A New Compiler Framework based on Superword Level Parallel

Zhanjie Guoa,* and Hui Liub

*Department of Electrical and Electronic Engineering, Zhengzhou Technical College, Zhengzhou, 450121, China
bCollege of Computer and Information Engineering, Henan Normal University, Xinxiang, 453007, China

Abstract

Superword level parallel (SLP) algorithm is an automatic vectorization method that is suitable for the applications including parallel codes. Existing SLP algorithm could not efficiently deal with the applications that contain few parallel codes. In the present study, a new compile framework based on the improved SLP algorithm is presented. The framework contains three phases: isomorphic processing for isomeric statements, establishment of superword statements, and data layout optimization. Firstly, isomeric statements with similar instruction in the codes were transformed to isomorphic statements by the improved SLP algorithm. Secondly, the superwords reuse patterns were obtained before making the optimization decisions from a global point of view. Finally, data layout optimization was combined for further performance improvement. The experimental results indicated that the optimization of the compile framework was better than existing SLP algorithm.

Keywords: SLP; compiler framework; superword parallel; isomorphic; superword reuse; data layout

(Submitted on July 21, 2018; Revised on August 23, 2018; Accepted on September 28, 2018)

© 2018 Totem Publisher, Inc. All rights reserved.

1. Introduction

In most cases, multimedia application calculation is compute-intensive calculation suitable for parallel processing, which requires advanced and efficient parallel processing technology. In order to meet the application requirements, more and more processors have integrated SIMD extension unit. The previous multi-media extensions can support data types with comparatively short length only, while the latest extension at present can support 256 bits; AVX2 of INTEL can even support 512-bit superword operation. Therefore, a new data parallelism type called superword level parallel (SLP) can process relatively wide data types is presented [1]. SLP means to pack data into big superwords, and conduct computing operation via SIMD vector instruction. Different from vector parallelism, which is suitable for the application containing a large number of parallel codes, SLP is more suitable for the application including moderate parallel codes.

The following jobs are examined in this paper. Firstly, SLP algorithm is improved in some aspects. Then, an efficient compiler framework covering three stages: isomorphic processing for isomeric statements, establishment of superword statements, and data layout optimization is presented. Isomorphic processing for isomeric statements mainly includes the process of seeking vectorization opportunities missed by SLP algorithm and establishing the SLP supplementary graph [2]. In other words, vectorization opportunities that might be missed by SLP algorithm in code optimization are searched. Then, dependency graphs that are similar but not the same are made isomorphic by adding some redundant nodes. This establishes the SLP supplementary graph supporting vectorization processing of SLP. Establishment of superword statements can be divided into statement grouping and statement scheduling. Statement grouping means to group SIMD operation on the premise of enhancing SIMD parallelism and acquiring superword reuse between groups as far as possible; statement scheduling means to sort various groups and statements in each group on the premise of reducing sorting cost of vector register as far as possible. At the stage of data layout optimization, the data access pattern of input data after experiencing the first two stages is analyzed. Data in the memory are reorganized according to the data access pattern to minimize the memory operation cost of SLP. Here, a new compiler framework is operated, and relevant experiment is conducted. The experimental results demonstrate that the new framework is superior to SLP algorithm in performance.
2. SLP and Relevant Researches

2.1. SLP

SIMD compiler optimization mainly aims to effectively convert various high-level languages into efficient SIMD instructions corresponding to multi-media extensions [3]. The automatic parallel traditional compilation technology of vector machine relies on complicated loop transformation and can act on application programs containing a large number of parallel codes only while having little effect on those applications not supporting parallel processing [3-9].

In order to solve the above problems, Larsen and Amarasinghe et al. [5] proposed a new parallel mode called SLP, which aims to solve the problems of multi-media extension applications. SLP algorithm mainly includes 5 steps: (1) Seek instruction codes that can be used as vectorization seeds; (2) Group the instructions supporting vectorization according to the data dependency graph of seed instructions; (3) Assess the code performance of scalar and vector; (4) Compare the costs of two forms; (5) Replace the scalar code with equivalent vector code if vectorization can produce returns. An important advantage of SLP is that it can efficiently use multi-media extension instructions even if the proportion of parallel codes is moderate.

2.2. Relevant Researches

Shin et al. [10-11] developed a strategy to manage files in the vector register, turning it into a cache of compiler control and enabling it to increase the management efficiency for local data when utilizing SLP. Moreover, they et al. [12-13] derive a large basic block when using the instruction assertion of the current control flow; it can be used to identify more superword level parallels.

Tenllado et al. [14-15] provided a technique, which can efficiently utilize SLP when loop carrying dependency exists in inner loop. Nuzman et al. [16] studied a compilation technique supporting efficient vectorization of inserted data. Nuzman and Zaks et al. [17] made a research on the surrounding loop vectorization technology of short SIMD framework. Barik et al. [18] put forward that automatic vectorization could be realized on the bottom layer IR approaching the machine layer to acquire higher compiling efficiency during dynamic compilation. We have proposed a more comprehensive and different strategy of extracting SLP on the basis of original SLP algorithm.

3. Overview of Overall Framework

Firstly, a basic block set is input; then, this basic block set is treated through the following optimization steps:

- **Step 1** Conduct isomorphic processing for dependency graphs that are similar but isomeric
- **Step 2** Enhance the parallelism by adding more superword statements
- **Step 3** Reduce the quantity of superword packing/unpacking through getting more superword reuses
- **Step 4** Reduce the compulsory packing/unpacking cost and cost of rearrangement instruction in the memory through data layout optimization

Figure 1 presents the overview of our compiler framework, which mainly has 4 modules: preprocessing, isomorphic processing, global SLP optimization [1], and back-end processing. After the source code of program is input, preprocessing module will conduct loop unrolling and data alignment analysis for the input, exposing more opportunities of SLP utilization. Isomorphic processing is conducted for isomorphic and similar dependency graphs, and global SLP optimization is utilized to optimize the program and generate vectorization codes. Back-end processing module can realize register allocation as well as other underlying optimization and output.

4. Isomorphic Processing

4.1. Seek the Vectorization Opportunity Missed by SLP

SLP algorithm can process several isomorphic instructions at the same time, but it might still miss many optimization possibilities. Before that, the program has already experienced many optimization steps that might remove the vectorization opportunity of the current stage. For instance, early redundant node deletion might change isomorphic dependency graphs...
that can be processed by SLP at the current stage into isomorphic dependency graphs that cannot be processed by SLP. Hence, the original vectorization opportunity will be missed. Therefore, the dependency graph should be formed with statements first and then an attempt can be made to establish the superword group to find out vectorization opportunities missed by SLP algorithm as far as possible.

Figure 1. Overview of the new framework

4.2. Establish the SLP Supplementary Graph

When the vectorization opportunity missed by SLP algorithm is found out, redundant nodes can be added to make isomorphic statements isomorphic and establish the SLP supplementary graph. When the SLP supplementary graph is established, the following problems should be solved.

4.2.1. Establish the Maximum Common Sub-Graph

Seeking the maximum common sub-graph of two graphs is considered as an isomorphic issue of NP. In practical situations, all graphs are small. Therefore, a fast backtracking algorithm approximate to the optimal algorithm is proposed here. The algorithm idea is as follows:

Firstly, two graphs $g_1$ and $g_2$ are input. Then, the graph nodes are classified and traversed from bottom to up. Finally, main recursive function of the algorithm is called. Starting from one node in graph $g_1$, the matching node is sought in the other graph $g_2$. In the searching process, the following nodes will be skipped:

- Nodes that have been matched
- Unmatched types
- Nodes that might produce loops
- Nodes in different basic blocks
- Nodes that do not support parallel scheduling

Once two matching nodes are found out, they should be set as the candidates and inserted into the mapping of $g_1$ and $g_2$ for reversal. Then, other possible matches should be found out within the given scope and a new round of searching is started. In order to reduce the complexity, many nodes whose whole mobility is higher than the critical value are ignored, and only matching pairs of nodes that might produce returns are considered.

When the algorithm jumps out of the loop, node mapping will be ended and the algorithm will establish two side mappings (common sub-graph sides in $g_1$ and $g_2$ will be mapped). Based on side and node mapping, nodes and sides that do not exist in the mapping graph are filtered to generate the maximum common sub-graph. Then, the best sub-graphs are compared and the maximum one among them is found.
4.2.2. Establish the Minimum Common Super-Graph

The maximum common sub-graph is used to calculate the minimum common super-graph. The minimum common super-graph is gained by combining the maximum common sub-graph with its differences from the original graph. For example, if \( g_1 \) and \( g_2 \) are original graphs, and \( \text{MaxCS} \) is the maximum common sub-graph of \( g_1 \) and \( g_2 \), then \( \text{MinCS}_1 = \text{MaxCS} + \text{diff}_1 + \text{diff}_2 \), in which \( \text{diff}_1 = g_1 - \text{MaxCS} \) and \( \text{diff}_2 = g_2 - \text{MaxCS} \).

4.2.3. Select Insertion Nodes

SLP supplementary graph will launch selection instruction and select side parameters to flow in before the side parameters enter the nodes. In order to make semantics of the original graph remain unchanged, \( \text{diff} \) belonging to the original nodes should be selected.

4.2.4. Remove Redundancy

The last stage of supplementary graph is to delete those selection instructions that can be optimized. This process can optimize four selection instructions; the optimization result is to improve the performance by deleting redundant selection instructions in codes.

- Replace the constant with the selected constant
- Select sides with the same processor node
- Select instructions of reading constant recognition operation and instruction nodes applicable to selection instructions and the current instruction.
- Degrade selection instructions

4.2.5. Establish the Minimum Super-Graph of Multiple Graphs

The construction of the SLP supplementary graph is to calculate the minimum super-graph of multiple graphs in a fast but not optimal way. We need to calculate the minimum super-graph of a group of graphs and use it as the left graph of the next set of the graphs. Repeat the above steps to get the supplementary graph of the multiple graphs.

5. Establishment of Superword Statements

Establishment of superword statements mainly includes statement grouping and statement scheduling.

5.1. Grouping

An iterative process is used to realize statement grouping. Firstly, a group with the size of 2 is found out via a block grouping algorithm, and then the existing groups are used as atomic statements. Groups with bigger blocks are further gained via block grouping algorithm until the width of SIMD data is completely utilized. The grouping stage is mainly divided into basic grouping algorithm and iterative grouping.

5.1.1. Basic Grouping Algorithm

Figure 2 shows an example of obtaining the basic block with the size of 2 via basic grouping algorithm.

The first step of the algorithm is to identify candidate groups. A candidate group refers to the SIMD instruction group containing two isomorphic statements \( \{S_i, S_j\}, S_i, S_j \in S \). The superword size of candidate group should not exceed the target SIMD data width, and there is no sequence among isomorphic statements of candidate group. For instance, the candidate group set of codes in Figure 2 is \( C = \{\{S_1, S_2\}, \{S_1, S_3\}, \{S_4, S_5\}\} \).

Based on the candidate group set, the second step is to establish the variable packet conflict graph. A variable packet refers to a group of variable sets that different statements in one candidate group have the same position. As shown in Figure 2, variable sets \( \{V_1, V_2\} \) and \( \{V_3, V_5\} \) from \( \{S_1, S_3\} \) are candidate sets of superword. The establishment process of variable packet conflict graph is as follows: traverse candidate groups in \( C \) successively, and establish a node set of all variable packets generated by statements for every candidate group \( \{S_1, S_2\} \), and insert sides between the newly established
node and original node. Figure 3 presents the variable packet conflict graph of codes produced by the candidate group set in Figure 2.

\[
\begin{align*}
S_1: & \quad V_1 = V_3; \\
S_2: & \quad V_2 = V_5; \\
S_3: & \quad V_3 = V_7; \\
S_4: & \quad V_4 = V_1 + V_1; \\
S_5: & \quad V_5 = V_2 + V_5;
\end{align*}
\]

Figure 2. A basic block

The third step is to establish the statement grouping graph \(G = (V', T')\) via candidate groups identified at the early stage and variable packet conflict graph gained previously. Every node \(e \in V'\) in the graph represents a statement \(S_p\) in the basic block, and every side \((e, T')\) means the candidate group between two statements. Statement grouping graph is a weighted graph, and the weight of every side represents its possible return. In order to calculate the weight of side between two statements \(S_p\) and \(S_q\) in the candidate group \(\{S_p, S_q\}\), all packets (nodes) in the variable packet graph \(VP\) are extracted to establish an auxiliary chart. When the weight of side between statements \(S_4\) and \(S_5\) in Figure 4 is calculated, the auxiliary chart is presented in Figure 5. It covers all nodes in Figure 3, which are the same as but do not collide with variable packets in \(\{S_p, S_q\}\). By combining the remaining nodes \(\{V_1, V_2\}-\{S_1, S_2\}\) and \(\{V_1, V_5\}-\{S_1, S_2\}\) in the auxiliary chart as well as variable packets \(\{V_2, V_3\}-\{S_3, S_5\}\), \(\{V_1, V_2\}-\{S_4, S_5\}\) and \(\{V_1, V_5\}-\{S_4, S_5\}\) of candidate group in \(VP\) graph, we can gain the average reuse value of variable packets (superwords) from the candidate group \(\{S_p, S_q\}\). For example, the value of candidate group \(\{S_4, S_5\}\) is \(2/3\). The reuse value is the weight of side between statements \(S_p\) and \(S_q\) in graph \(SG\). Therefore, the potential return (superword reuse) of statements in the basic block (global influence) is obtained when the ultimate code groups them into superwords.

The fourth step is to set up grouping decision on the basis of grouping graph established before.

The side with the largest weight is selected from all sides in the decreasing weight screening graph. If two sides have the same weight, one can be selected randomly. Once the decision is made, \(SG\) graph and \(VP\) graph should be updated, and statement nodes just found by SIMD group in \(SG\) graph and all associated nodes (such as conflict statements) should be deleted, as shown in Figure 6. Nodes of variable packets generated by SIMD group of the latest decision and all associated nodes (such as conflict variable packets) are deleted, as shown in Figure 7. Then, the weights of all sides in \(SG\) graph should be recalculated. This process should be repeated until no side is left in \(SG\) graph.

Figure 8 is the pseudo-code of basic grouping algorithm. The first line of the algorithm will identify the candidate statement group. Lines 2-10 will establish the variable packet conflict graph, and lines 11-17 will initialize the statement grouping graph. The key part of the algorithm lies in lines 20-41, which will make grouping decisions. Lines 21-29 will establish the auxiliary chart of weight calculation according to the current candidate group and existing groups. Conflicts in
the auxiliary chart will be solved in line 30, and lines 31-37 are used to calculate the weight (average superword reuse). Lines 39-40 will choose the side with the maximum weight as the current grouping decision, and then change the variable packet conflict graph and statement grouping graph. Lines 20-41 will repeat the above work until all groups are determined. The complexity degree of basic grouping algorithm is $O(E_{SG}^2 \times Nvp)$, in which $E_{SG}$ is the number of sides in the statement grouping graph, and $Nvp$ means the number of nodes in the variable packet conflict graph.

```
Input: Basic block statement sequence $S = <S_1, S_2, S_3, ..., S_n>$
Output: Group of statements $D'$
1: Identify candidate groups $g$;
2: Initialization Vp://VP: establish the variable packet conflict graph
3: for $\{S_i, S_j\} \in g$ do
4: for $(V_i, V_j), V_i \in S_{i,v}$ and $V_j \in S_{j,v}$ do
5: $Vp.v ← Vp.v \cup \{(V_i, V_j) \in SG\}$;
6: for $(V_i, V_j), (S_{i,S}) \cup (S_{j,S}) \neq 0$ do
7: $Vp.v ← Vp.v \cup \{(V_i, V_j) \in SG\}$;
8: end for
9: end for
10: end for
11: Initialization SG://SG: statement grouping graph
12: for $\{S_i, S_j\} \in g$ do
13: for $S \in \{S_i, S_j\}$ and $S \notin SG.v$ do
14: $SG.v ← SG.v \cup [S]$;
15: end for
16: $SG.e ← SG.e \cup \{(S_i, S_j)\}$;
17: end for
18: $D' ← 0$;
19: while $SG.e \neq 0$ do
20: for $\{S_i, S_j\} \in SG.e$ do
21: Initialization AG://AG: Auxiliary chart
22: for $(V_i, V_j) \in SG.v$ do
23: if $(V_i, V_j) \in VP.v \land (S_{i,S}) \neq (S_{j,S})$ do
24: $AG.v ← AG.v \cup \{(V_i, V_j)\}$;
25: end if
26: end for
27: for $P_n, P_t \in AG.v \land \{P_n, P_t\} \in VP.e$ do
28: $AG.e ← AG.e \cup \{P_n, P_t\}$;
29: end for
30: Resolve the conflict in AG1;
31: //Calculate the reuse superword of $\{S_i, S_j\}$
32: $r ← 0$;
33: for $(V_i, V_j) \in AG.v$ or $D' \cup \{(S_i, S_j)\}$ do
34: $r ← r + (N_{V_i, V_j} \cdot 1)$;
35: end for
36: $W_{\{(S_i, S_j)\}} ← r$;
37: //The number of packet in $N_{D'} \cup \{(S_i, S_j)\}$
38: $SG.w ← SG.w \cup W_{\{(S_i, S_j)\}}$;
39: end for
40: $D' ← D' \cup \{(S_i, S_j)\} / W_{\{(S_i, S_j)\}} = MAX(SG.w)$;
41: Update VP and SG;
42: $SG.w ← 0$;
end while
```

Figure 8. Pseudo-code at basic block grouping stage

5.1.2. Iterative Grouping

The size of SIMD (superword statement) generated by basic grouping algorithm is 2. In order to solve the problem that it cannot fully utilize SIMD data width under the specific framework, an iterative process is used to acquire a larger SIMD group to expand the basic grouping algorithm. Its essential concept is as follows. After the first pass, every SIMD group $\{S_{i,p}, S_{i,q}\}$ is set as a separate statement, and every variable packet is used as a new single variable. New statements are added, and the original statement set in the basic block is updated. The statement set after updating is used to apply basic grouping algorithm to the basic block. This strategy should be used iteratively until the ultimate superword statement can use SIMD
data width to the largest extent. Hence, even if the width of SIMD extension is increased in the future, our framework can still utilize it efficiently.

5.2. Statement Scheduling

The statement is grouped into superword statement via grouping, and the use of expensive memory access operation is reduced by acquiring more superword reuses. If the same data have different sequences in two superwords, an extra vector register should be used to reorder the instructions. At the scheduling stage, two problems should be solved. (1) An effective execution sequence should be arranged for all statements (including single-word statement and superword statement) in the basic block, and superword reuses in the superword statement should be approximate as far as possible. (2) The number of statements whose sequence is rearranged in superword statements should be reduced as far as possible.

Based on the execution sequence of original statements in the basic block, we establish a dependency graph between the superword statements that are produced. In this graph, every node represents one superword statement, and every direction side indicates the dependency between two associated superword statements. The dependency graph can provide many different scheduling modes so its flexibility can be fully utilized to gain more returns.

Based on this dependency graph, superword statements can be scheduled, and their execution sequence determined. A current superword set is defined as the superword set approaching the current vector register. In the current superword set, the sequence of statements in each superword is fixed. In the beginning, the current superword set is empty, and superword statements whose dependency relationship has been solved are extracted from the dependency graph as candidate statements of the target execution sequence. Then, the number of superword reuses every candidate superword statement and the number of superword reuses in the current superword set is calculated. The candidate statement with the maximum number of superword reuses will be set as statement to be executed next; superword reuses of close distance and high probability in the vector register are searched to decide the execution sequence of superword statements. By inserting new orderly superwords and accessing existing superwords of the same data, we can update the current superword set. Then, new existing superwords are extracted to repeat the above steps until all nodes in the dependency graph are processed.

The sorting decision of statements in every superword statement is separated from the grouping stage and postponed to the scheduling stage. In this way, our framework can take full advantage of direct and indirect superword reuses. Indirect superword reuse is the key, as it can avoid memory access operation by introducing the register to rearrange the instructions. After basic block grouping and scheduling are realized, a cost mode similar to that in literature [3] is applied to evaluate the code acceleration after conversion by considering all elements. If such conversion will reduce the program performance, it should be discarded.

6. Data Layout Optimization

The basic concept of data layout optimization is to organize superwords in the memory with a specific mode to minimize the cost when the vector register loads them (or they are written back into the memory from vector register). As shown in Figure 9, if the width of SIMD can load two basic data types, we form a group that can produce two superwords with statements $S_1$ and $S_2$ after the first stage: $(a, b)$ and $(A[4i], A[4i + 3])$. Suppose that the two superwords are not in the vector register during current execution, they need to be put into the vector register / taken out from the vector register through packing /unpacking before/ after SIMD operation is applied. At this time, if variables $a$ and $b$ are adjacent and aligned in the memory, only one memory operation is needed to write back $(a, b)$. In this paper, the data layout of two data types is optimized: scalar superword and array reference superword. Scalar superword is processed with address alignment, and array reference superword is processed with data conversion and copying.

\[
\begin{align*}
S_1: & \quad a = A[4i]; \\
S_2: & \quad b = A[4i+3];
\end{align*}
\]

Figure 9. Example of data layout optimization

7. Experimental Evaluation

Figure 10 gives the comparison between the new compiler framework and original SLP algorithm in the optimization process for a basic block. Suppose that one superword can load two variables, and Figure 10(a) is the source code. The ultimate superword statement is \{(S_1, S_2), (S_2, S_3), (S_3, S_4), (S_7, S_8)\} as shown in Figure 10(b). After isomorphic processing for isomeric statements and superword establishment of the framework, the superword shown in Figure 10(c) can be gained,
i.e. \( \{S_1, S_4, S_5, S_3, S_2, S_6, S_7, S_8\} \). The code generated is presented in Figure 10(d), in which arrays C and D are established via data reproduction and data renaming.

\[
\begin{align*}
&\text{S_1: } a = A[i]; \\
&\text{S_2: } c = a \times B[4i]; \\
&\text{S_3: } g = q \times B[4i-2]; \\
&\text{S_4: } b = A[i+1]; \\
&\text{S_5: } d = b \times B[4i+4]; \\
&\text{S_6: } h = r \times B[4i+2]; \\
&\text{S_7: } A[2i]=d+a \times c; \\
&\text{S_8: } A[2i+2]=g+r \times h;
\end{align*}
\]

(a)

\[
\begin{align*}
&\text{S_1: } a = A[i]; \\
&\text{S_2: } b = A[i+1]; \\
&\text{S_3: } d = b \times B[4i+4]; \\
&\text{S_4: } g = q \times B[4i-2]; \\
&\text{S_5: } c = a \times B[2i]; \\
&\text{S_6: } h = r \times B[2i+1]; \\
&\text{S_7: } A[2i]=d+a \times c; \\
&\text{S_8: } A[2i+2]=g+r \times h;
\end{align*}
\]

(b)

\[
\begin{align*}
&\text{S_1: } a = A[i]; \\
&\text{S_2: } b = A[i+1]; \\
&\text{S_3: } d = b \times B[4i+4]; \\
&\text{S_4: } g = q \times B[4i-2]; \\
&\text{S_5: } c = a \times B[2i]; \\
&\text{S_6: } h = r \times B[2i+1]; \\
&\text{S_7: } A[2i]=d+a \times c; \\
&\text{S_8: } A[2i+2]=g+r \times h;
\end{align*}
\]

(c)

\[
\begin{align*}
&\text{S_1: } a = A[i]; \\
&\text{S_2: } b = A[i+1]; \\
&\text{S_3: } d = b \times B[4i+4]; \\
&\text{S_4: } g = q \times B[4i-2]; \\
&\text{S_5: } c = a \times B[2i]; \\
&\text{S_6: } h = r \times B[2i+1]; \\
&\text{S_7: } A[2i]=d+a \times c; \\
&\text{S_8: } A[2i+2]=g+r \times h;
\end{align*}
\]

(d)

Figure 10. Example of overall optimization

7.1. Experimental Environment

Sixteen standard test sets are used to test the new compiler framework in two systems. Besides, this framework is executed in SW-VEC compiler structure. A comparison is made with the algorithm proposed in literature [5]. In the two times of execution, in order to guarantee more superword level parallels in compilation, optimization like data alignment analysis and loop unrolling is conducted in preprocessing. In the two times of execution, mapping is realized between register shuffling/rearrangement operation and instruction set loading/storing operation supported by underlying framework rather than loading/storing operation supported by physical memory. Table 1 shows the basic configuration of the two systems.

<table>
<thead>
<tr>
<th>Configuration of experimental environment</th>
<th>Intel</th>
<th>AMD Phenom ll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of kernels</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

In this study, ten floating point test sets in SPEC2006 and six NPB test sets are adopted. These test sets can display the parallel effect of our tool on multiple types of application programs. For each test set, the largest input block within the permissible range is used. The following report shows the comparison among the optimization results of different optimization modes.

7.2. Experimental Result

The first result set displays the comparison among global optimization, SLP optimization and local execution results of every test set in Intel system. In Figure 11, the test set on x-axis presents the results from the poorest performance to the optimal performance according to global optimization. Our global optimization and SLP optimization produce the same results for 3 applications in test set. In terms of other applications, the performance of global optimization is superior to that of SLP optimization. Moreover, when there are more effective scheduling candidates and more potential superword reuses in the basic block, our global optimization is much better than SLP algorithm optimization. According to the figure, SLP optimization group and local execution group show the same optimization results for 4 applications.
A New Compiler Framework based on Superword Level Parallel

In order to explain the differences between our global optimization and SLP optimization, Figure 12 displays relevant dynamic instruction cost and packing/unpacking cost when global optimization and SLP optimization are executed. According to the figure, our method can reduce the proportions of dynamic instruction and packing/unpacking operation by 14.6% and 43.7% respectively. It can be observed that our global optimization is effective for most tests.

Figure 12. Dynamic instruction cost and packing / unpacking cost saved by global optimization when compared with SLP optimization

Figure 13 shows the comparison of overall optimization and scalar execution results. It can be seen that data layout optimization presents extra performance improvement for 7 applications (ua, ft, soplex, bt, milc, gromacs, and dealII) in the test set. Under the specific constraints mentioned above, the remaining test sets do not show a good effect. Compared with the results of SLP algorithm in Figure 13, the overall optimization performance is about 15.3% better than the performance of SLP algorithm.

Figure 13. Comparison of global optimization and overall optimization performance in Intel

Figure 14 presents the execution effect of global optimization and overall optimization in AMD system. According to the figure, global optimization and overall optimization have improved the performance by an average of 10.8% and 14.1% respectively, similar to the results in Intel (global optimization 12.2% and overall optimization 14.9%). We consider that the cost of packing/unpacking operation causes the effect in AMD is lower than the effect in Intel.
8. Conclusion

The compiler framework that realizes automatic detection and utilizes superword level parallel in application programs is the major contribution of this paper. There are three strategies: isomorphic processing for isomeric statements, establishment of superword statements, and data layout optimization. Packing/unpacking operation is reduced by extracting superword reuses to decrease the compulsory packing/unpacking operation cost as well as the use of rearrangement instruction in memory during data reconstitution in memory. In this way, SIMD parallelism can be enhanced. According to the experimental results in two commercial systems, the performance of our new framework is superior to that of SLP algorithm, and a performance improvement of 15.3% is presented.

References


Zhanjie Guo graduated from College of Computer and Information Engineering, Henan Normal University, Xinxiang, China, for the degree of Bachelor and Master. Now she is a lecturer of Department of Electrical and Electronic Engineering, Zhengzhou Technical College, Zhengzhou, China. Her current research interests include high performance computing, program performance analysis and optimization.

Hui Liu graduated from College of Computer and Information Engineering, Henan Normal University, Xinxiang, China, for the degree of Bachelor and Master. Now she is a lecturer of College of Computer and Information Engineering, Henan Normal University. Currently she is also a Ph.D. candidate in State Key Laboratory of Mathematical Engineering and Advanced Computing at PLA Information Engineering University, Zhengzhou, China. Her research interests include high performance computing, program performance optimization and compiler optimization.