A Global Code Scheduling Technique Using Guarded PDG

Akira Koseki[†] Hideaki Komatsu[‡] Yoshiaki Fukazawa[†]

[†]School of Science & Engineering, Waseda University
3-4-1 Okubo, Shinjuku-ku,
Tokyo 169, Japan
{koseki,fukazawa}@fuka.info.waseda.ac.jp

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

Abstract

For instruction-level parallel machines, it is essential to extract parallelly executable instructions from a program by code scheduling. In this paper, we propose a new code scheduling technique using an extension of PDG. This technique parallelizes non-numerical programs, producing better machine codes than these created by percolation scheduling.

1 Introduction

Much research has been done on parallel processing at the instruction level with the aim of realizing high-performance computers. At this level, code scheduling is very important for obtaining parallelized code.

The purpose of code scheduling is to transform instructions so that they can be executed as fast as possible in accordance with the target architecture. In the case the of instruction-level parallelism, "code scheduling" means moving instructions and scheduling them to be executed in parallel for maximum efficiency. In general, code-scheduling techniques can be divided into two categories: local scheduling and global scheduling. A typical example of the former is List Scheduling [?], while examples of the latter are Trace Scheduling [?] and Percolation Scheduling [?].

In this paper, we propose a new code-scheduling technique that can parallelize large parts of non-numerical programs to which Trace Scheduling cannot be applied. It generates a more efficient code than Percolation Scheduling.

2 Background of this research

In the extraction of parallelly executable instructions from a program, dynamic behavior such as conditional branching is an obstacle to effective parallelization, since the behavior of a program cannot be determined until it is actually executed.

Trace Scheduling can make programs highly parallel by means of branch prediction. This type of scheduling is known to be efficient for numerical programs, but it involves problems when applied in other fields where branch prediction cannot be used.

In Percolation Scheduling, instructions in a program are moved toward the top of the program by four basic transformation rules. This method can parallelize non-numerical programs by moving instructions in both directions from the branch instruction, but it involves the following three problems:

1. Transformations in Percolation Scheduling do not always shorten the execution time of a program.
2. A transformation rule may produce anti-dependencies, which may interfere with other efficient transformations.
3. The application of the transformation rules is terminated in accordance with the number of arithmetic logical units (ALUs) in the target machine. As a result, an instruction occupying an ALU resource that does not affect the execution time of the program may prevent other efficient transformations.

To solve the above problems, we developed a new code-scheduling technique using Guarded Program Dependence Graph (GPDG), which is an extension of PDG [?], and the Static Single Assignment (SSA) [?] form

In this method, instructions in the innermost

0-7803-2018-2/95/$4.00 © 1995 IEEE

loop of a program that is transformed into SSA form are expressed by GPDG, and some basic transformations are applied to GPDG for parallelization. In this technique, the above problems are solved as follows:

1. The execution time of a code can always be estimated by interpreting GPDG, so our transformation guarantees that the execution time for GPDG will be shortened.

2. Anti-dependence of variables cannot be generated with the SSA form. Therefore, our transformation is not terminated by anti-dependence.

3. We can determine what instructions are critical to the execution time by interpreting GPDG. Therefore, such instructions occupy resources before others, and non-critical instructions occupy the resources after them so as not to prevent other efficient transformations.

This paper defines GPDG and its characteristics, and shows how to transform GPDG in order to parallelize a program. Later, an example of parallelization is given, and our method is evaluated. Because of restrictions on the length of the paper, a description of physical code generation from GPDG is omitted.

3 Definition of GPDG

3.1 Dependencies in a program and GPDG

Between instructions, there are some dependencies that indicate the order of the execution. If none exist, all instructions can be executed simultaneously, provided the machine has enough ALUs.

These dependencies have been widely used in research on program optimization, most notably by Ferrante et al. in PDG [4]. We use GPDG which is an extension of PDG to parallelize programs. One of the most important characteristics of GPDG is that every control dependence is expressed by a "guard", which indicates whether an instruction can be executed or not. Control dependence is described as the relation between the producer of a guard and its consumer. It is thus similar to data dependence. Therefore we can treat data dependence and control dependence in a single graph in the same way. These benefits are described in detail in Section 4.1.

3.2 Constitution of GPDG

GPDG is a graph whose nodes are instructions of a program and whose edges indicate dependence. Each node includes the source and destination of data produced by the corresponding instruction and its guard.

The types of these edges are classified as follows:

1. Data dependence

When a certain instruction A produces a set of data and another instruction B uses them, B cannot be executed until the data have been produced. This kind of relation is called data dependence.

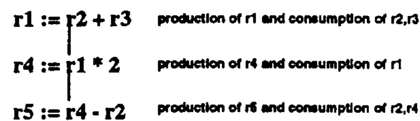

Fig. 1: Data dependence

2. Control dependence

In Von Neumann architecture machines, the execution of a program is controlled by changing the internal state of the machine. After a branch, instructions cannot be executed until the change of the state is complete. This kind of relation is called control dependence. We show this dependence as the relation between a producer of a guard and its consumer. In Fig. 2, the braced part of each node represents a guard. An instruction with (cc) is executed when the conditional register cc is true. An instruction with the guard !cc is also executed when cc is false.

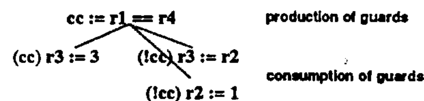

Fig. 2: Control dependence

3. Resource dependence

A relation between instructions that are not executed at the same time because of restricted resources is called resource dependence. Anti-dependence and output dependence are classified as belonging to this category.

4. Other types of dependence

There are some other cases in which a certain pair of instructions cannot be executed simulta-

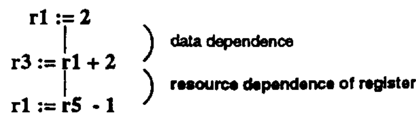

Fig. 3: Resource dependence

neously. For example, a branch instruction must not fire before previous instructions.

By using these dependences and some nodes, we can express any instruction in the innermost loop of a program as a graph. The execution of instructions satisfying the partial order of GPDG is equivalent to the execution of the original program. An example of GPDG is shown in Fig. 4-a.

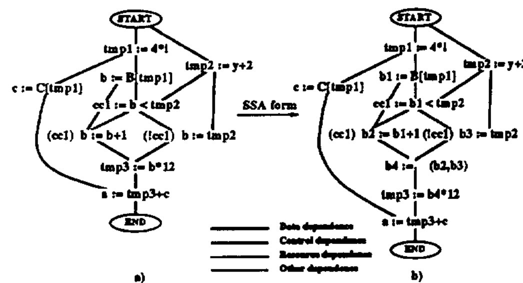

Fig. 4: Example of GPDG

3.3 Introducing the SSA form into GPDG

In this algorithm, the SSA form for removing anti-dependence is introduced. SSA transformation, which means modifying the original code to convert it into SSA form, is performed as follows:

1. Change a program into a representation where there is only one assignment for each variable. Variables are given new names (for example, a variable x is renamed x1,x2,..).
2. To maintain the meanings of the original code, Φ functions [5] are inserted at certain points called dominance frontiers to distinguish incoming control flows.

Figure 4-b shows a GPDG transformed into SSA form. In this graph, all anti-dependence is eliminated and several Φ functions are inserted at necessary points.

4 Characteristics of GPDG

4.1 The longest path of GPDG and its optimization

In List Scheduling (with the critical path method) [1], a data flow graph of each basic block is generated and the longest path in each graph is used for scheduling.

In our method, a control dependence is expressed as a relation between the producer of a guard and its consumer. Therefore, we can treat control dependences in the same way as data dependences. In GPDG, the instructions in the innermost loop, including branches, are expressed as a single graph. Therefore, we can regard GPDG as a "global data flow graph," and can obtain very efficient code by using a code generation method in which the number of execution cycles in the longest path remains constant.

We use the longest path of GPDG for optimization, but it is necessary to extend the notion of such a path in GPDG, to take account of branches. For example, when the longest path of GPDG includes a node with a guard that is executed when a conditional register is true, and the value of the conditional register is normally false, scheduling the instructions on the basis of this longest path makes no sense. Therefore, it is necessary to simultaneously use the longest paths of the subgraphs produced by the division of GPDG according to branches and the probability of branches.

GPDG, including branches, is divided into subgraphs corresponding to various combinations of branches. Each subgraph has its own longest path and probability. The probabilities of branches can be obtained from the execution profile of a program. If this is not provided, the probability of a branch is assumed to be 50%. The probability of a branch for exiting a loop is treated as 100%.

4.2 Relation between GPDG and the execution time of a program

In our method, the estimated execution time of a program is used to transform GPDG. Instead of the actual execution time, we use the number of machine cycles needed for the execution of programs. To estimate the number of cycles, we use the longest paths of the subgraphs produced by the division of GPDG according to combinations of branches and their probabilities.

We estimate the number of cycles as follows:

Estimated value = $\Sigma_{combination\ of\ branches}$ P × S,
where P is the probability when the combination is taken and S is the number of cycles in the longest path of the subgraph corresponding to the combination.

For example, in Fig. 4-b, we assume the condition b<tmp2 for c_1 and its negation for $!c_1$. Each combination is represented as an ordered n-tuple C∈({c1,!c1},{c2,!c2},...). The probability of the occurrence of C is denoted by P(C), the longest path of the subgraph corresponding to C is denoted by L(C) (it is expressed as a sequence of nodes), and the number of cycles needed for the path is denoted by S(L(C)). The estimation of the execution time of the graph Est(G) is represented as

$P(c_1)\times S(L(c_1))+P(!c_1)\times S(L(!c_1))$.

The value of P is calculated by using an execution profile if one exists. If not, P is set in the way described in the previous section. L is calculated by tracing the graph.

After P and L have been calculated, Est(G) is obtained by calculating S(L). Since a sequence of nodes shows the order of instructions in a critical path, S(L) can be calculated when the number of cycles each instruction takes is known. However, a load instruction and a branch instruction may take various numbers of cycles, according to the architecture. We use some special nodes corresponding to the delay of an instruction and a jump according to the target architecture, and insert them into GPDG to absorb these difference, as shown in Fig. 5. In this figure, we assume that the delay of a load instruction is two machine cycles, that the delay when a branch instruction is taken is one cycle, and that if the branch is not taken, the next instruction is executed without a break. The direction of a branch instruction is determined by its probability, and new nodes are inserted to prolong the path that has lower probability. If the probability is biased, new nodes are inserted to prolong the path that includes nodes having high probability.

In a graph containing such nodes, S(L) is equivalent to the length of the path.

5 Transformation of GPDG

5.1 Shortening the execution time

In this subsection, we describe a actual method of shortening the execution time of instructions by transforming GPDG. Essentially, it consists of

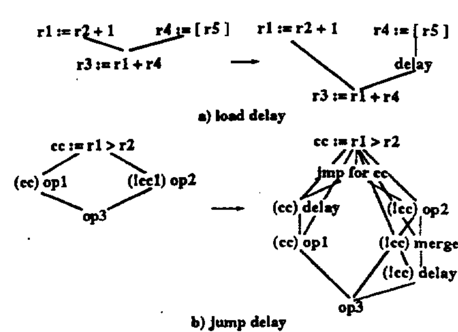

Fig. 5: Insertion of delay nodes

shortening the longest path of each subgraph corresponding to a combination of branches through graph transformations.

Our transformation rules consist of
- Speculative moving
- Node splitting

To optimize a program, each transformation is applied step by step to each longest path.

5.1.1 Speculative move transformation

Some instructions can be speculatively executed before the conditional value of a branch has been determined. In this way, we can shorten the longest paths. This transformation involves eliminating the edge of control dependence in GPDG. Figure 6 shows an example in which instruction 2 is speculatively moved by elimination of the edge. On the assumption that the longest path includes a sequence of nodes 1-2-4-5, it is transformed into a sequence 1-4-5 through this transformation. Consequently, it can be expected that the number of cycles for the execution of the code will be shortened by about (the probability that r7 == 2) × (the decrease in the length of the path).

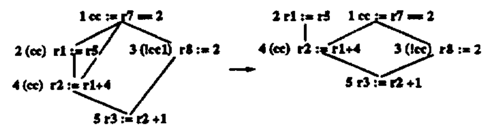

Fig. 6: Speculative move transformation

The targets of this transformation are limited to instructions that do not change the meanings of a program even if branch prediction fails. Recently,

architectures have been proposed that guarantee the invariance of program semantics if branch prediction fails [6, 7], so our system obtains what instructions can be executed speculatively from the specification of a target architecture, and instructions that cannot be executed speculatively are marked in order to inhibit transformation (in Fig. 8, such instructions are marked ✱).

5.1.2 Node split transformation

Node split transformation produces new instructions to be moved speculatively. In some cases, the path is shortened by this transformation.

Node split transformation involves duplicating a node dominated by a Φ function. Figure 7 shows an example in which the instruction that assigns [register 4] - 5 to register 5 is split. In splitting, we prepare new available registers 6 and 7, and change the destination register of the copied instructions to the new registers. The Φ function is then adjusted to set the final result to register 5.

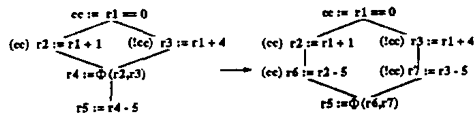

Fig. 7: Node split transformation

5.2 How to apply transformations

The algorithm for applying transformations is as follows:

1 Choose a path

- 1.1 Choose the path whose probability is highest and which is not yet marked with a termination mark. If more than one such path exists, choose all of them.
- 1.2 If more than one path is chosen, choose the longest path. If more than one longest path exists, choose all of them.
- 1.3 If more than one path is chosen, chose the path that has speculatively movable nodes. If more that one path such nodes, choose all of them.
- 1.4 If more than one path is chosen, chose a path at random.
- 1.5 If no path is chosen, terminate this algorithm.

2 Choose a transformation to apply

- 2.1 If the path has speculatively movable nodes, apply speculative move transformation and eliminate redundancy in the expressions. If it is recognized that application of the transformation will have no effect, nothing will be performed. This recognition rule is described in Section 5.3.
- 2.2 If the path has no speculatively movable nodes, apply node split transformation. If the length of the path is shortened after this transformation, go to step 3. Otherwise, apply speculative move transformation to the split node. If it is recognized that application of speculative move transformation will have no effect, split transformation itself will not be performed.
- 2.3 If no transformation is to be applied, mark the path with a termination mark.

3 Recalculate the length of the path and go to step 1.

5.3 Restriction on node splitting

If node splitting is applied without any restriction, the number of instructions will increase exponentially. To avoid this, we use the following heuristics.

We assume that all nodes in each longest path provisionally occupy ALU resources. As a result of the continued application of transformations, the longest path is shortened and its nodes come to have more than one child. If the application of transformations is continued still further, it will have no effect, since the number of nodes in the longest path will exceed the number of ALUs in the target architecture. Therefore, a transformation can be inhibited by watching the growth of the path to determine whether the transformation takes effect or not.

5.4 An example of the shortening of paths

An example of the shortening of paths in a graph is shown in Fig. 4-b. Here, a branch condition (b<tmp2) is denoted by c1, and the probability

that it is true is assumed to be 60%. Figure 8 is a graph containing load delay nodes, jump nodes, and jump delay nodes according to the architecture described in Section 4.2. Jump nodes and jump delay nodes are inserted to prolong the path that is executed when c1 is false, since the probability that c1 is true is assumed to be high.

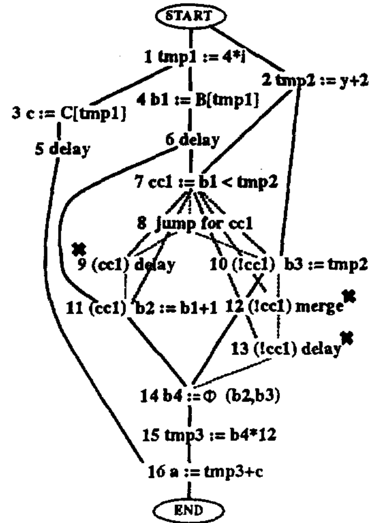

Fig. 8: Insertion of nodes

We assume that variable a is valid only after execution of the program expressed by this GPDG.

In Fig. 8,
P(c1)=0.6, P(!c1)=0.4,
L(c1)=start-1-4-6-7-8-9-11-14-15-16-end,
and
L(!c1)=start-1-4-6-7-8-10-12-13-14-15-16-end,
so
S(L(c1))=11, S(L(!c1))=12.
Therefore, the estimated number of execution cycles Est(G) is 11.4 in the initial state.

Transformations of the graph are now performed according to the algorithm described above. At first, the path L(c1) is chosen to be transformed, since its probability is the highest. In this path, node 11 is speculatively movable, so speculative move transformation is applied (Fig. 9-a), as follows:

1. The edge from the instruction that produces a guard is eliminated along with the guard of target node.
2. A new edge (between nodes 11 and 9) is generated so that the moved instruction can be executed before the branch.
3. An additional edge (the edge between nodes 9 and 14) is generated to maintain the form of the graph.
4. Redundancies in the expressions are eliminated.

After this transformation, L(c1) is transformed into L(c1)=start-1-4-6-7-8-9-14-15-16-end.

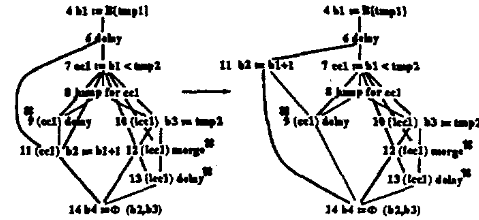

Fig. 9: Application of speculative move transformation

Next, L(c1) is again chosen for transformation. We can apply node split transformation to it.

Node 15 is split and the graph is transformed as follows (Figs. 10-a and 10-b):

1. The target node is copied over the Φ function, and the source register of the copied node is changed. The destination register of the Φ function is changed to guarantee single assignment rules.
2. The copied node is marked to inhibit speculative moving if the instruction corresponding to the node is not speculatively movable in the target architecture.
3. The transformation is terminated if the length of L is shortened (without a speculative move).
4. Redundancies in the expressions are eliminated (Figs. 10-b and 10-c). The transformation is terminated if the length of L is also shortened.
5. If the copied instruction is not speculatively movable, this transformation is terminated and GPDG is returned to the state it was in before copying began. If the copied instruction is speculatively movable, a speculative move is performed and this transformation is terminated (Figs. 10-c and 10-d). If a speculative move is recognized as having no effect on the heuristics described in Section 5.3, this transformation is terminated and GPDG is returned to the state it was in before copying began.

As a result, L(c1) is transformed into

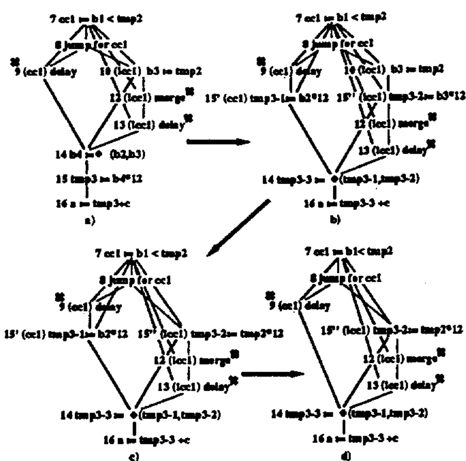

Fig. 10: Application of node splitting and speculative move transformations

start-1-4-6< $\begin{matrix} 7\text{-}8 \\ 11\text{-}15' \end{matrix}$ >9-14-16-end,
and L(!c1) is transformed into
start-1-4-6-7-8-15"-12-13-14-16-end;
thus Est(G) becomes 9.8.

L(c1) does not have any nodes that can be moved speculatively or split (splitting of node 16 is not performed because it has no effect), and therefore the target for the application of transformations is changed to L(!c1). Some transformations are applied, and the final result is shown in Fig. 11.

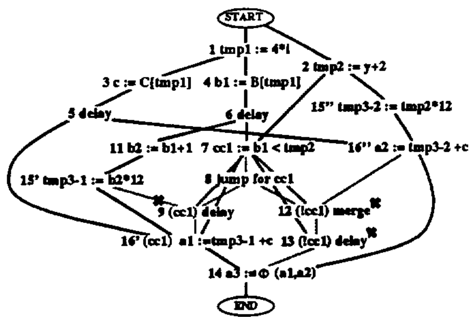

Fig. 11: The final result after application of some transformations

Consequently, each longest path is transformed into

L(c1)=start-1-4-6< $\begin{matrix} 7\text{-}8 \\ 11\text{-}15' \end{matrix}$ >9-16-14-end,
L(!c1)=start-1-4-6-7-8-12-13-14-end,
and Est(G) becomes 9.

6 Discussion

6.1 Properties of our transformations

In our method, as in Percolation Scheduling, a program is parallelized by the application of some transformation rules. This section describes the difference between Percolation Scheduling and our method.

The logically minimum execution time of a program depends on the relationships of the producer and consumer chains in the program. In the program, however, there are other dependences, such as control and resource dependences, which extend the execution time of the chain. For example, in the control flow graph in Fig. 12, the logically minimum execution time of the program is the time for execution of the path a-c-d-g if both branches to the left are taken. In this case, we can realize the minimum execution time if a-c-d-g is scheduled without any break. In Percolation Scheduling, there is no way to schedule this path without a break. Furthermore, only changes in the length of each critical path in the basic blocks are considered. Therefore, there is no index of the reduction in the total execution time.

In comparison with Percolation Scheduling, our method always shortens the longest paths, which affect the execution time. This means that applying our transformation always reduces the total execution time.

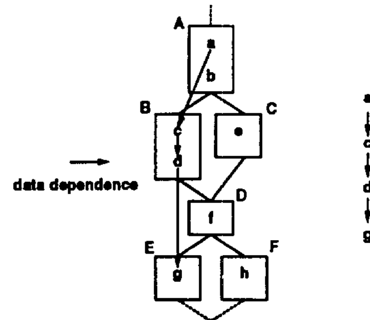

Fig. 12: Data dependence in a control flow graph

6.2 Effects of using the execution profile

We can find the logical minimum execution time for Fig. 12 in Trace Scheduling if all the biases of the directions of branches are given in advance.

Like Percolation Scheduling, our method can parallelize both directions of branches, and therefore it can optimize programs that have few biases of branches. Additionally, if a profile is available and the probability of each branch is known, the paths that have higher probability (corresponding to the trace in Trace Scheduling) are optimized before others. Therefore, if a profile is available and all biases of branch directions are known, we can get the same effect as with Trace Scheduling. If not, we can still get better codes than with Percolation Scheduling.

7 Evaluation

Table 1 shows the speedups of the codes of Stanford Benchmark programs optimized by our method. The innermost loops of the programs are unrolled several times and each recursive call is translated into simple loops. As a target for evaluation, we chose a VLIW architecture that has some hardware support for speculative moves like TORCH [7], and we evaluated our method on some machines with various numbers of parallel ALUs.

All the values in the table are speeds relative to those when the original programs are executed on a scalar processor. For example, in Permute, the optimized program is executed 1.6 times faster than the original code on VLIW with two parallel ALUs, and it is executed 2.3 times faster on VLIW with six parallel ALUs. The data in the column of ∞ means that the scheduling is performed with enough ALUs for the program. On this architecture, a 2.3 times speedup is achieved.

Table 1 also shows that for most programs the effect of our method is saturated in more than 6 parallel ALUs. This means that the programs do not have very large inherent parallelism, and therefore their longest paths cannot be further shortend. Thus, near six parallel ALUs, the factor determining the execution speed is changed from the number of instructions and parallel units to the length of the longest path. On the other hand, in Fft and Extab, the longest path is so short in comparison with the number of instructions that its execution speed is raised along with the number of parallel ALUs. In those programs, the number of instructions in basic blocks is comparatively large and they can easily be optimized by other methods. It is reported [8] that the number of instructions in basic blocks is small in non-numerical programs, and therefore the use of an effective optimizing method for such programs is very important.

Table 1: Speedups of programs

Programs	Number of ALUs				
	2	4	6	8	∞
Permute	1.556	2.333	2.333	2.333	2.333
Quicksort	1.947	2.846	3.083	3.083	3.083
Remove	·1.976	3.000	3.000	3.000	3.000
Try	1.464	3.061	3.741	4.040	4.208
Innerproduct	2.000	3.818	5.250	5.250	5.250
Initmatrix	2.000	3.840	3.840	3.840	3.840
Initarray	1.816	2.300	2.300	2.300	2.300
Fit	1.841	2.025	2.025	2.025	2.025
Fft	2.000	3.902	5.714	7.273	16.00
Exptab	2.000	3.875	5.636	6.889	7.750
Bublle	1.950	3.900	5.571	5.571	5.571

Table 2 shows the ratio of the number of instructions to which graph transformations are applied to the total number of instructions in VLIW with eight parallel ALUs. Application of graph transformations is terminated when the longest path of a graph cannot be shortened any more, or when each applicable transformation is judged to have no effect because the longest path has many nodes at the same level. A node once moved speculatively will never be transformed, and a split node will be speculatively moved after it has been split a finite number of times. The number of nodes to which graph transformations are applied is represented as $Nn + \sum_i 2^{Ns_i}$, and its maximum value is $Nn + Nn * 2^{Nf}$. Here, the number of nodes is denoted by Nn, the number of Φ functions by Nf, and the number of node splits for a node i by Ns_i. As explained in Section 5.3, node splitting is performed with some restrictions, and the number of splits has an upper bound determined by the constant number of parallel ALUs and the longest path of the original code. As the real value of $\sum_i 2^{Ns_i}$ is expressed by the linear term of Nn, the order of the total number of transformed nodes is Nn. In Table 2, our system is terminated when transformations are applied for 74% of all the instructions at most and for 26% on average; this shows that our system can be terminated after a reasonable number of transformations.

Table 2: ratio of instructions applied transformations

Programs	number of instructions all	applied	ratio[%]
Permute	33	4	12
Quicksort	39	17	44
Remove	81	24	30
Try	105	24	23
Innerproduct	84	16	19
Initmatrix	96	25	26
Initarray	75	1	1.3
Fit	86	35	41
Fft	160	11	6.9
Exptab	62	19	31
Bublle	39	29	74
Average	–	–	26

8 Related work

For optimization of programs, representation methods that use a graph representing dependence of a program are described by Ferrante et al. [4] and Girkar et al. [9]. Our GPDG can express control dependence such as data dependence, taking account of the control dependence on the relation between the producer of a guard and its consumer. Therefore, we can easily implement the algorithms described in Sections 5.2 and 5.4. Other approaches using PDG for optimization have been proposed, such as Region Scheduling [10]. This method can be applied for coarse grain parallel architectures, but it has same problems in code movement as Percolation scheduling.

A new approach to Percolation Scheduling, named Trailbrazing [11] has also been proposed. This method can solve problem 3 in Section 2, but does not have an index for applying code-moving rules, and therefore cannot guarantee the effectiveness if each code movement.

9 Conclusion and future tasks

This paper has introduced a code-scheduling technique using GPDG ,and has described the algorithm, its characteristics, and other details. Two important future tasks will be to develop a technique for connecting the innermost loop and its outside code, and for combining the scheduling method with the register allocator.

References

[1] A.V. Aho and J.D. Ullman, "Principles of Compiler Design," *Addison-Wesley* 1977.

[2] J.R. Ellis, "Bulldog: A Compiler for VLIW Architectures," *MIT Press* 1985

[3] A. Aiken, and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, pp. 584-594 1988

[4] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349 1987

[5] R. Cytron, J. Ferrante, B.K. Rosen,M.N. Wegman, and F.K. Zadeck, "An Efficient Method of Computing Static Single Assignment Form," *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pp. 25-35 1989

[6] K. Ebcioğlu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, North Holland 1988

[7] M.D. Smith, M.S. Lam, and M.A. Horowitz, "Boosting Beyond Static Scheduring in a Superscalar Processor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344-354 1987

[8] A. Nicolau, and J.A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers*, Vol. C-33, No. 11, pp. 968-976 1984

[9] M. Girkar and C.D.Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 2, pp. 166-178 1992

[10] R. Gupta and M. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp. 421-431 1990

[11] A. Nicolau and S. Novack, "Trailblazing: A Hierarchical Approach to Percolation Scheduling," *Proceedings of International Conference on Parallel Processing*, Vol. 2, pp. 120-124 1993