters. To the best of our knowledge, the parameters in program optimization are integers, and based on the domain-specific knowledge, each parameter has an upper bound up_i and a lower bound low_i, i.e. $x_i \in \mathrm{Z}$, $low_i \leq x_i \leq up_i, i = 1, 2, \dots, n$. The compositional parameter vector of all transformation parameters $(x_1, x_2, \dots, x_n)(x_i \in \mathrm{Z}, low_i \leq x_i \leq up_i, 1 \leq i \leq n)$ is called a parameter vector, represented as $x = (x_1, x_2, \dots, x_n)$. Each parameter vector is called a *feasible solution* or a *solution*. The space composed by all solutions, i.e. all the integer points embraced by the lower and upper bound of all parameters, is called *solution space* or *feasible region*, denoted as D. $\|D\| = \prod\limits_{1 \leq i \leq n} (up_i - low_i + 1)$. Denote the program execution time using parameter vector x as $T(x)$, the fitness function $f(x) = 1/T(x)$; then iterative compilation parameter search problem can be formalized as the following global optimization problem:

$$
\begin{aligned}
&\min T(x=(x_1,x_2,\ldots,x_n)) \quad \mathit{or} \quad \max f(x=(x_1,x_2,\ldots,x_n))\\
&\mathit{Subject} \quad \mathit{to}\\
&\begin{cases} low_i \le x_i \le up_i, i=1,2,\ldots,n\\ x_i \in Z.\end{cases}
\end{aligned}
\tag{1}
$$

In this paper, we do not distinguish between the term parameter vector and solution. The quality of each solution is determined by its fitness function value. The higher the fitness function value, the better the quality of the parameter vector.

3.2 Optimization Parameter Selection Process

The optimization parameter selection process selects the optimal x so that $T(x)$ is minimized, and the procedure is as follows:

1) Transform the original program P. For transformations that use parameters, define the parameters as constants (to be determined in 2) or 4)) in the head file of the program. This step generates program P'.
2) The search algorithm generates initial values for the constants in the head file.
3) Program P' is compiled with the native compiler.
4) The search engine executes the executable of program P', and measures its execution time. The search engine keeps track of different parameters evaluated, decides which parameters to apply next and updates the values in the head file. Goto 3) unless search stop condition is satisfied.
5) The parameter with the shortest execution time so far is selected as the near optimal parameter.

This paper utilizes UMDA/S to search the parameter values for three parametrized transformations: array padding, loop tiling, and loop unrolling.

3.3 Parameterization

Now consider the parameterization, through which the numerical parameters can be abstracted from the program. Parameterization makes use of the native compiler to simplify the implementation of iterative compiler and to eliminate repeated preprocessing procedure, if possible. It uses the head file to interface with the native compiler. The optimization parameters appear as compile-time constants in the head file of source program P'. These parameters are searchable by UMDA/S. For program P', the head file may be defined in the form

$$\boxed{\text{Parameter}(x_1\text{=[value 1]}, \ldots, x_n\text{=[value n]})}$$

Many high level language compilers provide loop unrolling directives. So for loop unrolling transformation, we can place an unroll directive before the loop in the source code and let the native compiler do the actual loop unrolling job. For example, if the native compiler is Compaq Visual Fortran, and the unroll factor is x_k, then the loop can be unrolled in this form

```
CDEC$ UNROLL(x_k)
   {the target loop }
```

For array padding transformation, the padding parameters are directly used in defining the array. If x_k is the intra-array padding factor, then the array can be padded in the form:

```
A(N,N) ⟹ A(N+x_k,N)
```

For optimizations like loop tiling, the tile sizes are directly used in the optimized program, and the source code is the same for different tile sizes. In this case, by regarding the parameters as runtime parameters, we can further remove repeated native compilation procedures from iterative compilation. This can be done by defining the parameters as variables, and insert file operations to read them from parameter file before they are used. For example, if x_k and x_j are two optimization parameters, PFile is the parameter file name, then in program P':

```
    open(10, file='PFile', status='old')
    read(10, 100) x_k, x_j
100 FORMAT(3I)
    close(10)
    {codes that use x_k and x_j}
```

Nearly all program transformation parameters can be made as runtime parameters by multi-version techniques theoretically. However, this may cause code expansion and other side effects. So we only apply this to some math kernels. Then UMDA/S is employed to find the near optimal parameter value, which will be introduced next.

4 UMDA/S: AN EFFECTIVE PARAMETER SEARCH ALGORITHM FOR ITERATIVE COMPILATION

Before discussing the algorithm and its implementation, let us look into the rationale of UMDA/S for the search of iterative compilation numerical parameters.

First, the main problem with the search for iterative compilation numerical parameters is that we still know little about the nature of the search space. Many transformations interact with each other in complex ways and this complex interaction can have significant impact on program performance; therefore, we do not have models that can describe the search space with a high degree of accuracy. EDAs use a probabilistic model of promising solutions to guide the exploration of the search space. They explicitly extract global statistical information from the parent population set and build a posterior probability distribution model of promising solutions, based on the extracted information. EDAs outperform other types of evolutionary algorithms on broad classes of challenging problems. Therefore, we employ EDAs to exploit the distribution of good points in the search space, guiding the search of excellent parameters.

Second, why should we use UMDA when much more advanced models are available, such as multiply connected Bayesian networks? The most important reason to prefer simple models is efficiency. Simple models can usually be created much faster than the complex ones and they still provide sufficient accuracy. It has been known that, in order to efficiently solve a given problem using EDAs, there is no necessity to represent and use all the dependencies [13, 14, 15, 16]. Problems with certain interactions between the variables may be candidates to be solved with UMDA [17].

Finally, the addition of other optimization methods to EDAs that use univariate probabilistic models permits the solution of problems with complex interactions without the computational burden of using more complex probabilistic models [18, 19]. Simplex method [20], a classical and powerful direct search method for optimization, fits in the role perfectly. First, the optimization space is too large to search completely. Second, the 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 cost of simplex algorithm can be flexibly controlled. The longer the search runs, the better the solution may be; but the algorithm can be interrupted at any time, returning the best solution found up to that time.

From the analysis above, we can see that UMDA/S should be a good choice for iterative compilation parameter selection strategy by combining UMDA and simplex method. It uses the probabilistic model of promising solutions in UMDA to guide the exploration of the search space, and utilizes the simplex method to optimize parameters in the population, which helps to locate the better-performing solutions more concisely and then guide the evolutionary direction better.

4.1 Preliminaries

1) **Encoding.** Most existing EDAs deal with binary-encoded optimization problem, but the binary encoding has the following shortcomings:

   - Two neighboring integers may have a long Hamming distance when using binary encoding, which will slow down the convergence speed.
   - When solving high-dimensional optimization problems, the string of binary encoding is very long, which will slow down the search speed of the algorithm.

   Integer encoding can overcome these problems; what is more, it facilitates the inclusion of domain-specific heuristic information, enhancing the search ability of algorithm. Therefore, this paper adopts integer encoding.

2) Initialization. The distribution of excellent solutions is unknown initially. To keep the diversity of solutions, and to avoid being trapped in the local optima, when generating the initial population, optimization parameter vectors should be dispersed in the feasible region as evenly as possible, so that the initial population can include as much information about feasible region as possible. Therefore, in UMDA/S, the initial population is generated following uniform distribution.

3) Selection policy. [10] argued that proportional selection, truncation selection and tournament selection can theoretically ensure the global convergence of EDAs. This paper considers proportional selection, and denotes the proportional selection factor as *rBest*.

4) Build the probability model of the selected solutions

- Learn the probability distribution of each optimization parameter based on counting. The probability of optimization x_i valued j is

$$p(x_i = j) = \frac{\#\left(x_i = j | D_l^S\right)}{N} \qquad (j \in [low_i, up_i], j \in Z) \tag{2}$$

where D_l^S is the selected population, N is the size of D_l^S, and $\#(x_i = j|D_l^S)$ is the number of x_i valued j in D_l^S. It can be proved to be a maximum likelihood estimation [21].

- For each $x^j \in D_l^S$, build the probability model $p_l(x^j)$:

$$p_l(x^j) = p_l\left(x^j | D_l^S\right) = \prod_{i=1}^{n} p_l(x_i^j) = \prod_{i=1}^{n} \frac{\#\left(x_i = x_i^j | D_l^S\right)}{N}. \tag{3}$$

4.2 UMDA/S Algorithm

UMDA/S works as follows:

1. Randomly generate M solutions from D following uniform distribution, then apply the simplex method to each solution, generating M new solutions, forming the initial population.
2. Run the test program, evaluate each solution, and select N ($N < M$) best-performing ones to form the excellent population. Build the probability model of the selected solutions.
3. Sample M solutions from the probability model, then apply the simplex method to each solution to generate M new solutions, which forms the new population.
4. If the stopping conditions are not satisfied, go to 2).

To clearly illustrate UMDA/S, Algorithm 1 and Algorithm 2 are decomposed from UMDA/S. Algorithm 1 is the main part of UMDA/S, which uses the probabilistic model of promising solutions in UMDA to guide the exploration of the search space; Algorithm 2 adopts simplex method to optimization parameter vectors generated by UMDA during initialization and sampling, to locate the better-performing solutions more concisely.

```
Input: test program testFile, and the number of optimization parameters $n$;
Output: optimal optimization parameter vector $x = (x_1, x_2, \cdots, x_n)$ and the corresponding program
        running time $T(x)$
(1) Setting algorithm parameters: population size $M$, the size of selected population $N$, and the
    maximum generation $N_{pop.}$
(2) Initialization.
  a) Generate $M$ optimization parameter vectors $\tilde{x}^1, \tilde{x}^2, \cdots, \tilde{x}^M$ from $D$;
  b) For each $\tilde{x}^i \in \{\tilde{x}^1, \tilde{x}^2, \cdots, \tilde{x}^M\}$ { $x^i \leftarrow$ NDSimplex (testFile, $\tilde{x}^i$, $n$) ; }
  c) $x^1, x^2, \ldots, x^M$ form the new population $D_l$, $l = 0$.
(3) Fitness value evaluation. For each $x^i \in \{x^1, x^2, \ldots, x^M\}$ { Run testFile with optimization
    parameter vector $x^i$, achieve $T(x^i)$, and compute $f(x^i)$ .}
(4) Selection. Select $N$ optimization parameter vectors from $D_l$ to form selected population $D_l^s$ .
(5) Build the probability model $p_l(x)$ .
    For each $x^j \in D_l^s$ , build the probability model $p_l(x^j)$ based on Formula (3).
(6) Generate the new population by sampling from the probability model.
  a) Sampling $M$ times from $p_l(x)$, achieve $M$ optimization parameter vectors $\tilde{x}^1, \tilde{x}^2, \cdots, \tilde{x}^M$ ;
  b) For each $\tilde{x}^i \in \{\tilde{x}^1, \tilde{x}^2, \cdots, \tilde{x}^M\}$ { $x^i \leftarrow$ NDSimplex (testFile, $\tilde{x}^i$, $n$) ; }
  c) $x^1, x^2, \ldots, x^M$ form the new population $D_{l+1}$, $l = l + 1$.
(7) Stop condition check. If $l < N_{pop}$, go to (3).
```

Fig. 1. Algorithm 1: UMDA/S

For the simplicity of algorithm description, we first list some notations in the simplex–based parameter selection algorithm; and during the process of NDSimplex(), if any parameter of the new generated point through reflection, contraction, expansion, and shrink operators do not belong to D, regenerate a new one.

Notation.

- $S^{(k)}$: The simplex in the kth iteration, i.e.

$$S^{(k)} = \left(x_0^k, x_1^k, \ldots, x_n^k\right);$$

- x_h^k: The highest (worst) point, i.e.

$$f\left(x_h^k\right) = \max\left\{f\left(x_i^k\right) \mid i = 0, 1, \ldots, n\right\};$$

- $x_{\inf h}^k$: The second highest point, i.e.

$$f\left(x_{\inf h}^k\right) = \max\left\{f\left(x_i^k\right) \mid i = 0, 1, \ldots, n, i \neq h\right\};$$

- x_l^k: The lowest (best) point, i.e.

$$f\left(x_l^k\right) = \min\left\{f\left(x_i^k\right) \mid i = 0, 1, \ldots, n\right\};$$

- $\bar{x}^k$: Average of all points, excluding the worst (highest) point;
- *MaxIter*, ε: The maximum iteration number and precision requirement;
- $\alpha, \beta, \gamma, \omega$: The coefficient of reflection, contraction, expansion, and shrink.

From the initial point generated by UMDA, denoted by $\tilde{x}$, the simplex method will generate the initial simplex in the following way:

$$\begin{cases} x_1^0 = \tilde{x} \\ x_i^0 = \tilde{x} + (-1)^i * (i/2) * ones(1, n) \quad i = 1, 2, \ldots, n \end{cases} \tag{4}$$

where $ones(1, n)$ returns an 1-by-n vector of 1s, so that several points will be generated around $\tilde{x}$ with step 1 or -1, and $\tilde{x}$ will be locally optimized.

4.3 Improvements Policies for UMDA/S

To avoid local convergence and improve the efficiency of optimization, the following improvement policies are added to the basic UMDA/S:

1) **Elitist preservation.** In standard UMDA, there is no guarantee that once a good solution vector is found, it will remain in the population of the subsequent generation. Therefore, elitist preservation is used to resolve this. When generating the next generation population from the current population, select a small proportion of elitists, which have the highest fitness, and copy them to the next generation, so that the evolution will not degrade. We denote $\eta_{Elitist}$ proportion of elitists, i.e. proportion of elitists to the population size.

2) **Weighted estimation.** Formula (3) is substituted with the following formulas:

$$\omega_i = 1/T(x^j) \tag{5}$$

$$p_l(x^j) = \prod_{i=1}^{n} \left(\frac{\omega_i \cdot \#\left(x_i = x_i^j | D_l^S\right)}{N \cdot \sum_{i=1}^{n} \omega_i} \right) \tag{6}$$

   Formula (5) computes the weight of x^j. As shown in Formula (6), when building the probability model, the shorter the execution time of the optimization parameter vector, the bigger weight it owns. This implies that the better solutions will play a greater role in the distribution estimation for the next population.

3) **Mutation.** In order to perform extensive search, genetic diversity must be maintained. Otherwise, it is possible for UMDA/S to settle into a sub-optimal state.

```
Input: initial parameter vector $\tilde{x}$, test program testFile, and the number of optimization parameter $n$;
Output: optimized parameter vector $x$.
(1) Setting parameter of simplex: MaxIter, $\alpha, \beta, \gamma, \omega$. $k \leftarrow 0$.
(2) Initialization. Based on initial point $\tilde{x}$, generate $n+1$ points from $D$ following Formula (4), to
    form initial simplex $S^{(0)}$.
(3) Fitness evaluation.
  a) For each $x_i^k \in \{x_1^k, x_2^k, \cdots, x_{n+1}^k\}$
       { Run test program testFile with parameter $x_i^k$, achieve $T(x_i^k)$, and compute $f(x_i^k)$.}
  b) Find $x_h^k, x_{\inf h}^k, x_l^k$, and compute $\overline{x}^k$.
(4) Reflection. $x_r^k = (1+\alpha)\overline{x}^k - \alpha x_n^k$.
  a) if $f(x_r^k) \le f(x_l^k)$ { go to (6); }
  b) else if $f(x_l^k) \le f(x_r^k) \le f(x_{\inf h}^k)$ { $x_n^k \leftarrow x_r^k$, go to (8); }
  c) else if $f(x_r^k) > f(x_{\inf h}^k)$ { go to (8). }
(5) Expansion. $x_e^k = (1-\gamma)\overline{x}^k - \gamma x_r^k$.
  a) if $f(x_e^k) \le f(x_r^k)$ { $x_n^k \leftarrow x_e^k$, go to (8); }
  b) else { $x_n^k \leftarrow x_r^k$, go to (8). }
(6) Contraction.
  a) if $f(x_r^k) < f(x_n^k)$ { $x_c^k = \beta x_r^k + (1-\beta)\overline{x}^k$ . }
       i. if $f(x_c^k) < f(x_r^k)$ { $x_n^k \leftarrow x_c^k$, go to (8); }
      ii. else { go to (7). }
  b) else { $x_c^k = \beta x_n^k + (1-\beta)\overline{x}^k$ . }
       i. if $f(x_c^k) < f(x_n^k)$ { $x_n^k \leftarrow x_c^k$, go to (8); }
      ii. else { go to (7). }
(7) Shrink. $x_i^k = x_0^k + \omega(x_i^k - x_0^k)$, $k = 1, 2\ldots, n+1, i \ne l$.
(8) Stop criterion check.
  a) if $k > MaxIter$ or $\{\frac{1}{n+1}\sum_{i=0}^{n}[f(x_i^k) - f(\overline{x}^k)]^2\}^{\frac{1}{2}} \le \varepsilon$ {Stop. Return $x_l^k$ as the final solution. }
  b) else { $k = k+1$, go to (3). }
```

Fig. 2. Algorithm 2: NDSimplex()

Mutations are introduced to the basic UMDA/S algorithm to help preserve diversity and to escape from local optima. We can apply mutation in each generation of UMDA/S; however, this will increase the search cost. If the generation interval for mutations is too long, diversity may be lost. Therefore, we apply mutation every several generations when the maximum fitness of the generation keeps unchanged, which indicates UMDA/S may have settled into a sub-optimal state. In our experiment, if the maximum fitness of the generation keeps unchanged over any three continuous generations, select two parameter vectors from the current population using the roulette wheel method, and mutate them using the full arithmetic mutation:

$$\begin{cases} x_i = Mod(x_i + irand(-muStep_i, muStep_i), up_i) & 1 \leq i \leq n \\ if(x_i < low_i) \quad x_i = irand(low_i, up_i) \end{cases} \tag{7}$$

where $Mod()$ is the modulo function, $irand(low_i, up_i)$ is the function which randomly generates an integer between low_i and up_i, and $muStep_i(i \leq i \leq n)$ is the possible maximum mutation step which is defined by users in advance and is usually set as $1/6 \sim 1/8$ of $(up_i - low_i)$.

4) **OPEOT (One Parameter Evaluated One Time).** Note that the most time-consuming part of our search algorithm 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 find that more than half of the parameters have already been executed previously. To improve the performance of the algorithm, we keep record of the parameters and execution time of previously executed parameters in a list: *ExecList*. When measuring execution time for one set of parameters, we first check *ExecList* to see if its execution time is available. If so, we move on to the next set of parameters.

5 PERFORMANCE EVALUATION

5.1 Environmental Setup

We test three typical numerical compute kernels, the matrix multiplication program (MxM) with matrix sizes 512 and 1 024, Successive Overrelaxation (Sor) with scale 512 and 1 024, and Red-Black Successive Overrelaxation (rbSor) with scale 192 and 256. The outer time step of Sor and rbSor is 100. In this paper we only consider three parameterized transformations: array padding with parameters up to 64, loop tiling with parameters up to 256, and loop unrolling with parameters up to 128; their parameterization has been described in Section 3.3. The search space consists of approximately 1 052 possible different transformations which are unrealistic to explore. Experiments are performed on two platforms – Intel Core2 Quad Q6600 (Core2) and Intel Pentium D 820 (PentiumD). Table 1 lists the architectural parameters of each platform.

Parameter settings for UMDA/S are as follows: population size $N = 100$, selected population size $M = 50$, maximum evolutionary generation $N_{pop} = 30$, elitist proportion $\eta_{Elitist} = 0.05$, proportion selection factor $rBest = 0.5$. Parameter settings for GA are as follows: population size $N = 100$, maximum evolutionary generation $N_{pop} = 30$, mutation probability is 0.05, and crossover probability is 0.7. Settings for simplex are as follows: $MaxIter = 10$, $\varepsilon = 0.0001$. As to the selection of $\alpha, \beta, \gamma, \omega$, Nelder and Mead [20] recommended that $\alpha = 1.0$, $\beta = 0.5$, $\gamma = 2.0$, $\omega = 0.5$.

	Core2	*PentiumD*
Frequency	2.4 GHz	2.8 GHz
L1Data	4×32 KB	2×16 KB
L1Instruction	4×32 KB	2×12 KB
L2 Cache	$2{\times}4\,096$ KB	$2 \times 1\,024$ KB
Memory	DDR2 2 G	DDR2 1 G
OS	Windows XP professional	Windows XP professional
Compiler	Intel Fortran Compiler 9.0 -O3	Intel Fortran Compiler 9.0 -O3

Table 1. Experimental Platforms

5.2 Experimental Results

Before analyzing and comparing execution times or the effect of different search algorithms, we need to determine the precision of the timing on the particular platform. The execution time is generally measured using the system timer and can oscillate from run to run due to varying operating system management processes. We also created an additional tool to measure the precision of the execution time. In this paper, we execute the same application 10 times and compare the average performance of the application.

5.2.1 Performance Comparison

5.2.2 Comparison with Existing Static Techniques

We first compare our UMDA/S–based iterative compilation with state-of-the-art static techniques. Here it is compared to two well-known static optimization techniques proposed by Lam et al. in [22] and by Coleman and McKinley in [23]. Lam, Rothberg, and Wolf presented LRW [22] for computing the tile size. They select the largest square tile that does not incur self-interference conflicts for matrix multiply based on the periodicity in the addressing of a direct-mapped cache and the constant-stride accesses. This algorithm takes the matrix size N and the cache size C as the input and returns the largest tile size without conflict misses. Coleman and McKinley presented TSS [23]. They select rectangular tiles to remove both capacity and conflict misses. TSS also assumes that the cache is direct-mapped. It takes the cache size C, the line size and the array column dimensions (N and M) as the input. It calculates the number of complete columns that fit into cache.

Both techniques attempt to reduce conflict and capacity misses by using loop tiling. Both techniques are applied to statically determine the tile size. We evaluate them and compare them with our UMDA/S based iterative optimization. Since both of them only apply loop tiling, iterative compilation is restricted to loop tiling to have a fair comparison. Tables 2 and 3 demonstrate the speedup over native compilation with optimization option -O3 of static optimization algorithms: LRW and TSS, that of iterative compilation with loop tiling and that of iterative compilation with all transformations enabled with our UMDA/S search algorithm.

	MxM 512	MxM 1 024	Sor 512	Sor 1 024	rbSor 192	rbSor 512
LRW	20.1 %	20.5 %	0 %	0 %	0 %	0 %
TSS	21.3 %	21.9 %	0 %	0 %	0 %	0 %
UMDA/S (only tiling)	26.8 %	28.6 %	21.8 %	22.4 %	11.7 %	16.0 %
UMDA/S (all transformations)	33.2 %	35.1 %	28.3 %	30.2 %	14.9 %	20.3 %

Table 2. Speedup on Core2 of LRW, TSS, and UMDA/S based iterative compilation

	MxM 512	MxM 1 024	Sor 512	Sor 1 024	rbSor 192	rbSor 512
LRW	19.9 %	21.0 %	0 %	0 %	0 %	0 %
TSS	21.6 %	21.9 %	0 %	0 %	0 %	0 %
UMDA/S (only tiling)	27.0 %	28.1 %	18.5 %	25.7 %	11.0 %	15.3 %
UMDA/S (all transformations)	33.8 %	35.9 %	24.2 %	33.9 %	15.0 %	18.6 %

Table 3. Speedup on PentiumD of LRW, TSS, and UMDA/S based iterative compilation

Results show that LRW and TSS perform well on MxM, outperforming native compilers which are optimized for maximum speed performed by the compiler on both platforms. This is because when performing optimizations the native compilers try to avoid degradation of performance. Therefore, they either do not apply loop tiling or apply it with a small factor that generally does not degrade performance but improvement is not so significant. Hence, the above static techniques have a greater potential to pick up tile size better. However, for Sor and rbSor, LRW and TSS fail to achieve any performance improvement, mainly due to assuming the use of the direct-mapped cache that is not the case and by using approximations to count interferences. Nevertheless, iterative compilation actually executes different versions of the program with different tiling factors, with UMDA/S searching over the optimization space; therefore, it obtains more performance improvement.

5.2.3 Comparison with Existing Search Algorithms

In our previous work [24], the comparison of the simplex method with other search algorithms has been done, and the results show that for iterative compilation parameter selection problem, the Nelder-Mead simplex search strategy can produce parameter values with better performance than that of GA and random search. Therefore, in this paper, we only evaluate the effectiveness of UMDA/S by comparisons with UMDA, simplex algorithm and GA, and test the impact of improvement policies to UMDA/S.

Because the execution time of different programs varies greatly and it is hard to integrate all these programs' information in one graph, we normalize the performance as the speedup over native compilation with optimization option -O3 which is optimized for maximum speed performed by the compiler.

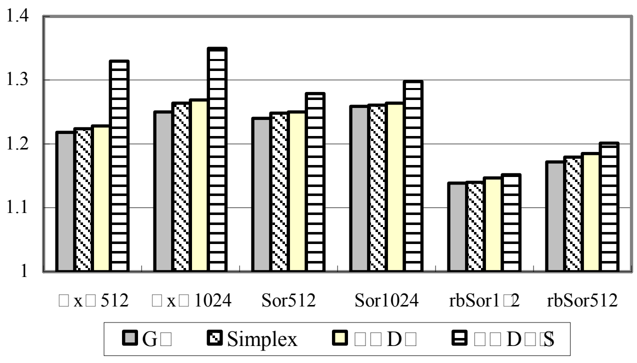

Fig. 3. Speedup on Core2 of different search algorithms

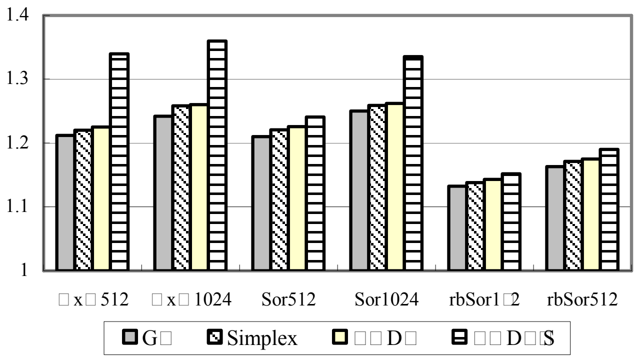

Fig. 4. Speedup on PentiumD of different search algorithms

Figures 3 and 4 show that the optimization parameter obtained by UMDA, simplex and UMDA/S is more excellent than that by GA, but simply application of UMDA or simplex will not bring great improvement. Only the combination of them can boost application performance significantly. This is because by combining UMDA and simplex, UMDA/S can control the structure of optimization space and the evolutionary direction on the whole; meanwhile by improving the quality of optimization parameter vectors in each generation using simplex method, the search algorithm can locate the local optima better.

5.2.4 Convergence Comparison

Because UMDA/S, UMDA and GA are all population-based evolutionary algorithms, and the ultimate goal of search algorithms is to find the parameters with the maximum fitness, we test their convergence through comparing how the maximum fitness of each generation varies with generations. Figures 5 and 6 give the convergence comparison of MxM on PentiumD.

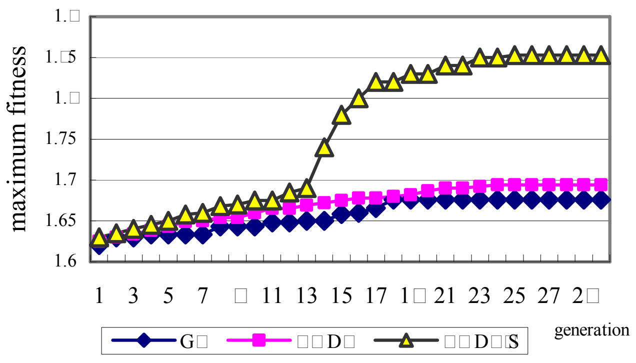

Fig. 5. Convergence comparison of MxM512 on PentiumD

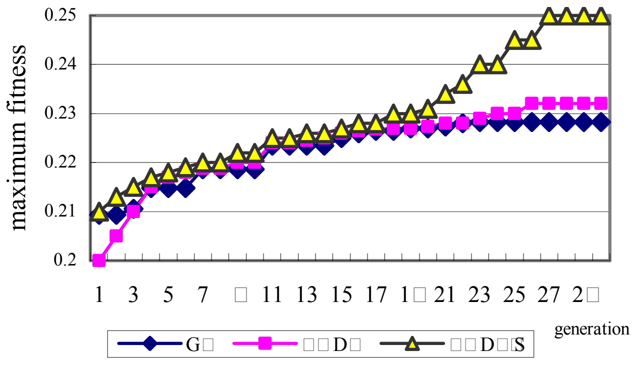

Fig. 6. Convergence comparison of MxM1024 on PentiumD

We can see from Figures 5 and 6 that UMDA has better learning ability than GA for optimization parameter vectors. It is worth mentioning that although the maximum fitness achieved by UMDA is lower than that of GA under some circumstances, UMDA performs better than GA gradually. For example, in Figure 6, during Generation 1 through 4, the maximum fitness searched by UMDA is lower than that by GA, but after Generation 5, the maximum fitness found by UMDA is higher than that by GA, i.e. UMDA performs better gradually. This is because UMDA/S adopts the simplex method to optimize parameter vectors generated by UMDA during initialization and sampling, which will facilitate locating the better-performing solutions more concisely, so as to better guide the evolutionary direction. In addition, three improvement policies are proposed in UMDA/S, which help avoid local convergence while improving the efficiency.

5.2.5 The Impact Analysis of Improvement Policies

To test the impact of different improvement policies, we eliminate elitist preservation, weighted estimation, mutation and all of them from UMDA/S, denoted by w/o ER, w/o WE, w/o mutation, and none, respectively, and test how the elimination affects the convergence of maximum fitness. The results of MxM on PentiumD are illustrated in Figures 7 and 8.

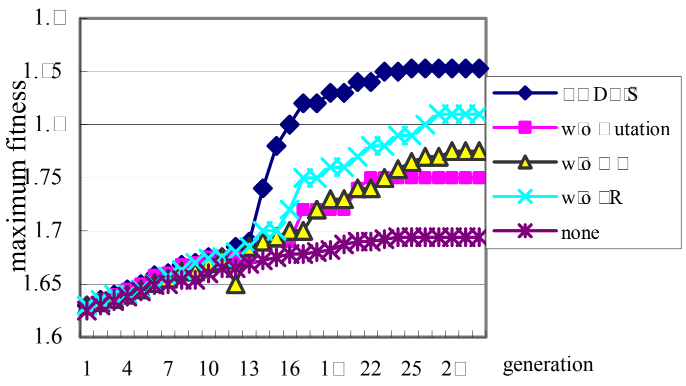

Fig. 7. The impact of improvement policies to MxM512 on PentiumD

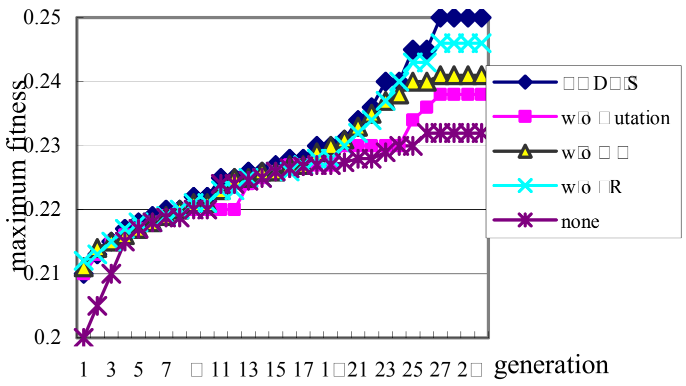

Fig. 8. The impact of improvement policies to MxM1024 on PentiumD

Experiments show that adding elitist preservation, weighted estimation, and mutation to UMDA/S can speed up the convergence and improve the quality of convergence. Among them mutation affects the convergence the most, weighted estimation less, and elitist preservation the least. This is because when generating optimization parameter vectors by sampling from the probability model of the selected population, optimization parameter vectors are gathering around excellent populations, thus the impact of elitist preservation is weakened. Weighted estimation also affects convergence, because it makes the optimization parameter vector with less execution time own bigger weight when building probability model. Thus excellent optimization parameter vectors play a more important role in the distribution estimation of next population, which improves the quality of the population. As for mutation, it should be mentioned that a shortcoming of EDAs is that, occasionally, solutions that are not very representative of the current model, but nevertheless they are good, are not well exploited in future iterations. This is because the information of such solutions is too different from that in the model. Therefore, such solutions' impact is of little importance. To overcome this limitation, the mutation operator was introduced. Due to the diversity provided by mutation operators, the algorithm is less likely to be trapped in a local minimum. In conclusion, the mutation affects UMDA/S' performance most, especially in the latter part of the search.

6 CONCLUSION AND FUTURE WORK

Present compilers face various difficulties to model the complex interplay between different optimizations and their effects on code on all the different processor architecture components. Iterative compilation approach has become a practical and portable means to implement architecture-aware optimizations for high-performance applications, boosting the performance of optimizing compilers and bridging the performance gap for high-performance applications.

This paper explores UMDA/S: the hybrid of UMDA and simplex algorithm, to search the near optimal parameter value in iterative compilation, and presents the improvement policies of elitist preservation, weighted estimation, and mutation to improve the performance of UMDA/S. Experimental results show that UMDA/S can be an effective strategy for the transformation parameter space exploration. Its ability to discover better solutions while keeping good convergence makes it a good choice for iterative compilation search strategies. In the future, we plan to improve UMDA/S by combining static models and architectural information to prune the search space and to use training data sets during the tuning process to cut down the program execution time.

Acknowledgement

We thank the anonymous reviewers for their constructive comments and suggestions. This work was partially supported by the National Grand Fundamental Research 973 Program of China under Grant No. G2009CB723803, 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] Apan Qasem: Automatic Tuning of Scientific Applications. Ph. D. thesis, RICE UNIVERSITY, 2007.

[2] Cohen, A.—Donadio, S.—Garzaran, M.-J.—Padua, D.—Herrmann, C.: In Search for a Program Generator to Implement Generic Transformations for High-performance Computing. 1st MetaOCaml Workshop (associated with GPCE), Vancouver, British Columbia, 2004.

[3] Knijnenburg, P. M. W.—Toru Kisuki—O'Boyle, M. F. P.: Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. The Journal of Supercomputing, Vol. 24, 2003, No. 1, pp. 43–67.

[4] Knijnenburg, P. M. W.—Toru Kisuki—Gallivan, K.—O'Boyle, M. F. P.: The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. Concurrency and Computation: Practice and Experience, Vol. 16, 2004, No. 2, pp. 247–270.

[5] Knijnenburg, P.—Kisuki, T.—Boyle, M. F. P.: Iterative Compilation. Embedded Processor Design Challenges System Architecture, Modeling and Simulation (SAMOS), Lecture Notes in Computer Science 2268, Springer Verlag, 2002, pp. 171–187.

[6] Fursin, G.—O'Boyle, M. F. P.—Knijnenburg, P. M. W.: Evaluating Iterative Compilation. Proceedings of Languages and Compilers for Parallel Computing (LCPC '02), 2002, pp. 305–315.

[7] Chen, C.—Chame, J.—Hall, M.: Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy. In International Symposium on Code Generation and Optimization, San Jose, CA, 2005.

[8] Jaume Abella, Bastoul, C.—B'echennec, J.-L.—Drach, N.—Eisenbeis, Ch.—Feautrier, P.—Franke, B.—Fursin, G.—Gonzalez, A.—Toru Kisku—Knijnenburg, P. M. W.—Llosa, J.—O'Boyle, M. F. P.—S'ebot, J.—Vera, X.: Guided Transformations. REPORT M3.D2, February 2001.

[9] Larrañnaga, P.—Lozano, J. A. eds.: Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Boston, MA: Kluwer, 2002.

[10] Zhang, Q.—Mühliebei, H.: On the Convergence of a Class of Estimation of Distribution Algorithms. IEEE Trans on Evolutionary Computation, Vol. 8, 2004, No. 2, pp. 127–136.

[11] Apan Qasem—Kennedy, K.—Mellor-Crummey, J.: Automatic Tuning of Whole Applications Using Direct Search and a Performance-Based Transformation System. Proceedings of the LACSI Symposium, 2004, pp. 183–194.

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

[13] Hauschild, M.—Pelikan, M.—Lima, C.—Sastry, K.: Analyzing Probabilistic Models in Hierarchical BOA on Traps and Spin Glasses. Proceedings of the genetic and evolutionary computation conference GECCO-2007, Vol. I. London, UK, ACM Press, New York, 2007, pp. 523–530.

[14] Mühlenbein, H.—Mahnig, T.—Ochoa, A.: Schemata, Distributions and Graphical Models in Evolutionary Optimization. J. Heuristics, Vol. 5, 1999, No. 2, pp. 213–247.

[15] Santana, R.—Larrañaga, P.—Lozano, J. A.: Interactions and Dependencies in Estimation of Distribution Algorithms. Proceedings of the 2005 congress on evolutionary computation CEC-2005, Edinburgh, U.K.,IEEE Press, 2005, pp. 1418–1425.

[16] Soto, M. R.—Ochoa, A.: A Factorized Distribution Algorithm Based on Polytrees. Proceedings of the 2000 congress on evolutionary computation CEC-2000, La Jolla Marriott Hotel La Jolla, California, USA, IEEE Press, July 6–9, 2000, pp. 232–237.

[17] Santana, R.—Larrañaga, P.—Lozano, J. A.: Research Topics in Discrete Estimation of Distribution Algorithms Based on Factorizations. Memetic Comp., Vol. 1, 2009, pp. 35–54.

[18] Mühlenbein, H.—Mahnig, T.: Evolutionary Optimization and the Estimation of Search Distributions With Applications to Graph Bipartitioning. Int. J. Approx. Reason, Vol. 31, 2002, No. 3, pp. 157–192.

[19] Santana, R.—Larrañaga, P.—Lozano, J. A.: Combining Variable Neighborhood Search and Estimation of Distribution Algorithms in the Protein Side Chain Placement Problem. J. Heuristics, Vol. 14, 2008, pp. 519–547.

[20] Nelder, J. A.—Mead, R.: A Simplex Method for Function Minimization. The Computer Journal, Vol. 8, 1965, pp. 308–313.

[21] Pingfan, Y.—Changshui, Zh.: Artificial Neural Networks and Evolutionary Computing (in Chinese). Qinghua university press, 2005, pp. 612.

[22] Lam, M. S.—Rothberg, E. E.—Wolf, M. E.: The Cache Performance and Optimizations of Blocked Algorithms. Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 1991, pp. 63–74.

[23] Coleman, S.—McKinley, K.: Tile Size Selection Using Cache Organization and Data Layout. Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI), June 1995, pp. 279–290.

[24] Lu, P.—Che, Y.—Wang, Zh.: An Effective Iterative Compilation Search Algorithm for High Performance Computing Applications. 10th IEEE International Conference on High Performance Computing and Communications (HPCC), 2008, pp. 368–373.

Pingjing Lu received her B. S. degree and M. Sc. degree in computer science at the National University of Defense Technology in China in 2004 and 2006 respectively. Since 2006, she has been a Ph. D. student in computer science at the National University of Defense Technology. Her current research interests include iterative compilation and adaptive optimization.

Yonggang Che received his B. Sc. degree, M. Sc. degree and Ph. D. degree in computer science at the National University of Defense Technology in China in 1997, 1999, and 2004 respectively. Since 2006 he has been an associate professor of computer science at the National University of Defense Technology. His current research interests include computer architecture and iterative compilation.

Zhenghua Wang received his B. Sc. degree, M. Sc. degree and Ph. D. degree at the National University of Defense Technology in China in 1983, 1986, and 1991 respectively. He has been a professor of computer science at the National University of Defense Technology since 2000. His research interests are in computer systems performance evaluation and compiler optimization.