Computing and Informatics, Vol. 30, 2010, 943–963

# PIT: A FRAMEWORK FOR EFFECTIVELY COMPOSING HIGH-LEVEL LOOP TRANSFORMATIONS

Pingjing Lu, Bao Li, Yonggang Che, Zhenghua Wang

*School of Computer*
*National University of Defense Technology*
*Changsha 410073, China*
*e-mail:* pingjinglu@gmail.com

Communicated by Ján Kollár

**Abstract.** The increasing complexity of modern architectures and memory models challenges the design of optimizing compilers. It is mandatory to perform several optimizing transformations of the original program to exploit the machine to its best, especially for scientific, computational-intensive codes. Aiming at investigating the best transformation sequence and the best transformation parameters simultaneously, this paper presents a novel loop transformation framework, which integrates the advantages of polyhedral model and model-guided iterative compilation to create a powerful framework that is capable of fully automated non-parametric transformations and model-guided parametric transformations as well as automatic parameter search. The framework employs polyhedral model to facilitate the search of non-parametric code transformation composition, and designs a transformation model based on hardware performance counters to guide when, where and in what order to apply transformations to get the most benefit, finally uses Nelder-Mead simplex algorithm to find the optimal parameters. The framework is demonstrated on three typical computational kernels for code transformations to achieve performance that greatly exceeds the native compiler, and is significantly better than state-of-the-art polyhedral model based loop transformations and iterative compilation, generating efficient code on complex loop nests.

**Keywords:** Program optimization, loop transformation, polyhedral model, iterative compilation

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

## 1 INTRODUCTION

Although loop transformations have been applied by compilers for many years, certain problems with the application of transformations have yet to be addressed, including when, where and in what order to apply transformations to get the most benefit, as well as the selection of optimal transformation parameters.

Existing compilers are ill-equipped to address these challenges, because of improper program representations and inappropriate conditioning of the search space structure. They are based on static analysis and a hardwired compilation strategy; therefore they only uncover a fraction of the peak performance on typical benchmarks. Iterative compilation [1, 2, 3, 4] is a maturing framework to address these limitations, but so far, it was not successfully applied because present day iterative compilation approaches select the optimal transformation parameters at a predefined transformation sequence, and because of the high cost due to multiple, costly "runs" and the combinatorics of the optimization space. The ability to perform numerous compositions of program transformations is key to the extension of iterative optimizations to finding the appropriate program transformations instead of just the appropriate program transformation parameters. The polyhedral model is a well studied, powerful mathematical framework to represent loop nests and their transformations [5, 6, 7, 8, 9], facilitating compilers to compose complex loop transformations in a mathematically rigorous way to insure code correctness. However, existing polyhedral frameworks are often too limited in supporting a wide array of loop transformations required to achieve high performance on today's computer architecture, and they do not allow exploring jointly the best sequence of transformations and the best value of transformation parameters. Usually, the community tries to find the "best" parameter combination when the transformation sequence is fixed [10]. Clearly, there is a need for the infrastructure that can apply long compositions of transformations and find the best transformation parameters in a rich, structured search space.

This paper presents a loop transformation framework PIT (combining Polyhedral model and ITerative compilation for loop transformations, PIT) to simultaneously explore the best sequence of transformations and the best value of transformation parameters. It integrates polyhedral model and model-guided iterative compilation to create a powerful framework that is capable of fully automated non-parametric code transformations and model-guided parametric transformations as well as automatic parameter search. Experimental results show that compared with present day loop transformation frameworks, PIT can optimize applications more efficiently.

## 2 FORMAL DESCRIPTION

Let $P$ be the source program, $\tau$ an arbitrary performance evaluation function (not limited to program execution time, it can be cache miss rate etc.), $\Psi$ a finite set of loop transformations, including $l$ parametric transformation modules

$\varphi_i \in \Psi(i = 1, 2, \ldots, l)$. Denote $\circ$ the transformation joint symbol, then applying a finite sequence $\varphi_1, \ldots, \varphi_n$ of transformation modules to $P$ can be represented as $\phi = \varphi_n \circ \varphi_{n-1} \circ \ldots \circ \varphi_1$, and all the sequences form optimization sequence space $\Phi$. Parametric module $\varphi_i$ contains $m_i$ transformation parameters: $p_{ik} \in \mathrm{Z}(k = 1, 2, \ldots, m_i)$, and its upper bound $up_{ik} \in \mathrm{Z}$ and lower bound $low_{ik} \in \mathrm{Z}$ can be achieved based on domain-specific information. Denote $v = \sum_{i=1}^{l} m_i$, then the optimization parameters $(p_{11}, \ldots, p_{1m_1}, \ldots, p_{l1}, \ldots, p_{lm_l})$ of all parametric transformation modules constitute optimization parameter vector $\overrightarrow{K} \in \mathrm{Z}^v$. Applying transformation sequence $\phi$ to $P$ and adopting optimization parameter vector $\overrightarrow{K}$ results in the program $P' = \phi(P, \overrightarrow{K})$, and the corresponding test result is $\tau(P, \phi, \overrightarrow{K})$. Then the optimal loop transformations problem converts to a combinational optimization problem: having a program $P$ and a set of loop transformation modules $\Psi$, how to select the optimal transformation sequence $\phi^*$ and the optimal parameter vector $\overrightarrow{K}$, such that the performance of the final generated program $P^*$ is "optimal", i.e.

$$(\phi^*, \overrightarrow{K}^*) = \arg\min \tau(P, \phi, \overrightarrow{K}) \quad \text{subject to} \begin{cases} \phi \in \Phi \\ \overrightarrow{K} \in \mathrm{Z}^v \end{cases} \tag{1}$$

where arg min means that $(\phi^*, \overrightarrow{K}^*)$ are the optimal values of parameters $\phi$ and $\overrightarrow{K}$ that minimize the object function $\tau(P, \phi, \overrightarrow{K})$, and *subject to* introduces the requirements that $\phi$ and $\overrightarrow{K}$ have to satisfy.

## 3 POLYHEDRAL MODEL

```
for (i = 0; i <= M; i++) {
  for (j = 0; j <= M; j++) {
    S1: C[i][j] = 0;
      for (k = 0; k <= M; k++) {
    S2: C[i][j]=C[i][j]+A[i][k]* B[k][j];}}}
```

Fig. 1. Code for matrix multiplication program

The polyhedral model is a unified mathematical framework to represent loop nests and their transformations. It represents the code through the iteration domain, affine schedules, and array access functions [11, 6, 7]. We will briefly introduce polyhedral model through matrix multiplication program.

### 3.1 Iteration Domain

*Iteration domain* is a geometrical abstraction of loop bounds and strides shaping loop structures. The loop control statements surrounding statement $S$ form itera-

tion domain $D^S$. It can be defined through a set of affine inequalities, which form the parametric polyhedra. Each point in the polyhedra stands for one execution instance. Iteration domain depends on surrounding loop counters and global parameters (e.g. loop bounds). For example, in Figue 1, surrounding loop counters of statement $S_2$ is $i$, $j$ and $k$, and the scope of loop is bounded by $(M, M, M)$; therefore, the iteration domain of $S_2$ can be represented as Equation (2), where $(i, j, k)$ is called an iteration vector, and $(M)$ is called a global parameter.

$$D^{S_2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ M \\ 1 \end{pmatrix} \geq \overrightarrow{0} \tag{2}$$

where $\overrightarrow{0}$ is a vector, and in Formulae (2) $\overrightarrow{0} = (0, 0, 0, 0, 0)^t$.

### 3.2 Array Access Functions

*Array access functions* capture the data locations on which a statement operates. In polyhedral model, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts of the form of affine expressions which may depend on surrounding loop counters (e.g., $i$, $j$ and $k$ for statement $S_2$) and global parameters. Each array access function is linked to an array that represents a read or a write access.

$L^S$ and $R^S$ are sets of polyhedral representations of array references, describing array references written by $S$ (left-hand side) or read by $S$ (right-hand side), respectively; it is a set of pairs $(A, f)$ where $A$ is an array variable and $f$ is the access function mapping iterations in $D^S$ to locations in $A$. For example, in Figure 1, $S_2$ reads $A[i][k]$, $B[k][j]$ and $C[i][j]$, and writes the result to $C[i][j]$, so the array access functions of $S_2$ are:

$$R^{S_2} = \left\{ \left( A, \begin{bmatrix} 10000 \\ 00100 \end{bmatrix} \right), \left( B, \begin{bmatrix} 00100 \\ 01000 \end{bmatrix} \right), \left( C, \begin{bmatrix} 10000 \\ 01000 \end{bmatrix} \right) \right\} \tag{3}$$

$$L^{S_2} = \left\{ \left( C, \begin{bmatrix} 10000 \\ 01000 \end{bmatrix} \right) \right\}. \tag{4}$$

### 3.3 Affine Scheduling

Iteration domains define exactly the set of dynamic instances for each statement. However, this algebraic structure does not describe the order in which each instance has to be executed with respect to other instances. Of course, we do not want to rely on the inductive semantics of the sequence and loop iteration for this purpose, as it would break the algebraic reasoning about loop nests. A convenient way to

express the execution order is to give each instance an execution date [7, 8]. It is obviously impractical to define all of them one by one since the number of instances may be either very large or unknown at compile time. An appropriate solution is to define, for each statement, a scheduling function that specifies the execution date for each instance of a corresponding statement. This work deals with multidimensional schedules: multidimensional dates can be seen as clocks: the first dimension corresponds to days (most significant), the next one is hours (less significant), the third to minutes, and so on.

$\theta^S$ is the *affine schedule* of S; it is another geometrical abstraction of the ordering of iterations and statements which maps iterations in $D^S$ to multidimensional time stamps, i.e., logical execution dates. Multidimensional time stamps are compared through the lexicographic ordering over vectors, denoted by $\ll$: iteration $\overrightarrow{i}$ of $S$ is executed before iteration $\overrightarrow{i}'$ of $S'$ if and only if $\theta^S(\overrightarrow{i}) \ll \theta^{S'}(\overrightarrow{i}')$. In Figure 1 the affine schedules of $S_1$ and $S_2$ are $\theta^{S_1}(\overrightarrow{i}) = (0, i, 0, j, 0)$, $\theta^{S_2}(\overrightarrow{i}) = (0, i, 0, j, 1, k, 0)$. $\theta^{S_1}(\overrightarrow{i}) \ll \theta^{S_2}(\overrightarrow{i})$, therefore, $S_1$ executes before $S_2$.

Assume the loop nest includes $d$ statements, the schedule dimension is $s$, the iteration vector is $\overrightarrow{x}$, the global parameter is $\overrightarrow{n}$; then each program version can be represented as one point in the optimization space through schedule matrix $\Theta$ [7]:

$$\Theta\overrightarrow{x} = \begin{pmatrix} \overrightarrow{i}^1_1 \dots \overrightarrow{i}^1_d \overrightarrow{p}^1_1 \dots \overrightarrow{p}^1_d c^1_1 \dots c^1_d \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \overrightarrow{i}^s_1 \dots \overrightarrow{i}^s_d \overrightarrow{p}^s_1 \dots \overrightarrow{p}^s_d c^s_1 \dots c^s_d \end{pmatrix} \begin{pmatrix} \overrightarrow{x}_1 \\ \vdots \\ \overrightarrow{x}_d \\ \overrightarrow{n}_1 \\ \vdots \\ \overrightarrow{n}_d \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{5}$$

We can see that in polyhedral model each loop transformation corresponds to a set of matrix operations. An arbitrary complex loop transformation sequence can be applied within one step. Searching the compositions of transformations is equivalent to searching the matrix parameters; therefore, polyhedral model avoids the typical code complexity explosion of long compositions of program transformations [7, 8], facilitating the composition of complex loop transformations in a mathematically rigorous way.

## 4 PIT: COMBINING POLYHEDRAL MODEL AND ITERATIVE COMPILATION FOR LOOP TRANSFORMATIONS

This paper presents a novel loop transformation framework PIT to search the best transformation sequence and best transformation parameters. The framework includes three phases as illustrated in Figure 2, which are described in Section 4.1,

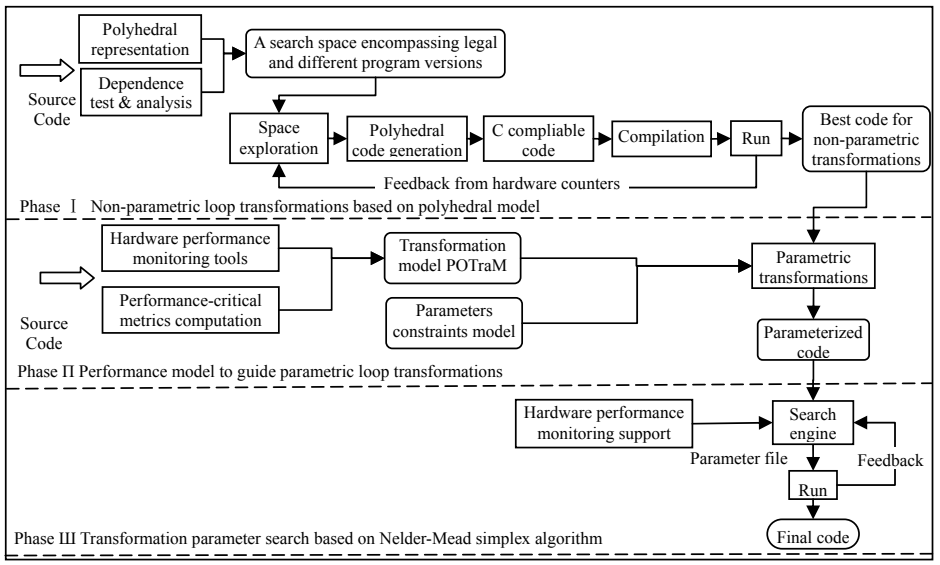



Fig. 2. Overview of three-phase loop transformation optimization framework PIT

Section 4.2, and Section 4.3, respectively. Phase 1 utilizes polyhedral model to find the optimal sequence of non-parametric transformations. We introduce LeTSeE (the LEgal Transformation SpacE Explorator) [7] to find the optimal sequence of non-parametric transformations in the first phase as described in Section 4.1. Phase 2 first utilizes hardware performance counters to collect the accurate information on the program behavior, and achieve a transformation model POTraM (Program Optimization Transformation Model based on Hardware Performance Counters, POTraM) [12] as described in Section 4.2. The program transformation model POTraM in Section 4.2.1 is inherited and extended from my previous work [12]. Section 4.2.2 is initially presented and added to our optimization framework. Phase 3 employs Nelder-Mead simplex method to select the optimal transformation parameters [13], and finally achieves the best transformation parameters in the best transformation sequence, which is inherited from my previous work [13]. The combination of these three phases enables PIT to simultaneously explore the best sequence of transformations and the best value of transformations parameters, creating a powerful framework that is capable of fully automated nonparametric code transformations and model-guided parametric transformations as well as automatic parameter search.

### 4.1 Phase 1: Non-Parametric Loop Transformations

Polyhedral model can analyze the dependence exactly, facilitating the legality check of loop transformations; therefore, it is utilized to find the optimal sequence of non-parametric loop transformations in the first phase of PIT. This paper makes

use of polyhedral transformation tools LeTSeE to apply iterative compilation based on polyhedral model and to find the best non-parametric transformation sequence. LeTSeE first builds a search space encompassing legal and distinct program versions, thanks to its algebraic representation, and then traverses the search space, where each point represents a different program version. For each tested point in the search space, it

1. generates the kernel C code with CLooG [9],
2. then integrates this kernel in the original benchmark along with instrumentation to measure running time,
3. compiles this code with the native compiler and appropriate options,
4. finally runs the program on the target architecture and gathers performance results, using the information collected to drive the exploration according to user objectives.

The transformations implemented in LeTSeE include statement reordering, loop reversal, loop skewing, loop interchange, loop peeling, index-set splitting, loop pipelining/shifting, loop fusion and loop distribution. LeTSeE will generate the best non-parametric transformation sequence.

### 4.2 Phase 2: Models for Parametric Transformations

Our models include program transformation model POTraM and constraints on transformation parameters.

#### 4.2.1 Program Transformation Model POTraM

| **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 |

Table 1. Performance counters used

Performance counters are a special set of registers that allow for measuring performance counter events with no disruption to the running program. The information obtained from them is a compact summary of a program's dynamic behavior. Performance counters have been extensively used for performance analysis in explaining program behavior. This paper uses PAPI (Performance Application Programming Interface) [14] to access performance counters. The performance counters used in this study are shown in Table 1. Next we will illustrate how POTraM is achieved based on the characterization of important properties of programs.

1. *L1 data cache miss rate:*

$$MSR_{L1D} = PAPI\_L1\_DCM/PAPI\_L1\_DCA. \quad (6)$$

2. *Program Balance and Machine Balance.* Steve Carr [15] utilized Loop Balance ($BL$) and Machine Balance ($BM$) of floating-point applications to guide program optimization. They defined if $BL > BM$, then the loop needs data at a higher rate than the machine provides, and idle computational cycles exist. Such a loop is called memory bound and its performance can be improved by lowering $BL$ through memory optimization. $BL$ and $BM$ are designed for specific memory hierarchy. Because our test platform has two cache levels, there exist 3 levels' balance: CPU to L1 data cache (i.e. Hierarchy 1), L1 data cache to L2 data cache (i.e. Hierarchy 2), and L2 data cache to main memory (i.e. Hierarchy 3). Let $w_1$,$w_2$,$w_3$ be L1 data cache bus width, L1 data cache line size and L2 data cache line size, respectively; then $BL$ can be computed as follows:

$$BL_1 = (PAPI\_L1\_DCA * w_1)/PAPI\_FP\_INS \quad (7)$$
$$BL_2 = (PAPI\_L1\_DCM * w_2)/PAPI\_FP\_INS \quad (8)$$
$$BL_3 = (PAPI\_L2\_DCM * w_3)/PAPI\_FP\_INS \quad (9)$$

3. *Performance influence ratio.* In modern processors, the memory and instruction pipeline utilization of programs influences performance most. Mo et al. [16] present the concepts of Influence Ratio for Memory Reference ($\eta_m$) and Influence Ratio for Pipeline ($\eta_p l$) to quantify the impact of memory and pipeline operations. $\eta_m$ quantifies the impact of cache miss to performance. The lower the cache miss rate, the less $\eta_m$ will be. $\eta_{pl}$ quantifies the impact of instruction level parallelization to performance. The higher reuse ratio of the operators in the register, the less $\eta_{pl}$ will be. $\eta_{fp}$ describes the utilization ratio of peak performance on single processors, which depends on $\eta_m$ and $\eta_{pl}$. Let $C_1$ and $C_2$ be the cost of L1 cache and L2 cache miss (in cycles), $F$ the main frequency (in Hz), $G$ peak performance (in Mflops), $T$ program running time (in seconds), and $T_m$ the total cost of cache miss (in seconds); then $\eta_m$, $\eta_{pl}$ and $\eta_{fp}$ can be achieved as follows:

$$T_m = (PAPI\_L1\_TCM * C_1 + PAPI\_L2\_TCM * C_2)/F \quad (10)$$
$$\eta_m = T_m/T \quad (11)$$
$$\eta_{pl} = 1 - PAPI\_FP\_INS/(G*(T - T_m)) \quad (12)$$
$$\eta_{fp} = PAPI\_FP\_INS/(G*T). \quad (13)$$

4. *POTraM.* Based on these characterization data, transformation model POTraM, as described in Figure 3, is proposed to provide source-code level feedback about when, where and in what order to apply transformations to get the most benefit,

ExtractTransformationModel (*src*)

**Input**: *src*, which is the source of program to be tuned

**Output**: *X*, which is a vector of transformation parameters, and *src*, which is the transformed code of source program

**1. Initialization.**

$N \leftarrow$ the number of dimensions of the loop, assuming the loops are $l_1, l_2, \ldots, l_N$;

$X \leftarrow \varnothing$ ;

Run, collect and compute hardware performance information of *src*, get $MSR_{L1D}$, $BM_i$, $BP_i$ (i=1,2,3), $\eta_m$, $\eta_{pl}$ and $\eta_{tlp}$;

**2. Program optimization based on Cache miss rate and program balance**

***if*** $(MSR_{L1D}>5\%) \| (\exists i(i=1,2,3))(BP_i>BM_i))$ {

*src* ← *ApplyLoopExtrange* (*src*);}

**3. Program optimization based on performance influence ratio**

***if*** $(\eta_m > 30\%)$ {

*src* ← *ApplyLoopTiling* (*src*);

*src* ← *ApplyArrayPadding* (*src*);

$X \leftarrow (t_1, t_2, \ldots, t_N, p_1, p_2, \ldots, p_N)$, where $t_1, t_2, \ldots, t_N$ and $p_1, p_2, \ldots, p_N$ are factors of loop tiling and array padding respectively;}

***else if*** $(\eta_m > 20\%)$ {

*src* ← *ApplyLoopTiling* (*src*);

$X \leftarrow (t_1, t_2, \ldots, t_N)$ ;}

***if*** $(\eta_{pl} > 80\%)$ {

*src* ← *ApplyInnermostLoopUnrolling* (*src*);

$X \leftarrow X \circ (u)$, where $\circ$ denotes the symbol of function combination (concatenation), and $u$ is inner most loop unrolling factor;}

**4. Terminate.**

*src** ← *src*.

Fig. 3. Algorithm for POTraM

therefore guides the restructuring of program loops to achieve high performance on specific target architectures. To optimize a program *src*, first run the program and collect the aforementioned hardware performance information using hardware performance counters. Then decide the type of transformation to optimize *src* based on these characterization data. If L1 data cache miss rate is greater than 5 %, or the Loop Balance is greater than Machine Balance, such a loop is called memory bound and its performance can be improved by loop exchange transformation. If Influence Ratio for Memory Reference is greater than 30 %, which indicates that the memory performance is quite poor and there may be cache conflict misses, then *src* needs array optimization and data layout transformation, so loop tiling and array padding are applied to *src*; if Influence Ratio for Memory Reference is greater than 20 % but less than 30 %, then only apply loop tiling to *src*. Further, if Influence Ratio for Pipeline is greater than 80 %, it indicates that the impact of instruction level parallelization to performance is great, and the operators in the register are not fully reused; then apply loop unrolling to improve the reuse ratio of the operators

in the register. Up to now, the transformation set is achieved, and in the next phase apply the search algorithm to find the optimal value for the transformation set.

#### 4.2.2 Optimization Parameter Constraints Model

Based on empirical knowledge about architecture and programs, a set of constraint information about transformation parameters can be achieved. The parametric transformations considered in this paper are loop tiling, loop unrolling and array padding, and the parameter constraints are given in the following.

Loop unrolling factors are mainly restricted by registers. Too small unrolling factors cannot fully utilize registers, but too large ones may generate many spills to memory and decrease performance. Denote $U_1, U_2, \ldots, U_n$ loop unrolling factors, $n$ the depth of loop nest, $N_R$ the number of registers, and $a_i$ the register number occupied by the data in loop $L_i$, which is a program-specific constant; then the constraints on unrolling factors can be summarized as follows:

$$\begin{cases} a_1U_1 * a_2U_2 * \ldots * a_nU_n \leq N_R \\ 1 \leq U_i \leq N_R \quad (U_i \in \mathrm{Z}, i = 1, \ldots, n). \end{cases} \tag{14}$$

Loop tiling factors are mainly restricted by cache size, because the working sets after tiling should fit in the cache. Choosing too large or too small tiling factors increases the L1 cache miss ratio and leads to inefficient cache utilization. We set the lower bound of tiling factors as 16. Denote $T_1, T_2, \ldots, T_n$ tiling factors, and $C$ L1 cache size, then constraints on tiling factors can be summarized as follows:

$$\begin{cases} T_1 * T_2 * \ldots * T_n \leq C \\ 16 \leq T_i \leq C \quad (T_i \in \mathrm{Z}, i = 1, \ldots, n). \end{cases} \tag{15}$$

Array padding scheme is to remove cache conflict misses; however, it will lead to additional memory cost, and too big padding factors will increase TLB miss. It is pointed out that small padding factors are more liable to achieve high performance in [17]. Therefore, we limit the range of padding factors as $4 \sim 64$, which means the upper limit that padding factors can value is 64, and the lower limit that padding factors can value is 4, and they cannot exceed the range.

The Nelder-Mead simplex method [18, 19] is a classical and powerful direct search method for optimization. It is probably the most widely used optimization method. A "simplex" is a geometrical figure consisting of $n+1$ points in $n$-dimensions, e.g. a 2-dimension simplex is a triangle. Through a sequence of geometric transformations, the initial simplex moves towards minimum. It appears to be a good fit to the problem of finding optimization parameters to minimize program runtime. First, the optimization space is too large to search completely when several optimizations are applied. Second, our objective function is discrete and nonlinear, making the problem difficult to address with classical combinatorial optimization techniques. Third, simplex method is useful for training parameters, especially for

searching minima of multi-dimensional functions when dimension is less than 20. Finally, the amount of time spent by simplex method is flexible. More computation time may result in better solutions, but the algorithm can be interrupted any time and return the best solution found. Thus the simplex method is used to solve the problem of optimization parameters selection.

We first list some notations in the simplex method.

- $S^{(k)}$: The simplex in the $k^{th}$ iteration, and $S^{(k)} = (x_0^k, x_1^k, \ldots, x_n^k)$;
- $x_h^k$: The highest (worst) point, i.e. $f(x_h^k) = \max\{f(x_i^k)|i = 0, 1, \ldots, n\}$;
- $x_{\inf h}^k$: The second highest point, i.e. $f(x_{\inf h}^k) = \max\{f(x_i^k)|i = 0, 1, \ldots, n,$ and $i \neq h\}$;
- $x_l^k$: The lowest (best) point, i.e. $f(x_l^k) = \min\{f(x_i^k)|i = 0, 1, \ldots, n\}$;
- $\bar{x}^k$: Average of all points excluding the worst point;
- $\max iter$: The maximum iterations;
- $\varepsilon$: The precision requirement;
- $\alpha, \beta, \gamma, \omega$: The coefficients of reflection, contraction, expansion, and shrink.

The Nelder-Mead simplex based parameter selection algorithm is shown in Figure 4.

### 4.3 Phase 3: Search for Optimal Parameters

Note that the most time-consuming part of the algorithm is the measuring of the execution time. At least the set of parameters with the minimum execution time remains unchanged from one generation to the next, so there is no need to recalculate its execution time. 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.

## 5 PERFORMANCE EVALUATION

### 5.1 Environmental Setup

We test three typical numerical compute-intensive kernels: the matrix multiplication program (*mm*) with matrix sizes 512 and 1 024, DSP kernel *fir* [20] abstracted from UTDSP benchmark suit with size 10 000 and 20 000, and kernel *mvt* which includes two matrix vector multiplication statements and these two matrices are transpose of each other with size 1 024 and 2 048. For brevity we name the program with small size as the name of program followed by 1, and that with large size are followed by 2. For example, programs *mm* with size 512 and 1 024 are named as

**NMSimplex (***testFile*, *n* **)**

**Input:** *testFile*, which is the source of program to be tuned, and *n*, the number of optimization parameters;

**Output:** $x^*$, the optimal optimization parameter vector, and $T(x^*)$, the minimum running time of the program.

1.**Initialization**. For each parameter, we define an empirical range $\Omega$. Randomly generate a non-degenerate initial simplex $S^{(0)}$ that belongs to $R$ on $Z_n$, then do a measurement at each point. Set parameters: $\max iter$, $\varepsilon$, $\alpha, \beta, \omega, \gamma$, and set $k = 1$.

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

3. **Reflection**. $x_r^k = (1+\alpha)\overline{x}^k - \alpha\overline{x}^k$ where $\alpha > 0$. If any parameter of $x_r^k$ outreaches the range $\Omega$, 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 $\Omega$, 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 $\Omega$, 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* { $x_c^k = \beta x_n^k + (1-\beta)\overline{x}^k$, where $0 < \beta < 1$, if any parameter of $x_c^k$ outreaches the range $\Omega$, 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 $0 < \omega < 1$, $i = 0,1,\ldots,n,\ i \ne l$.

7. **Stop criterion check.**

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

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

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

Fig. 4. The Nelder-Mead simplex based parameter selection algorithm

*mm1* and *mm2*, respectively. Experiments are performed on Intel Pentium D 820 platform described in Table 2. $BM_i$ $(i = 1, 2, 3)$, $C_i$ $(i = 1, 2)$ and bus speed of L1 cache, L2 cache and memory are calculated using the method presented by Jack Dongarra in [21]. Parameter settings for simplex are as follows: *MaxIter* = 10, $\varepsilon = 0.0001$. Recommended values of $\alpha$, $\beta$, $\gamma$, $\omega$ are $\alpha = 1.0$, $\beta = 0.5$, $\gamma = 2.0$, $\omega = 0.5$ [18].

| CPU | Intel Pentium D 820 2.8 GHz |
|---|---|
| L1Data cache | $2 \times 16$ (KB) |
| L1 Instruction cache | $2 \times 12$ (KB) |
| L2 cache | $2 \times 1\,024$ (KB) |
| Memory | DDR2 1G |
| OS | Ubuntu kernel 2.6.15-23-386 |
| Compiler | gcc 4.2.1 -O3 -Dtest_malloc -lm |
| $BM_1/BM_2/BM_3$ | 16/32/2.3 |
| Bus speed of L1 cache | 44.8 (GB/s) |
| Bus speed of L2 cache | 89.6 (GB/s) |
| Bus speed of Memory | 6.4 (GB/s) |
| $C_1/C_2$ | 4/31 (cycles) |

Table 2. Experimental platform

### 5.2 Experimental Results

#### 5.2.1 Effectiveness Verification of POTraM

We have tested programs' behavior. The experimental results and responding transformations (transfo) based on POTraM are listed in Table 3. $T$, $P$ and $U$ stand for loop tiling, array padding and loop unrolling, respectively. We can see that POTraM applies different transformations to the programs according to their characteristics. Before adopting POTraM, all programs will apply transformations $(T, P, U)$; after adopting the model, the transformations applied to *mm* remain $(T, P, U)$, but those applied to *mvt* and *fir* have been reduced; thus the optimization space is reduced because the search space grows as a function of the number of transformations.

| | mm1 | mm2 | mvt1 | mvt2 | fir1 | fir2 |
|---|---|---|---|---|---|---|
| $MSR_{L1D}$ (%) | 18.4 | 31.5 | 20.8 | 22.7 | 18.8 | 22.7 |
| $BL_1$ | 206 | 108 | 89 | 85 | 49 | 36 |
| $BL_2$ | 87 | 94 | 75 | 81 | 38 | 30 |
| $BL_3$ | 3.1 | 3.9 | 4.8 | 5.8 | 1.5 | 2.4 |
| $\eta_m$ (%) | 32.2 | 35.7 | 17.0 | 17.0 | 13.5 | 13.9 |
| $\eta_{pl}$ (%) | 90.2 | 91.1 | 90.0 | 9.0 | 68.9 | 37.7 |
| $\eta_{fp}$ (%) | 6.6 | 5.7 | 8.5 | 8.0 | 26.9 | 27.8 |
| Transfo | $(T, P, U)$ | $(T, P, U)$ | $(T, U)$ | $(T, U)$ | $(T)$ | $(T)$ |

Table 3. Performance data of original programs and corresponding transformations based on POTraM

To compare the quality of reduced set of loop transformations, we test the programs' performance with parameters searched under the original and reduced set of loop transformations. We also test the performance when $\max iter = 150$ and $\max iter = 300$. The original loop transformation sets of all programs with

no using of POTraM model are all $(T, P, U)$, and we notate this method as no-POTraM for the simplicity of description. When $\max iter = 150$, the reduced set of transformations will enable search algorithm to explore more values of performance-critical transformation parameters; therefore, they concentrate on more profitable transformations, and bring performance boost. When $\max iter = 300$ the search space has been doubled, thus more points in the optimization space will be searched. We want to verify how much performance will be lost by the reduced set under this circumstance. Figure 4 shows the speedup over no-POTraM in these two search spaces, where *Same search space* and *Expanded search space* stand for the results of $\max iter = 150$ and $\max iter = 300$, respectively.

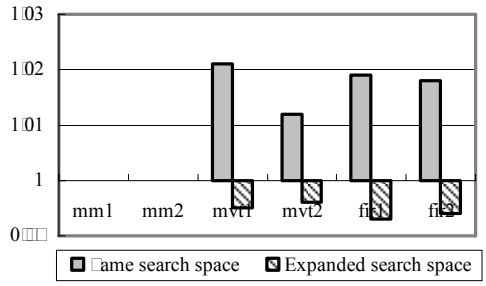



Fig. 5. Speedup over no-POTraM

From Figure 4, we can see that by adopting POTraM, the performance will be improved by 2 % when $\max iter = 150$, and when $\max iter = 300$, the performance loss is about 0.5 %; however, the search cost is almost doubled. The results indicate that the omitted transformations have minor impact on programs performance, and the performance gained with them is insignificant compared with the cost. Considering the tradeoff between the cost and the performance benefit, we decide not to apply them to the program. Results show that POTraM focuses the set of transformations on performance-critical ones, and the reduced set drastically narrows down the search space and yet achieves most of the performance benefit. With the POTraM model, higher quality code is achieved in less time than what is possible with full transformations.

### 5.2.2 Comparison with Related Work

We apply the following three strategies to test programs:

1. LeTSeE (represented with Poly), a non-parametric transformation system based on polyhedral model;
2. traditional parametric iterative compilation (represented with IC);
3. PIT, we compare their performance with that of original programs (represented with Orig).

We test programs' characteristics listed in Section 4.2.1, and compare the performance of three different optimization methods.

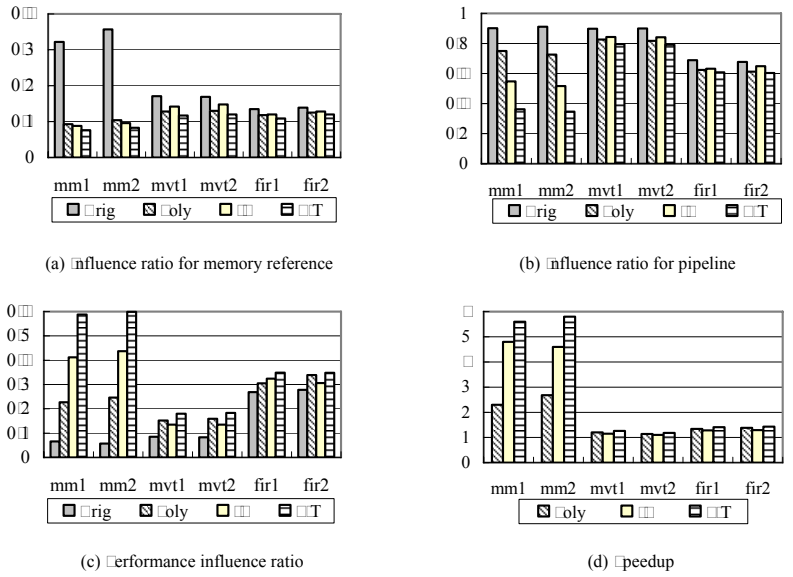



Fig. 6. Comparison of different optimization methods over state-of-the-art research

Figure 6 a) illustrates the influence ratio for memory reference of the programs. Three optimization methods all reduce $\eta_m$, therefore, programs' memory performance has increased greatly. The influence ratio for memory reference of program *mm* improves most, and IC improves more than Poly, because *mm* has high ratio of floating-point operations to memory operations, and loop tiling can effectively improve performance. The original $\eta_m$ of *mm* are greater than 30 %, demonstrating that the memory utilization of programs is very low, which will influence programs performance. In Section 5.2.1 we have achieved the transformation set of *mm* based on POTraM, which includes loop tiling and array padding. Loop tiling and array padding improve the memory performance greatly. The improvement of memory performance will result in the boost of program floating-point performance (Figure 6 d)). Figure 6 b) shows the performance influence ratio for pipeline of the programs. $\eta_{pl}$ of *mm* and *mvt* 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, $\eta_{pl}$ of *fir* are lower than 80 % (68.9 and 37.7); 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 (Figue 6 c)). The figure shows the performance influence ratio for floating-point operation of the programs. The original $\eta_{fp}$ of all test programs except *fir* are less than 10 %, after optimization,

$\eta_{fp}$ of programs improved up to 60 %. Figue 6 d) illustrates the benefits of PIT – the reduction of program execution time. Because the execution time of different programs varies greatly and it is hard to integrate all these programs' information in one graph, we normalize them as the speedup relative to original program, which equals the running time of original program divided by that of optimized program.

In Figue 6, we can see that program *mm* improves most, and IC improves more than Poly, because *mm* has high ratio of floating-point operations to memory operations, loop tiling can effectively improve performance. As to *mvt*, the loop body will access matrices in different lines and columns, tiling and unrolling will not bring significant improvements, while Poly can compose the transformations of loop skewing, loop distribution etc. Therefore, Poly improves programs' performance more effectively than IC. *fir* includes branch statements, so it is difficult to apply tiling, unrolling and padding to it, but polyhedral model based non-parametric transformations can effectively apply various transformations due to exact dependence test and analysis, more effectively optimizing programs. Figure 5 d) shows that although Poly and IC adapt to different programs, PIT – the combination of them improves performance most.

The results show that PIT can effectively improve programs' floating-point performance, reducing programs' runtime; therefore, it lessens the performance gap for high-performance applications. Meanwhile the optimization space has been reduced using performance model POTraM. Experiments validate that through the combination of Polyhedral model and iterative compilation for loop transformations, PIT can simultaneously explore the best sequence of transformations and the best value of transformations parameters, creating a powerful framework that is capable of fully automated nonparametric code transformations and model-guided parametric transformations as well as automatic parameter search. Experimental results show that compared with present day loop transformation frameworks, PIT can optimize applications more efficiently; therefore, it can be a practical and portable means to implement architecture-aware optimizations for high-performance applications.

## 6 RELATED WORK AND CONCLUSIONS

There are several ongoing research projects in iterative compilation. In [22] iterative high level optimizations are applied to several embedded processors using two probabilistic algorithms. Good speedups are obtained at the expense of very large number of evaluations. [23] combines static models and empirical search to reduce the search space of optimizations. This method is more restrictive than ours as it considers only tiling and unrolling, excluding array padding, and its model is not based on dynamic characteristics of the programs.

There are also many researches on exploring search heuristics in iterative compilation; 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 [24]; however, [25] finds that simple techniques, when allowed running over multiple iterations, can often outperform complex techniques such as GA. [26] also shows that in finding transformation parameters, random search performs as well as other sophisticated techniques such as GA and simulated annealing. Apan Qasem et al. [27] find that direct search can be an effective technique for finding good values for transformation parameters in a reasonable time. Haihang You et al. [28] apply simplex method to replace the search heuristics of ATLAS (Automatically Tuned Linear Algebra Software) [29], and find that simplex search scheme can produce parameters with better performance.

This paper describes a general and robust framework for composing loop transformations for program optimization. We demonstrate the effectiveness of this framework for well-known computational kernels that require complex transformations to achieve high performance. As we are developing a framework that supports composition of transformations, the research most closely related to ours is Petit [31], WRaP-IT [9], Pluto [32], CHiLL [30] and LeTSeE. These frameworks all use a polyhedral representation. The main difference of our framework with them is that our work considers a much broader range of loop transformations, and allows exploring the best transformation sequence and best parameter values; what is more, in our framework a hardware performance counters based model is presented to reduce the set of transformation. By applying this framework to well-known computational kernels that require complex transformations to achieve high performance, we demonstrate that the resulting code quality is quite high than traditional iterative compilation and polyhedral model. These results show that, with a systematic framework, it has now become feasible for compiler-generated code to achieve performance comparable to manually-tuned ones, even for more complex code constructs than have been previously demonstrated. Experimental results show that the transformation model can effectively narrow down the parameter space; our PIT framework improves performance most, 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, to better guide the application of program transformations, and use training data sets during the tuning process to cut down the program execution time.

## Acknowledgments

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61103014, the National Grand Fundamental Research 973 Program of China under Grant No. G2009CB723803 and the National High Technology Development 863 Program of China under Grant No. 2008AA01A202.

## REFERENCES

[1] Fursin, G.: Iterative Compilation and Performance Prediction for Numerical Applications. Ph. D. Thesis, School of Informatics, The University of Edinburgh, Boston, Massachusetts, June 2005, pp. 67–82.

[2] Kisuki, T.—Knijnenburg, P. M. W.—O'Boyle, M. F. P.: Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. SuperComputing, Vol. 24, 2003, No. 1, pp. 43–67.

[3] Kisuki, T.—Knijnenburg, P. M. W.–O'Boyle, M. F. P.—WIJSHOFF, H. A. G.: Iterative Compilation in Program Optimization. Proceedings of Workshops on Compilers for Parallel Computers, Aussois, France, January 2000, pp. 35–44.

[4] Bodin, F.: Compilers in the Manycore Era. Proceedings of HiPEAK 2009, Paphos, Cyprus, LNCS 5409, Springer 2009, pp. 2–3.

[5] Tiwari, A.—Chen, C. et al.: Scalable Autotuning Framework for Compiler Optimization. Proceedings of the IEEE IPDPS '09, Rome, Italy, May 2009.

[6] Feautrier, P.: The Polytope ModelPast, Present, Future. Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, October 8–10, 2009, LNCS 5234, Springer-Verlag 2009, pp. 4–5.

[7] Pouchet, L.-N.—Bastoul, C.—Cohen, A.—Vasilache, N.: Iterative Optimization in the Polyhedral Model: Part I – One-Dimensional Time. Proceedings of ACM Conf. on Code Generation and Optimization, San Jose, California 2007, pp. 144–156.

[8] Pouchet, L.-N.—Bastoul, C.—Cohen, A.—Vasilache, N.: Iterative Optimization in the Polyhedral Model: Part II – Multidimensional Time. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, Arizona 2008, pp. 90–100.

[9] Girbal, S.–Vasilache, N.—Bastoul, C.—Cohen, A.—Parello, D.—Sigler, M.—Temam, O.: Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. Int. J. of Parallel Programming, Vol. 34, 2006, No. 3, pp. 261–317.

[10] Touati, S.—Barthou, D.: : On the Decidability of Phase Ordering Problem in Optimizing Compilation. Proceedings of the International Conference on Computing Frontiers, Ischia, Italy, May 2006, pp. 147–156.

[11] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: Part II – Multidimensional Time. Int. J. of Parallel Programming, Vol. 21, 1992, No. 5, pp. 315–348.

[12] Lu, P.—Che, Y.—WANG, Z.: A Framework for Effective Memory Optimization of High Performance Computing Applications. Proceedings of 11th IEEE International Conference on High Performance Computing and Communications, Seoul, Korea, June 2009, pp. 95–102.

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

[14] Browne, S.—Dongarra, J. et al.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications, Vol. 14, 2000, No. 3, pp. 189–204.

[15] Carr, S.: Combining Optimization for Cache and Instruction – Level Parallelism. Proceedings of 1996 Conference on Parallel Architectures and Compilation Techniques, Boston, Massachusetts, October 20–23, pp. 238–247.

[16] Zeyao, M.: Realistic Performance Analysis Methods for Parallel Codes. Journal of Numerical Computing and Computer Applications, Vol. 21, 2000, No. 4, pp. 266–275.

[17] Li, Z.—Song, Y.: Automatic Tiling of Iterative Stencil Loops. ACM Transactions on Programming Language and Systems, Vol. 26, 2004, No. 6, pp. 975–1028.

[18] Nelder, J. A.—MEAD, R.: A Simplex Method for Function Minimization. The Computer Journal, Vol. 7, 1965, No. 4, pp. 308–313.

[19] Spendley, W.—Hext, G. R.—Himsworth, F. R.: Sequential Application of Simplex Designs in Optimization and Evolutionary Operation. Technometrics, 1962, No. 4, pp. 441–461.

[20] Wiegand, T.—Sullivan, G.—Luthra, A.: Itu-t rec. h.264-iso/iec 14496-10 avc. Technical report, Joint Video Team of ISO/IEC MPEG and ITU-T VCEG, May 2003.

[21] Dongarra, J.: Performance Optimization for Cluster Computing. Proceedings of the Myrinet Users Group Conference, Vienna, Austria, 2002, available at `http://www.netlib.org/utk/people/JackDongarra/SLIDES/mug-0502.pdf`.

[22] Franke, B.—O'Boyle, M. et al.: Probabilistic Source-Level Optimization of Embedded Programs. Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05), Chicago, Illinois, USA, June 15–17, 2005, pp. 78–86.

[23] Chen, C.—Chame, J.—Hall, M. W.: Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy. International Symposium on Code Generation and Optimization, San José, CA, March 2005, pp. 111–122.

[24] Knijnenburg, P.—Kisuki, T.—O'Boyle, M.: Iterative Compilation. Embedded Processor Design Challenges System Architecture, Modeling and Simulation (SAMOS), Samos, Greece 2002, LNCS 2268, Springer Verlag, pp. 171–187.

[25] Knijnenburg, P. M. W. et al.: The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. Concurrency and Computation: Practice and Experience, Vol. 16, 2004, No. 2-3, pp. 247–270.

[26] Yotov, K.—Pingali, K.—Stodghill, P.: Think Globally, Search Locally. Proceedings of the 19$^{\text{th}}$ Annual International Conference on Supercomputing (ICS 2005), Cambridge, Massachusetts, USA, ACM Press, June 20–22, 2005, pp. 141–150.

[27] Qasem, A.—Kennedy, K.—Mellor-Crummey, J.: Automatic Tuning of Whole Applications Using Direct Search and a Performance-Based Transformation System. The Journal of Supercomputing, Vol. 36, 2006, No. 2, pp. 183–196.

[28] You, H.—Seymour, K.—Dongarra, J.: An Effective Empirical Search Method for Automatic Software Tuning. UTK CS Technical Report, ICL-UT-05-02, Computer Science Department, University of Tennessee, May 2005, pp. 1–8.

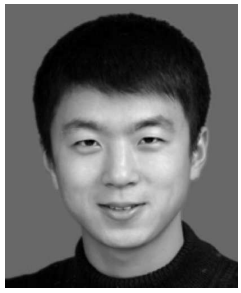
[29] Whaley, R. C.—Petitet, a.—Dongarra, J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing, Vol. 27, 2001, No. 1-2, pp. 3–35.

[30] Chen, C.—Chame, J.—HALL, M.: CHiLL: A Framework for Composing High-Level Loop Transformations. Technical report, University of Southern California 2008.

[31] Kelly, W.—Pugh, W.: A Framework for Unifying Reordering Transformations. Technical report, College Park, MD, USA, CS-TR-2995, 1993, pp. 1–23.

[32] Bondhugula, U.—Hartono, A.—Ramanujam, J.—Sadayappan, P.: A Practical Automatic Polyhedral Program Optimization System. Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, ACM, June 2008, pp. 101–113.

**Pingjing Lu** received her B. Sc. degree, M. Sc. degree and Ph. D. degree in Computer Science at the National University of Defense Technology in China in 2004, 2006 and 2010, respectively. Since 2010 she has been an Assistant Professor of computer science at the National University of Defense Technology. Her current research interests include iterative compilation and adaptive optimization.



**Bao Li** received his B. Sc. degree and M.S degree in Computer Science at the Ocean University of China and National University of Defense Technology in China in 2004 and 2006, respectively. Since 2011 he has been an Assistant Professor of computer science at the National University of Defense Technology. His current research interests include computer graphics and visualization in scientific Ccmputing.

**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.