A Method for Estimating Optimal Unrolling Times for Nested Loops

A. Koseki
School of Science & Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku
Tokyo 169, Japan
koseki@fuka.info.waseda.ac.jp

H. Komastu
Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato-shi
Kanagawa 242, Japan
komatsu@trl.ibm.co.jp

Y. Fukazawa
School of Science & Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku
Tokyo 169, Japan
fukazawa@fuka.info.waseda.ac.jp

Abstract

Loop unrolling is one of the most promising parallelization techniques, because the nature of programs causes most of the processing time to be spent in their loops. Unrolling not only the innermost loop but also outer loops greatly expands the scope for reusing data and parallelizing instructions. Nested-loop unrolling is therefore a very effective way of obtaining a higher degree of parallelism. However, we need a method for measuring the efficiency of loop unrolling that takes account of both the reuse of data and the parallelism between instructions. This paper describes a heuristic algorithm for deciding the number of times and the directions in which loops should be unrolled, through the use of information such as dependence, reuse, and machine resources. Our method is evaluated by applying benchmark tests.

1 Introduction

Optimization of loops can greatly reduce the execution time of a program, because most of the execution time of a program is spent in the loops. Especially in numerical programs, the restrictions on parallel execution due to dependences between instructions in loops are not tight. Since such programs are easily executed in parallel on vector processors or instruction-level processors, a good deal of research has been done on loop optimization [1, 2].

Loop unrolling is a kind of loop optimization technique that unrolls a loop body several times. This method is effective because it removes the **Compare** and **Jump** instructions that create loop structures, and reduces pipeline disruptions. Especially, important is the fact that it removes control dependences and allows instructions to move beyond loop boundaries. In iterations of a loop, the same member of an array is often referred to repeatedly, or the same value is calculated repeatedly. If the loop is unrolled, the reference and the calculation can be done just once, and subsequent repetitions can be replaced by reference to a register containing the data. This can reduce the number of instructions and the execution time of the program.

Loop unrolling is so effective that several methods have been proposed, some of which work not only for the innermost loop but also for nested loops. We think that methods for unrolling nested loops are very important, because several loops can be unrolled simultaneously in fully permutable nested loops [3], and unrolling of nested loops greatly increases the scope for reusing data and parallelizing instructions. In previous work on unrolling nested loops [4], "balance" was considered to consist in maintaining the same number of floating-point operations and the same amount of memory traffic. However, no use was made of the information on the dependences between instructions and on the reuse of data, which strongly affects the efficiency obtained by loop unrolling.

The efficiency obtained by unrolling nested loops is affected by the dependences between instructions and by the reuse of data over the iteration of the loops; therefore, the direction of unrolling (i.e., which loops should be unrolled) and the number of times it is done have an extremely strong effect on the efficiency of the execution of the unrolled loop. It is thus important to obtain information on dependence and reuse among the iterations by analyzing program structures.

In addition, the efficiency obtained by unrolling will saturate, because hardware resources are limited; therefore, at certain times, no further effect can be obtained by unrolling the loop. To obtain the best effect, we have to decide the number of times and the direction in which loops should be unrolled, on the basis of information such as the dependences among iterations, the reuse of data, and the machine resources.

2 Characteristics of the unrolling of nested loops

2.1 Unrolling of nested loops

The characteristics of the unrolling of nested loops are illustrated by means of an example. Figure 1 shows the target program to be unrolled. An example of the unrolling of nested loops is as follows: First, Fig. 2 shows the dependence graph of the target program in Fig. 1. This program consists of double-nested loops,

1087-4089/97 $10.00 © 1997 IEEE

```
for i := 1 to IMAX do
  for j := 1 to JMAX do
    D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

Figure 1: Target program

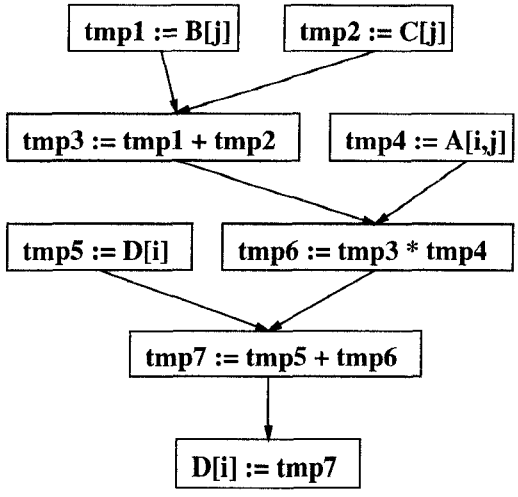

Figure 2: Dependence graph of a loop body

and can be unrolled by index **i** (direction **i**) and index **j** (direction **j**) simultaneously.

Next, in order to unroll the loops twice in direction **i** and three times in direction **j**, the loop indices are divided as shown in Fig. 3. A loop block consisting of double-nested loops that contains six iterations is obtained by exchanging loops **i** and **jj** (Fig. 4). This process can be performed only when loops **i** and **jj** are exchangeable; in short, there is no dependence that prevents the loops from being exchanged.

Finally, the loop block is unrolled to form a new loop body (Fig. 5). The process shown in Figs. 1 to 5 is called nested loop unrolling. In this example, we have a new loop body that contains six iterations of the original loop It was unrolled twice in direction **i** and three times in direction **j**.

2.2 Optimization of an unrolled loop body

After unrolling, instructions in the loop body are parallelized and data are reused.

First, in Fig. 5, there are two instructions that refer to **B[j]**, **B[j + 1]**, and **B[j + 2]**, respectively, and each data item has the same value, so we can use a single LOAD instruction for the first reference to the data, and then move the data to a register. At the second reference, we can use the data in the register without a LOAD instruction. Thus, three LOAD instructions can be removed by data reuse. In the same way, for references to array **C**, three LOAD instructions can be removed.

Removal of instructions by the reuse of data can be adapted not only to LOAD instructions but also to arithmetic calculations. In this example, the calculation **B[j] + C[j]** appears twice, but each calculates the same value. Therefore, the data can be calculated once, and moved to a register. For the second calculation, we can use the data in the register instead of using an ADD instruction. In the same way, for calculations **B[j + 1] + C[j + 1]**, **B[j + 2] + C[j + 2]**, three ADD instructions can be removed.

```
for ii := 1 to IMAX step 2 do
  for i := ii to ii + 1 do
    for jj := 1 to JMAX step 3 do
      for j := jj to jj + 2 do
        D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

Figure 3: Division of loop indices

```
for ii := 1 to IMAX step 2 do
  for jj := 1 to JMAX step 3 do
    for i := ii to ii + 1 do
      for j := jj to jj + 2 do
        D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

Figure 4: Loop blocking

In the reference to array **D**, dependence between instructions referring to the array complicates the task of optimization. The instruction that assigns a value to **D[i]** has a flow-dependence on the instruction that refers to **D[i]** in the next line. Among these instructions referring to **D[i]**, the value of **D[i]** is same, so the LOAD instruction that refers to **D[i]** can be replaced by the data in the source register of the STORE instruction that previously assigned a value to **D[i]**. This optimization is called scalar replacement. Other LOAD instructions can be replaced in the same way.

There are also output-dependences among the instructions that assign values to **D[i]**, and each STORE instruction transfers data to the same address. Therefore, the first and second STORE instructions can be removed. In the same way, the first and second STORE instructions that set **D[i+1]** can be removed.

Finally, there is no dependence among the instructions that refer to arrays **A**, **B**, and **C**. Therefore, LOAD instructions that refer to **A[i,j]**, **A[i,j + 1]**, **A[i,j + 2]**, **A[i + 1,j]**, **B[i + 1,j + 1]**, **B[i + 1,j + 2]**, **B[j]**, **B[j + 1]**, **B[j + 2]**, **C[j]**, **C[j + 1]**, and **C[j + 2]** can be executed in parallel.

In this example, we have been optimizing the loop body. This is an example in which we unrolled the loop twice in direction **i** and three times in direction **j**. We are showing that parallelization and reuse can be parameterized by the direction and the number of times unrolling is performed. For example, LOAD instructions that refer to array **B** can be reused as many times as the loop is unrolled in direction **i**, and parallelized as many times as the loop is unrolled in direction **j**.

We have to analyze how reuse and parallelization are carried out, and then use them to unroll a loop effectively.

```
for i := 1 to IMAX step 2 do
  for j := 1 to JMAX step 3 do
  begin
    D[i] := D[i] + A[i,j] * (B[j] + C[j]);
    D[i] := D[i] + A[i,j+1] * (B[j+1] + C[j+1]);
    D[i] := D[i] + A[i,j+2] * (B[j+2] + C[j+2]);
    D[i+1] := D[i+1] + A[i+1,j] * (B[j] + C[j]);
    D[i+1] := D[i+1] + A[i+1,j+1] * (B[j+1] + C[j+1]);
    D[i+1] := D[i+1] + A[i+1,j+2] * (B[j+2] + C[j+2]);
  end
```

Figure 5: Unrolling nested loops

3 Extraction of the characteristics of a program

In this section, we describe how to parameterize the optimization of a loop body, and how to use information derived from a program.

3.1 Extraction of dependences and reuse

A dependence graph is built by analyzing the global data flow of a loop body. The information described below is obtained by using reuse vectors and loop-carried dependence [5, 6].

3.1.1 Extraction of loop-carried dependence

In the execution of a loop, some instructions have dependences over the iterations of the loop. This is called loop-carried dependence. Loop-carried dependences are analyzed using array references derived from the names of arrays and indices.

In this paper, loop-carried flow-dependences, anti-dependences and output-dependences are denoted as follows:

- **flow-dependence**($\mathbf{n_i}, \mathbf{n_j}, ...$)
- **anti-dependence**($\mathbf{n_i}, \mathbf{n_j}, ...$)
- **output-dependence**($\mathbf{n_i}, \mathbf{n_j}, ...$)

Each parameter shows the numbers of times unrolling is performed for each index. The numbers are ordered from outer loops to inner loops, and from left to right. For example, **flow-dependence**($\mathbf{n_i}, \mathbf{n_j}$) shows that the loops are double-nested, and a flow-dependence appears when the outermost loop is unrolled $\mathbf{n_i}$ times and the innermost loop is unrolled $\mathbf{n_j}$ times.

3.1.2 Extraction of reuse over iterations

In the execution of a loop, some data can be reused in later iterations. This is called reuse over iterations. Reuse over iterations is also analyzed by using array references derived from the names of arrays and indices.

In this paper, reuses over iterations are denoted as follows:

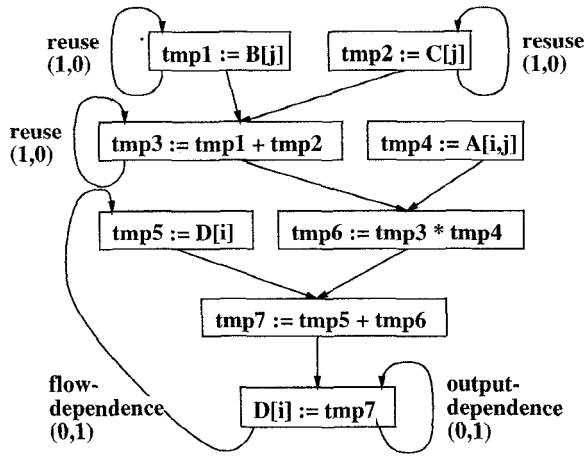

Figure 6: Dependence graph and information derived from the graph

- **reuse**($\mathbf{n_i}, \mathbf{n_j}, ...$)

Each braced part shows the numbers of times unrolling is performed for each index. The numbers are ordered from outer loops to inner loops, and from left to right. For example, **reuse**($\mathbf{n_i}, \mathbf{n_j}$) shows that a reuse of data appears when the outermost loop is unrolled $\mathbf{n_i}$ times and the innermost loop is unrolled $\mathbf{n_j}$ times.

3.1.3 Example

Figure 6 shows the result of analyzing Fig. 2 as described above. In Fig. 6, three reuses appear when the program is unrolled once in direction **i**, and two dependences appear when the program is unrolled once in direction **j**.

3.2 Calculation of parameters for unrolling

We have two parameters for unrolling. One is an increase in the length of the longest path of a dependence graph when a loop is unrolled once; this increase is derived from the information on dependence. The other is a decrease in the number of instructions in a loop body when a loop is unrolled once; this decrease is derived from the information on reuse.

An increase in the length of the longest path of a dependence graph when a loop is unrolled once in a particular direction is obtained through the analysis of dependences that may appear when the loop is unrolled in that direction. In detail, for each dependence, after calculation of the increase in the length of the longest path when the loop is unrolled enough times to make the dependence appear, and calculation of the average increase in each dependence when the loop is unrolled once, the maximum of the average increases is the increase in the length of the longest path when the loop is unrolled once in that direction.

A decrease in the number of instructions in a loop body when a particular loop is unrolled once in a direction is obtained through the analysis of reuses, flow-dependences, and output-dependences that may appear when the loop is unrolled in that direction. The decrease is achieved by removal of instructions, scalar replacement, and reuse of instructions. In detail, for each flow- and output-dependence, after calculation of the decrease in the number of instructions in a loop body when the loop is unrolled enough times to make the dependence appear, and calculation of the average decrease in each dependence when the loop is unrolled once, the sum of the average decreases is the decrease in the number of instructions achieved by removal of instructions and scalar replacement when the loop is unrolled once in that direction. The decrease in the number of instructions achieved by reuse is calculated as the number of reuses that exist in the dependence graph.

Taking Fig. 2 for an example, analysis of Fig. 6 shows that there is a decrease of two instructions and an increase of one unit in the length of the longest path when the loop is unrolled once in direction **j**, and a decrease of three instructions and no increase in the length of the longest path when the loop is unrolled once in direction **i**.

4 Method overview

4.1 Unimodular loop transformation

If we want to improve the efficiency of the execution by blocking or unrolling nested loops, the more permutable loops there are, the better the result is. Therefore, before actual loop unrolling, we maximize the number of permutable loops, using unimodular loop transformations [3].

4.2 Unrolling nested loops

The number of times unrolling is performed for each direction is selected in such a way as to maximize the efficiency of loop unrolling, using the information on dependence and reuse. The decision algorithm is described in Section 6. The algorithm has a heuristic evaluation based on the efficiency of the execution of a loop, which is described in Section 5.

5 Estimating the efficiency of the execution of a loop

Of the many factors determining the efficiency of the execution of a loop, we believe that loop-carried dependences and reuses over iterations are the most important. This section describes a method for estimating the efficiency by using this information.

The efficiency of the execution of a loop improves according to the number of times unrolling is performed. However, the improvement saturates when the number of instructions in a loop body exceeds a machine's capacity for parallel execution. Furthermore, the efficiency improvement is also limited by the number of registers and capacity of the cache memory. Therefore, it is very important to determine the optimal number of times for unrolling.

Generally, the more the parallelism of a machine is, the shorter the execution cycle (time) of a program

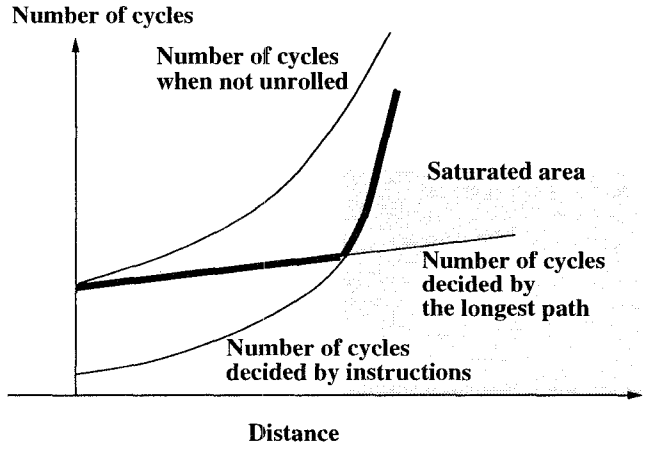

Figure 7: Saturation of the efficiency of loop execution

becomes. But no matter how much the parallelism of a machine is, there is always a lower bound (software bound) on the execution cycle. Similarly, the more the parallelism in a program, the shorter the execution cycle of that program. But no matter how much parallelism a program contains, there is always a lower bound (hardware bound) on the execution cycle. The former cycle (software bound) is the one needed to execute the longest path of a dependence graph, and is determined by the dependences in a program. The latter cycle (hardware bound) is the one needed to execute instructions if all the instructions are executed in parallel, and is determined by the number of instructions, the parallelism of a machine, and the number of cycles needed to issue the next instructions consecutively. While a loop is being unrolled, the latter exceeds the former. When the latter exceeds the former, the efficiency of the loop execution starts to saturate, and the number of cycles needed to execute a program increases.

Figure 7 shows how the efficiency of loop execution reaches saturation. In this figure, Distance represents the length of the vector as the number of times unrolling is performed in each direction, and the thick line shows the actual execution time after loop unrolling. As will be described later, when n-fold nested loops are unrolled, the number of cycles determined by instructions in the loop increases by the nth power of D, where D represents distance. On the other hand, the number of cycles determined by the longest path of the dependence graph increases by the first power of D. The effectiveness of loop unrolling is the difference between the number of cycles in an unrolled body and that of a pre-unrolled body. The effectiveness decreases drastically when saturation of the efficiency of loop unrolling occurs.

To determine the optimal number of times unrolling should be performed, it is necessary to estimate the efficiency of loop execution – that is, the number of iterations of an original loop in an unrolled loop per cycle (the number of iterations / the execution cycle)

– before actually starting loop unrolling.

The target is assumed to be a fully permutable loop obtained by unimodular loop transformations. All the directions of loop unrolling are numbered from the outside in. The number of times unrolling should be performed, k_i ($k_i = 1$ when it is not unrolled), is determined by using the following parameters:

$m(> 0)$ The number of instructions that can be executed in one cycle in parallel

$p(> 0)$ The number of cycles in which the next instructions can be issued

$N(> 0)$ The number of instructions in a loop body before unrolling

$C(\geq c_i)$ The number of cycles of the longest path of a dependence graph before unrolling

$c_i(\geq 0)$ The increase in the number of cycles when a loop is unrolled once in direction i

$n_i(\geq 0)$ The decrease in the number of instructions when a loop is unrolled once in direction i

Of these parameters, m and p are constant values determined by the architecture of a machine. The others are determined by the characteristics of a program. They can be calculated by analysis of the dependence and reuse of the program, as described above.

Assuming $\vec{k}$ to be the vector consisting of the number of times unrolling is performed in each direction, we have the following formulas:

- The number of cycles determined by instructions in the loop

$$C_p(\vec{k}) \equiv C + \sum_i \{c_i(k_i - 1)\}$$

- The number of cycles determined by the longest path of the dependence graph after unrolling

$$C_h(\vec{k}) \equiv (N \prod_i k_i - \sum_i (n_i(k_i - 1) \prod_{j \neq i} k_j))p/m$$

- The condition of saturation (saturated when it is true)

$$S(\vec{k}) \equiv C_h(\vec{k}) > C_p(\vec{k})$$

- The number of iterations per cycle (the efficiency of loop execution)

$$P(\vec{k}) \equiv \frac{\prod_i k_i}{\max(C_p(\vec{k}), C_h(\vec{k}))}$$

When no reuse is available (in other words, when $n_i = 0$ for each i) the efficiency $P(\vec{k})$ has a maximum value when $S(\vec{k})$ is true, and then $P(\vec{k})$ keeps $m/(pN)$ as a constant value. Therefore, every point in the graph at which $S(\vec{k})$ is true provides an optimal value. However, the decrease in the number of instructions resulting from reuses in loop unrolling slightly increases the efficiency as k_i increases, and the efficiency approaches

$$\frac{m}{p(N - \sum_i n_i)}$$

at the limit of $\forall i(k_i \to \infty)$. There are many points in the saturated area that provide a lower value than the limit value. However, we consider that all the points in the saturated area provide optimal values, because they are almost same.

The number of times unrolling is performed should be close to the boundary of the saturated area, because the rate of increase of the efficiency is much lower in a non-saturated area than in a saturated area. We consider that all the points in the saturated area provide optimal values, but it is necessary to choose the point for which unrolling is performed the fewest times, in order to decrease the number of instructions in the unrolled body. The number of times and the directions in which unrolling is performed are determined by the algorithm described in the next section.

6 Algorithm

This section describes a method for calculating a set of the numbers of times unrolling is performed in all directions $\{k_i\}$. The values of k_i are limited to natural numbers. Initially, every value is 1. Next, the direction i, in which the loop can be most effectively unrolled, is selected, and 1 is added to $\{k_i\}$ until $\{k_i\}$ enters the saturated area or the number of instructions exceeds the capacity of an instruction cache. The algorithm can be represented as follows:

1. For each value of i, set 1 to k_i.
2. Calculate S for $\{k_i\}$, and terminate the algorithm if S is true.
3. Terminate the algorithm if the number of instructions exceeds the capacity of an instruction cache.
4. For each value of i, calculate

$$P_i \equiv P(\{k_1, .., k_i + 1, .., k_n\}),$$

   select the value of i for which the value of P_i is maximum, and add 1 to k_i.
5. Go to step 2.

According to the formula for determining the efficiency of loop execution P, P increases monotonically until it saturates. Therefore, this algorithm never chooses directions that lead to local maximum points. Moreover, it can find the point for which unrolling is performed the fewest times, because it chooses the nearest saturated point.

7 Evaluation

In this section, we evaluate our method by using Livermore Fortran Kernels, which are benchmark programs for parallel architectures. In these kernels, loops Nos. 4, 8, 18, 21, and 23 are selected, because they have nested loops. We estimate the optimal unrolling times of the loops for a VLIW processor, and compare the efficiency of the loop execution, which is the number of iterations of an original loop per cycle (the number of iterations / the execution cycle).

Table 1 shows the relative efficiencies of the executions of the loops and the estimated unrolling times. Each value of the efficiency is relative to the efficiency of loop execution when the loop is not unrolled. Each number of Unrolling Vector(UV) is represented by a vector consisting of the number of times unrolling is performed for all the directions from the outside in. Each number of UV in innermost loop unrolling is the number of times the loop is unrolled for the efficiency of loop execution to reach saturation without using our method. We suppose that the target machine can execute four floating- and fixed-point instructions, four load/store instructions, and four jump instructions per cycle, and assume that all the instructions are executed in one cycle, except for floating-point and load/store instructions, which require two cycles. In addition, the machine is assumed to have a sufficient number of registers and to have some hardware support for speculative movement of instructions such as TORCH [7].

The values in Table 1 show that higher efficiencies are obtained by unrolling nested loops. This is because more instructions are reused and executed in parallel than with innermost loop unrolling. The improvements in efficiency are especially notable for kernels Nos. 21 and 23. In kernel No. 21, the outermost loop is unrolled three times, and the innermost two loops twice. In kernel No. 23, each loop is unrolled five times. These programs have loop-carried dependences, and therefore the enough parallelism can not be obtained by innermost loop unrolling. When nested loops are unrolled simultaneously, the number of instructions increases by the nth power of the distance (described in section 5), where n is the number of nested loops, while the number of cycles needed increases by the first power of the distance. The parallelism in a program is determined by the number of instructions can be executed in parallel. Therefore, unrolling of nested loops provides higher parallelism.

Next, we describe the adequacy of the method used

Table 1: The relative efficiency of loop execution for each kernel

	Innermost loop		Nested loops	
No.	UV	Efficiency	UV	Efficiency
4	{1,5}	5.00	{3,2}	6.67
8	{1,1}	1.00	{1,1}	1.00
18	{1,2}	1.43	{1,2}	1.43
21	{1,1,6}	5.41	{3,2,2}	7.41
23	{1,1}	1.00	{5,5}	2.99

		k1							
k3	k2	1	2	3	4	5	6	7	8
1	1	0.143	0.222	0.273	0.308	0.333	0.353	0.368	0.381
1	2	0.286	0.444	0.545	0.615	0.667	0.706	0.737	0.762
2	2	0.571	0.889	1.091	1.143	1.176	1.200	1.217	1.231
2	3	0.828	1.043	1.143	1.200	1.237	1.263	1.282	1.297
3	3	0.857	1.091	1.200	1.263	1.304	1.333	1.355	1.371
3	4	0.873	1.116	1.231	1.297	1.341	1.371	1.394	1.412
4	4	0.889	1.143	1.263	1.333	1.379	1.412	1.436	1.455
4	5	0.899	1.159	1.283	1.356	1.404	1.437	1.462	1.481
5	5	0.909	1.176	1.304	1.379	1.429	1.463	1.489	1.509

Figure 8: Estimated efficiency of loop execution of kernel No.21

		k1							
k3	k2	1	2	3	4	5	6	7	8
1	1	0.143	0.222	0.300	0.364	0.417	0.462	0.500	0.533
1	2	0.286	0.444	0.545	0.667	0.714	0.750	0.778	0.842
2	2	0.571	0.800	0.923	1.000	1.053	1.091	1.120	1.143
2	3	0.667	0.923	1.059	1.143	1.154	1.200	1.235	1.263
3	3	0.818	1.059	1.174	1.241	1.286	1.317	1.340	1.358
3	4	0.857	1.091	1.200	1.297	1.333	1.358	1.377	1.412
4	4	0.889	1.143	1.263	1.333	1.379	1.412	1.436	1.455
4	5	0.870	1.143	1.277	1.356	1.389	1.429	1.458	1.481
5	5	0.893	1.163	1.293	1.370	1.420	1.456	1.483	1.504

Figure 9: Measured efficiency of loop execution of kernel No.21

to determine the number of times and the directions in which loop unrolling is performed. Figures 8 and 9 show the estimated and actually measured efficiencies of the executions of loops when kernel No. 21 is unrolled. Figures 10 and 11 show the estimated and actually measured efficiencies of the executions of loops when kernel No. 23 is unrolled. Each value is obtained by dividing the number of iterations by the number of cycles. In each figure, the meshed area indicates that the efficiency is saturated, and thick rectangles show the points that provide the actual optimal efficiencies.

First, the adequacy of the method for determining the number of times is described. Figures 9 and 11 show that the efficiency is not greatly improved in the saturated area. In Fig. 9, the average increase in efficiency at each of the adjoining two points is 1.800 in the non-saturated area, and 1.028 in the saturated area. In Fig. 11, the average increase in efficiency at each of the adjoining two points is 1.175 in the non-saturated area and 0.6696 in the saturated area. Thus, there is an obvious difference between the gains in non-saturated areas and in saturated areas. Therefore, we believe that loops are most effectively unrolled on the boundaries of saturated areas. This method

k2 \ k1	1	2	3	4	5	6	7	8
1	0.067	0.074	0.077	0.078	0.079	0.080	0.080	0.081
2	0.074	0.103	0.118	0.127	0.133	0.138	0.141	0.144
3	0.077	0.118	0.143	0.160	0.172	0.182	0.189	0.195
4	0.078	0.127	0.160	0.184	0.202	0.212	0.213	0.213
5	0.079	0.133	0.172	0.202	0.213	0.214	0.214	0.214
6	0.080	0.138	0.182	0.212	0.214	0.214	0.215	0.215
7	0.080	0.141	0.189	0.213	0.214	0.215	0.215	0.216
8	0.081	0.144	0.195	0.213	0.214	0.215	0.216	0.216

Figure 10: Estimated efficiency of loop execution of kernel No.23

k2 \ k1	1	2	3	4	5	6	7	8
1	0.067	0.080	0.086	0.089	0.091	0.092	0.093	0.094
2	0.080	0.114	0.133	0.146	0.154	0.160	0.165	0.168
3	0.086	0.133	0.163	0.177	0.183	0.190	0.194	0.199
4	0.089	0.146	0.177	0.186	0.194	0.199	0.203	0.205
5	0.091	0.154	0.183	0.194	0.200	0.204	0.207	0.211
6	0.092	0.160	0.190	0.199	0.204	0.208	0.211	0.213
7	0.093	0.165	0.194	0.203	0.207	0.211	0.214	0.215
8	0.094	0.168	0.199	0.205	0.211	0.213	0.215	0.216

Figure 11: Measured efficiency of loop execution of kernel No.23

can find an adequate number of times that provides the optimal efficiency, because it increases the number of times until the efficiency is saturated. In Fig. 9, the efficiency is saturated when the method chooses the point $\{k_1,k_2,k_3\}=\{3,2,2\}$ from $\{2,2,2\}$, and the optimal numbers of times and directions are obtained.

Second, the adequacy of the method for determining the directions is described. Though every point on the boundary of the saturated area is supposed to provide optimal efficiency, this does not mean that any point will do. For example, in Fig. 11, the point $\{4,8\}$ has 1.3 times as many numbers of times unrolling is performed as the point chosen by our method. Therefore, to decrease the number of instructions in the unrolled body, it is necessary to choose the point for which unrolling is performed the fewest times, because the numbers of times unrolling is performed are different in the saturated area. At each point, our method chooses the direction that increases the efficiency the most. The efficiency is guaranteed to increase monotonically, so we can find the point for which unrolling is performed the fewest times. This is guaranteed by the function P described in section 5. The characteristic of the function is imaginary but this is sufficiently practical to search for the point for which unrolling is performed the fewest times. For example, the process shown in Fig. 8 of searching for the point shows that our method always chooses adequate directions according to comparison with Fig. 9, and this indicates the adequacy of our method. In addition, comparisons of Figs. 8 and 9 and Figs. 10 and 11 show that the change in the estimated efficiency is very similar to the change in the actually measured efficiency, and that they are identical at 93% of points. Therefore, choosing directions by using estimated efficiencies turns out to be adequate.

8 Conclusions

In this paper, we introduced a method for estimating the optimal number of times for unrolling nested loops, and described the characteristics of the method. We also explained how a decrease in the number of instructions in a loop body and an increase in the length of the longest path of a dependence graph affect the efficiency of the execution of a loop. Finally, we proposed an algorithm that determines the optimal number of times that unrolling should be performed for each direction. By using this method, which unrolls outer loops simultaneously, rather than methods unrolling only the innermost loop, we can take advantage of the higher level of parallelism of instruction-level parallel machines. We are now researching a faster algorithm with a deep analysis of loop-carried dependences and reuses. This algorithm will take account of the effect of code scheduling and register allocation.

References

[1] Weiss, S. and Smith, J. E.: A Study of Scalar Compilation Techniques for Pipelined Supercomputers, *Proc. 2nd International Conference on Architectural Support for Programing Languages and Operating Systems*, pp. 105–109 (1987).

[2] Su, B., Ding, S. and Xia, J.: URPR–An Extension of URCR for Software Pipelining, *Proc. MICRO-19*, pp. 104–108 (1986).

[3] Wolf, M. E. and Lam, M. S.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 452–471 (1991).

[4] Carr, S. and Kennedy, K.: Improving the Ratio of Memory Operations to Floating-Point Operations in Loops, *ACM Trans. Programming Languages and Systems*, Vol, 16, No. 6, pp. 1768–1810 (1994).

[5] Wolf, M. E. and Lam, M. S.: A Data Locality Optimizing Algorithm, *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30–44 (1991).

[6] Amarasinghe, S. P. and Lam, M. S.: Communication Optimization and Code Generation for Distributed Memory Machines, *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 126–138 (1993).

[7] Smith, M. D., Lam, M. S. and Horowitz, M. A.: Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 344–354 (1987).