A Register Allocation Technique Using Register Existence Graph

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

Optimizing compilation is very important for generating code sequences in order to utilize the characteristics of processor architectures. One of the most essential optimization techniques is register allocation. In register allocation that takes account of instruction-level parallelism, anti-dependences generated when the same register is allocated to different variables, and spill code generated when the number of registers is insufficient should be handled in such a way that the parallelism in a program is not lost. In our method, we realized register allocation using a new data structure called the register existence graph, in which the parallelism in a program is well expressed.

1 Introduction

The number of registers in a processor is limited; therefore, variables and pseudo-registers in the intermediate code used by a compiler (symbolic registers) should be mapped to a restricted number of registers. This mapping is called register allocation [1, 2, 3, 4, 5]. In a register allocator, the most important consideration is how symbolic registers are used and how many of them are live at the same time. Symbolic registers live at the same time cannot be mapped to the same registers; therefore, if the number of such symbolic registers exceeds the number of registers, some code changes are needed to decrease their number.

The mapping process has hitherto been performed by algorithms using a register interference graph [1, 2] that expresses the overlap of the lifespans of symbolic registers. It can map symbolic registers and decrease their number by spilling them out and in. In this process, anti-dependence, which arises when symbolic registers are mapped, and spill code are generated.

Register allocation using a register interference graph gives better code for scalar processors. However, register interference graphs cannot express parallelism among instructions [6]; therefore, it is difficult to handle the generation of spill code and anti-dependence without losing the parallelism.

This paper describes a new register allocation technique using a register existence graph that can express the interference among symbolic registers and the parallelism among instructions in a program simultaneously.

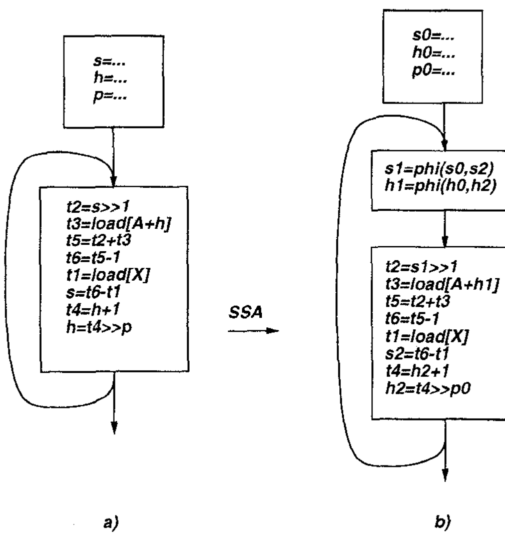


Figure 1: Source Program and SSA Transformation

2 Background

Generally, registers are allocated by coloring a register interference graph with the number of the registers. Register interference graphs are produced by scanning instructions in a program and by adding edges between nodes that indicate symbolic registers live at the same time.

Fig. 1 shows an example of the construction of a register interference graph.

Fig. 1a) shows the source program, and Fig. 1b) shows the target program after applying SSA (Static Single Assignment) transformation [7]. We assume the use of the hierarchical register allocation proposed by Callahan and Koblenz [3], so the target code should comprise the instructions in the innermost loop.

Then, we obtain a register interference graph by

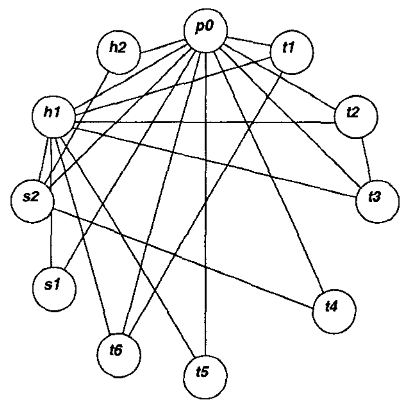


Figure 2: Register Interference Graph

```
h1:r1 p1:r2 s1:r3

r3=r3>>1
r4=load[A+r1]
r3=r3+r4
r4=r3-1
r3=load[X]
r3=r4-r3
r4=r1+1
r1=r4>>r2
```

Figure 3: Possible Register Allocation

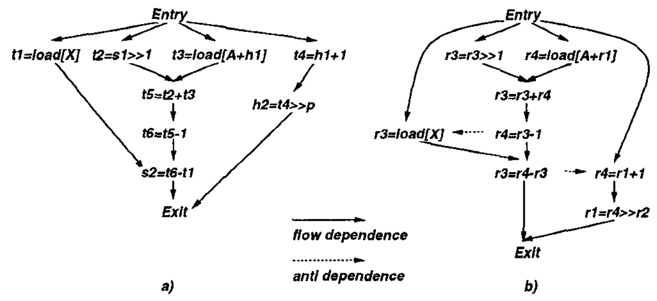


Figure 4: Loss of Parallelism

s1:r1 h1:r3 p0:r2

r1=r1>>1	r2=load[A+r3]	r3=r3+1	store[&p0] = r2
	r2=r1+r2		
r1=r2-1	r2=load[X]		
r1=r1-r2	r2=load[&p0]		
r3=r3>>r2			

a)

s1:r1 h1:r3 p0:r2

r1=r1>>1	r2=load[A+r3]	r3=r3+1	store[&p0] = r2
		store[&t4] = r3	
r3=load[X]	r2=r1+r2		
r1=r2-1			
r1=r1-r3	r2=load[&p0]	r3=load[&t4]	
r3=r3>>r2			

b)

Figure 5: Utilizing Spill Code

scanning the code. Fig. 2 shows the register interference graph.

Fig. 3 shows the result of a possible register allocation, assuming that four registers are available.

Coloring a register interference graph produces a maapping of physical registers to symbolic registers. It works in order to obtain the lowest number of symbolic registers that should be spilled. On scalar processors, decreasing the ammount of spill code improves the performance of computing. Therefore, register allocation by coloring is very effective for producing better code for scalar processors, and has contributed to speedups of the execution of programs. However, taking account of instruction-level parallel processors, it cannot fully utilize the parallelism in a processor. We can demonstrate these three problems in conventional register allocators using a register interference graph.

1. Loss of Parallelism Due to Generation of Anti-dependence

Fig. 4a) shows a data dependence DAG of the target code in Fig 1b). Fig. 4b) shows a DAG after the allocation of registers as shown in Fig. 3 and the addition of edges expressing anti-dependence. Assuming the delay of load instructions to be 2 cycles and the delay of the others to be 1 cycle, the code shown in Fig. 4a) can be executed in 5 cycles on an instruction-level parallel processor that has sufficient ALUs. However, the code after register allocation in Fig. 4b) needs 2 more cycles, because of the generation of anti-dependence. The generation of anti-dependence is inevitable, but it is important to consider ways of generating anti-dependence without losing the parallelism in a program.

2. No Consideration of Spill Code That Gives Higher Parallelism

Fig. 5 shows the result of allocating 3 registers and scheduling instructions to obtain the minimum execution time. Fig. 5a) shows that 8 cycles are needed and that 1 symbolic register is spilled out. However, the code shown in Fig. 5b) has 2 symbolic registers that are spilled and the necessary execution time is 7 cycles. This is because spill code can be simultaneously executed with other instructions. This example indicates that there are some cases in which optimizations that take account of parallel execution of spill code and other instructions are needed when a program is executed on an instruction-level parallel processor.

3. No Consideration of Reference Differences of Symbolic Registers in Different Contexts

If the same symbolic register is referred to in several different places, these references have different importance. For example, "h1" in Fig. 4 is referred to by t3=load[X+h1] and t4=h1+1. Conventional register allocators do not take account of the difference between the reference to h1 in t3=load[X+h1] and the reference in t4=h1+1. However, Fig. 4 obviously shows that the reference to h1 in t3=load[X+h1] is more important than that in t4=h1+1, because the execution of t3=load[X+h1] is more critical to the total execution time of the program. Therefore, a register

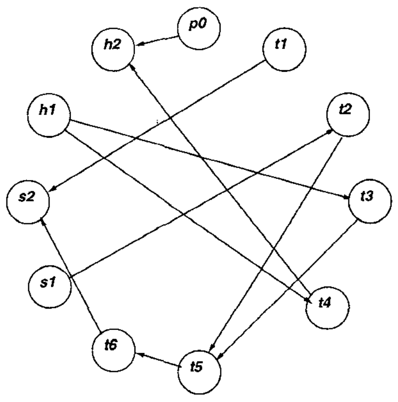


Figure 6: DAG

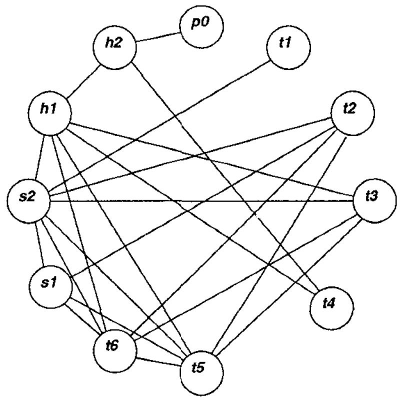


Figure 7: Transitive Closure

should be preferentially allocated to the part of the lifespan of h1 from the beginning to the time when t3=load[X+h1] is executed. This kind of allocation gives the following technique for obtaining better code:

1. Allocate a register to h1.
2. Spill out h1 after executing t3=load[X+h1].
3. Spill in h1 before executing t4=h1+1.

3 Recent Approaches

To solve the problem of generating anti-dependence, several approaches have been suggested.

The methods of Pinter [8] and Norris and Pollock [9] consider the parallelism in a program in such a way that not only the interferences between symbolic registers obtained by scanning code lexically but also possible interferences that may appear when instructions are reordered by the code scheduler are used.

This is performed as follows. First, make a DAG from a source code (Fig. 6).

Next, make the transitive closure of the graph, and eliminate the edge directions (Fig. 7).

In this graph, there is no flow-dependence between nodes not connected by edges. Therefore, allocating the same register to symbolic registers represented by these nodes may cause the generation of anti-dependence that results in the loss of parallelism in a program.

Finally, add edges between the nodes in a register interference graph that are not connected by edges in the non-directed transitive closure of a DAG (Fig. 8). Pinter called this graph a parallelized register interference graph [8]. A similar graph is obtained by the method of Norris and Pollock. Register allocation based on this kind of graph allows code schedulers to extract the parallelism in a program despite the generation of anti-dependence.

Theoretically, register allocators that use parallelized register interference graph coloring give a better code if a very large number of registers are available. However, a parallelized register interference

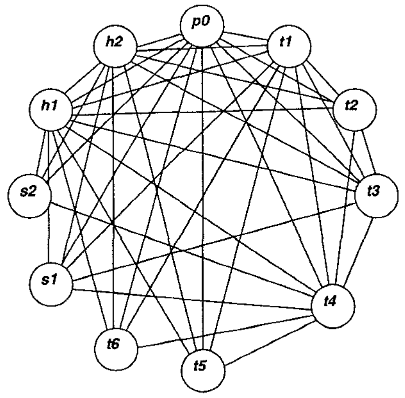


Figure 8: Interference Graph Considering Parallelism

graph contains too many edges, and therefore produces unnecessary spill code and impairs the code if the number of registers is not very large.

Pinter, and also Norris and Pollock, attempted to avoid this problem by removing edges from a parallelized register interference graph. However, removing edges after adding them makes allocation more complicated and results in futile processes. Moreover, removing processes does not ensure the retention of parallelism in the graph. The method of Norris and Pollock can possibly maintain the parallelism, but how it does so was not mentioned.

4 Our Approach

This section describes a new approach to register allocation that is very different from existing approaches based on register interference graph coloring. We introduce a data structure for register allocation in which the parallelism in a program and data flows over symbolic registers can be well described. The data structure is called a **Register Existence Graph**.

Section 4.1 describes the method of constructing a

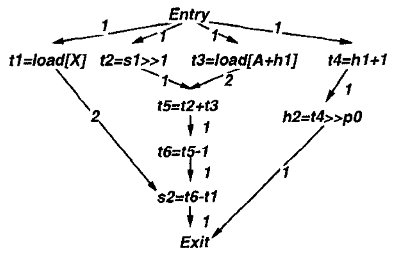


Figure 9: Weighed Data Dependence DAG

register existence graph. Section 4.2 explains how the interference among symbolic registers is expressed in a register existence graph. Then, section 4.3 presents a register allocation model using a register existence graph. After a description of the method of handling spill code in section 4.4, the specifications of the register allocation algorithm are given in section 4.5.

4.1 Making a Register Existence Graph

This section describes the construction of a register existence graph. First, a data dependence DAG is made from source code in which nodes express instructions and edges express data dependence. Then, each edge is weighted according to the delay needed to execute the instructions that the nodes connected by the edge represent. Fig. 9 shows a DAG of Fig. 1b). For example, there is a delay of 2 cycles between instruction t3=load[A+h1] and instruction t5=t2+t3, so the weight of the edge is 2.

In this DAG, the distance between nodes is defined as the largest total weight of nodes in the path. The path that has the largest distance is called the critical path. When the code expressed by a DAG is executed on processors that have a sufficient number of ALUs, it needs at least as many cycles as the critical path, so the execution time of a program should be determined by the length of the critical path [10]. Using this length as an index for optimizing elements such as register allocator and code scheduler is obviously effective.

Next, make a register existence graph from the obtained DAG. Except in some cases, in register existence graphs, nodes represent symbolic registers and edges represent the use of data in symbolic registers. For example, if data in a symbolic register *b* are defined by data in a symbolic register a, add an edge from *a* to *b*. Each node in a register existence graph is weighted by the delay needed to generate data in the symbolic register that the node represents.

The method for obtaining a register existence graph from a DAG is as follows:

- Put Entry and Exit as the source and the sink of the register existence graph, respectively.

- Analyze symbolic registers that are live in the entrance of the code, put nodes that represent the symbolic registers, and then put edges from Entry to the nodes.

- Analyze symbolic registers that are live in the exit of the code, put nodes that represent the symbolic registers, and then put edges from the nodes to Exit.

- For each instruction in the DAG, apply the transformation shown in Fig. 10.

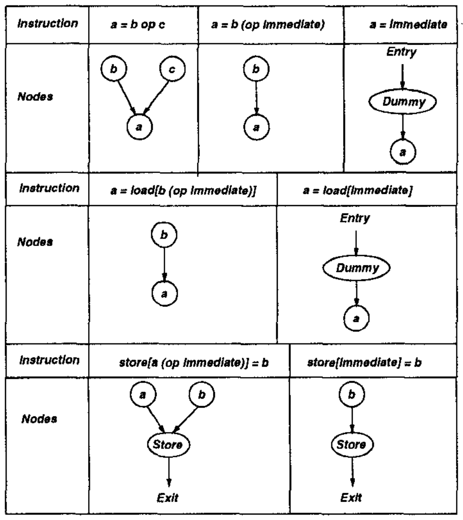


Figure 10: Making Nodes in a Register Existence Graph

Fig. 11 shows a register existence graph made by these processes.

4.2 How a Register Existence Graph Expresses Interference among Symbolic Registers

Consider the register existence graph in Fig. 12, in which two thick lines are drawn.

These lines divide the graph into two planes: one includes Entry and the other includes Exit. A line of this kind indicates the following:

- A result of code scheduling exists in which the symbolic registers represented by nodes crossed by the line are live at the same time.

Fig. 12b) shows the result of code scheduling. Symbolic registers t2, h1 and p0 are crossed by line1 and are simultaneously live between cycles n and n+1. In addition, symbolic registers t1, t5, t4 and p0 are crossed by line2 and are simultaneously live between cycles n+3 and n+4.

We call these lines **con-time lines** in the sense that the line indicates the time at which the symbolic registers in the nodes that the line crosses are live simultaneously. The existence of the possibility of drawing

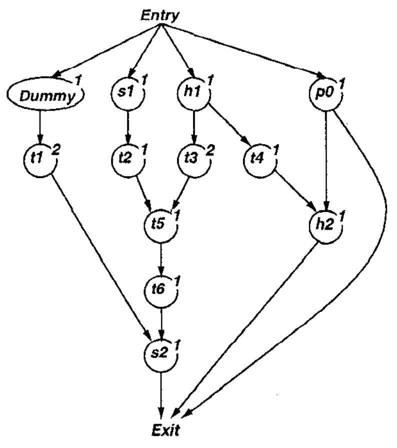
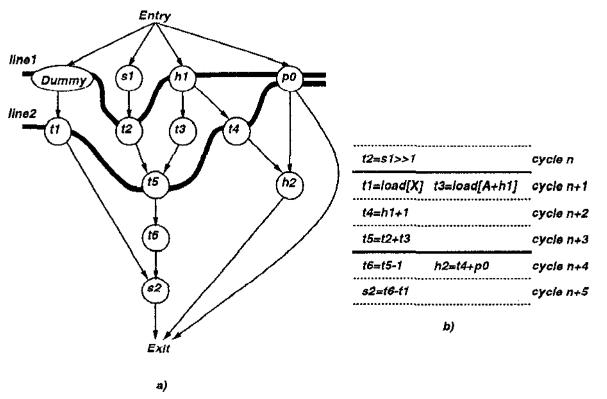


Figure 11: Register Existence Graph



```
t2=s1>>1                         cycle n
t1=load[X]   t3=load[A+h1]       cycle n+1
t4=h1+1                          cycle n+2
t5=t2+t3                         cycle n+3
t6=t5-1      h2=t4+p0            cycle n+4
s2=t6-t1                         cycle n+5
```

Figure 12: Cutting a Register Existence Graph

such a line means that there exists a possible interference among the symbolic registers in the nodes that the line crosses, so does a result of code scheduling that produces such interference.

4.3 Register Allocation and Code Scheduling in a Register Existence Graph

As mentioned in the previous section, the drawing of a con-time line is related to the interference among symbolic registers and the results of code scheduling. Therefore, drawing con-time lines as follows on a register existence graph somewhat restricts the allocation of registers and scheduling of instructions.

- Con-time lines never cross each other.
- For each node, one or more con-time lines representing weights equivalent to or greater than that of the node are drawn horizontally.

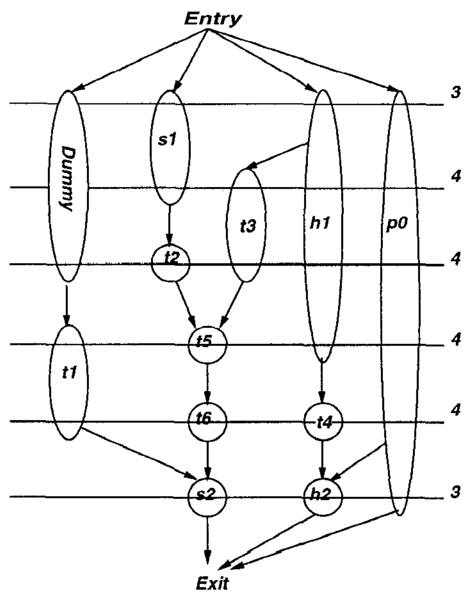


Figure 13: Leveling a Register Existence Graph

We call this process **leveling**. Leveling fixes the overlaps of the lifespans of symbolic registers for the entire code, in other words, it determines which symbolic registers interfere with each other at the same time. Thus, leveling means performing a part of the register allocation and a part of the code scheduling.

Fig. 13 shows an example of leveling.

Here, the number put on each line indicates the number of symbolic registers the line crosses. This indicates the number of symbolic registers that are live simultaneously. Therefore, we call the number "the degree of interference of symbolic registers." Leveling determines the degrees of interference of symbolic registers for each con-time line. The maximum degree means the smallest number of physical registers the code needs after leveling without spilling. In this example, 4 registers can be mapped to the symbolic registers without spilling.

4.4 Handling Spill Code in a Register Existence Graph

This section describes the method of spilling out and in symbolic registers on a register existence graph. The symbolic registers are spilled out and in by inserting nodes that express spilled symbolic registers into the register existence graph. These nodes are called spill nodes, and do not produce an increase in the degree of interference among symbolic registers, even if a con-time line goes through the nodes.

Spill nodes are inserted as shown in Fig. 14.

Fig. 14 a) shows an example in which a temporal register used in an intermediate code is spilled out and in. The lifespan of the temporal register is within the code expressed by the register existence graph, so the register is not live at either the top or the bottom of

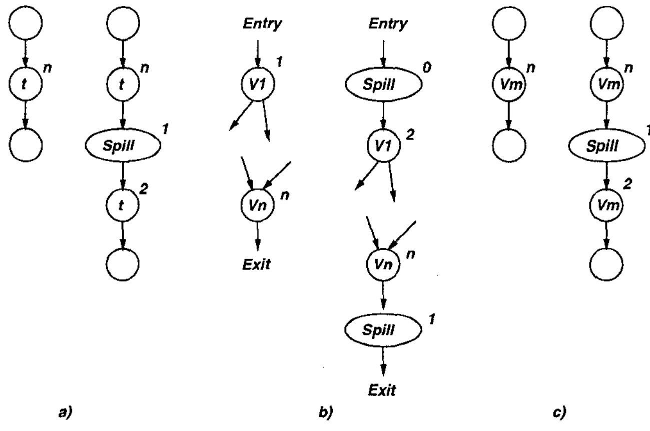


Figure 14: Inserting Spill Nodes

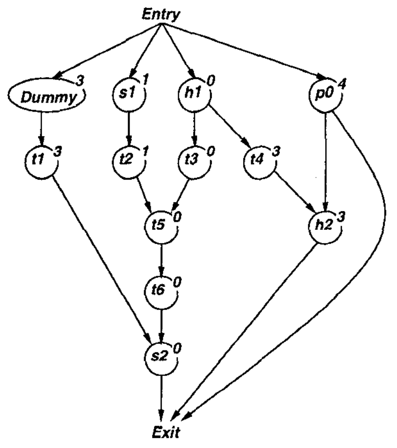


Figure 15: Slackness of Nodes

the code. In this case, after duplication of the node that expresses the symbolic register being spilled, a spill node is inserted between the duplicated nodes. Thus, the degree of interference is decreased by making the con-time lines that used to go through the temporal register go through the spill node.

Figs. 14 b) and c) are examples in which a variable (transformed into SSA form) is spilled out and in. Its lifespan is not within the code expressed by a register existence graph, so it is live at both the top and bottom of the code. Here, we have two cases of inserting a spill node. One is a case in which data in the variable that are live over the iteration, i.e., live at both the top and bottom of the code, are spilled. The other is a case in which other data in the variable are spilled. For a variable "V" transformed as V1, V2 ..., Vn, the datum defined first is V1 and that defined last is Vn, and edges are put from Entry to V1, and from Vn to Exit. In this case, V1 and Vn are live over the iteration. V1 and Vn are spilled out and in as shown in Fig. 14 b). Others are spilled out and in as shown in Fig. 14 c).

All nodes inserted are weighted as in Fig. 14.

4.5 Required Specifications of the Register Allocation Algorithm

The specification of register allocation with leveling a register existence graph is as follows:

- Leveling and the insertion of spill nodes are performed to ensure that every node is crossed by as many or more lines than its weighted value, and to ensure that the degree of interference on every con-time line does not exceed the number of available registers.

An algorithm that satisfies this specification will be described in Section 5.

4.6 Information for Leveling Derived from a Register Existence Graph

Number of con-time lines

We have shown that the distance of the critical path of a DAG can be considered as the execution time of a program, and its utilization as an index of optimization techniques using DAG is effective.

Since a register existence graph is a type of transformation of a DAG, it includes some information that corresponds to the distance of the critical path of a DAG. In a register existence graph, this information is the number of con-time lines that satisfy the restriction mentioned in Section 4.3. Therefore, since this number can be considerd as the execution time of a program, an algorithm using a register existence graph should work in such a way that it does not increase this number.

Slackness of a node

The slackness of a node is defined as the maximum weight that can be added to the node without an increase in the number of con-time lines. Taking Fig. 15 as an example, the addition of one weight to node s1, whose weight is 1, does not increase the number of con-time lines. Therefore, the slackness of the node is 1. Fig. 15 shows the slackness of each node in Fig. 11. This information is useful for minimizing the increase in the number of con-time lines, which is achieved by allocating hardware resources preferentially to less slack nodes and spilling symbolic registers expressed by slacker nodes.

5 An Example of an Algorithm

This section presents an algorithm for leveling a register existence graph. The algorithm works at the order of n^2, where n is the number of nodes.

```
Nodes = choose-successors(Entry);
Level = 1;

while(Nodes != {Exit}){
  if(the degree of interference exceeds
     the number of registers){
```

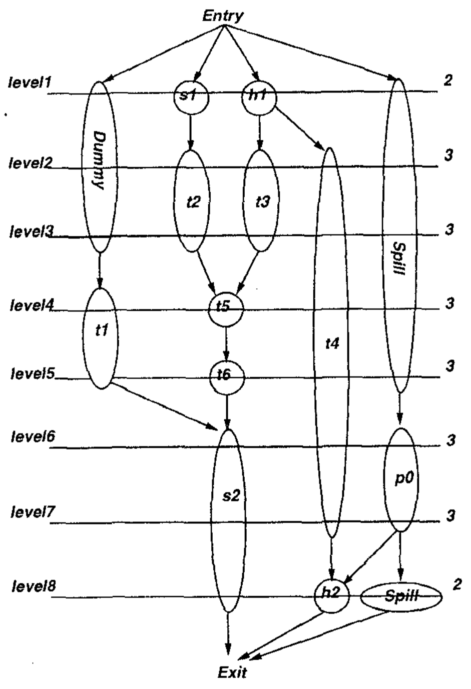


Figure 16: Result of Leveling

```
    Nodes = sort-nodes-by-slackness(Nodes);
    ChosenNodes = choose-as-many-nodes-as-
                  the-number-of-registers-
                  from-the-top(Nodes);
    finished[Level++] = ChosenNodes;
    spill nodes except ChosenNodes
  }else{
    ChosenNodes = Nodes;
  }
  Remains = Null;
  foreach(Node in ChosenNodes){
    Node.weight--;
    if(Node.weight > 0)
      Remains = Remains + Node;
  }
  Nodes = Null;
  foreach(Node in ChosenNodes){
    Nodes = choose-successors(Node)
    if (exists Successor1 and Successor2
        in Nodes such that
        Successor1 is the successor
        of Successor2)){
      Nodes = Nodes + Node;
      Nodes = Nodes - Successor1;
    }
  }
  Nodes = Nodes + Remains;
}
```

The result of leveling the code is shown in Fig. 16.

6 Evaluation

This section compares our approach using a register existence graph, an ordinary register allocation using a register interference graph, and a register allocation using the parallelized interference graph described in section 3. We selected some programs from the Stanford Benchmark for evaluation, and the lengths of their critical paths after register allocation using each method and code scheduling are listed in Tables 1 and 2. Our target machine is a VLIW that can execute fixed-point instructions in 1 cycle, and floating-point and load/store instructions in 2 cycles.

Table 1 shows a comparison of critical path lengths when the register pressure is high (8 registers available). Table 2 shows a comparison of critical path lengths when the register pressure is low (32 registers available).

If the register pressure is high, our method gives better results than the other two methods. Consequently, it performs 2.01 times better on average than the method using a parallelized interference graph, and 2.17 times better on average than the method using an interference graph. Table 1 shows that the ratio of performance improvement with the method using a parallelized interference graph is higher when the number of available ALUs is larger. This is because that method increases the edges of the interference among symbolic registers so much that it is difficult to determine which edges should be deleted. Table 1 also shows that the ratio of performance improvement with the method using an interference graph is higher when the number of available ALUs is smaller. This is because that method increases the number of instructions by inserting spill code to decrease the interference among symbolic registers.

If the register pressure is low, our method gives almost the same result as that using an interference graph because, spill code is rarely generated. As a result, it performs 1.28 times better on average than the method using a parallelized interference graph, and 1.02 times better on average than the method using an interference graph. The method using a parallelized interference graph gives the worst results, because of an increase in the number of edges of the interference. In the method using a parallelized interference graph, it is importntant to improve the method of deleting the added edges if the register pressure is not very high.

In our method, the interference among symbolic registers changes during leveling. This method of handling the interference, using a register existence graph, allows us to consider the parallelism among instructions and to handle the interference according to the number of registers available.

7 Conclusions

We have introduced a register existence graph that can express the interference among symbolic registers and the parallelism among instructions in a program. We have also shown that leveling a register existence graph realizes the generation of anti-dependence and spill code taking account of the parallelism in a program, which existing methods rarely do. We are now

Table 1: Comparison When Register Pressure Is High (8 Registers Available)

	Our method					Method using a parallelized interference graph					Method using an interference graph				
Number of ALUs	1	2	4	8	∞	1	2	4	8	∞	1	2	4	8	∞
Bubble	39	22	14	11	11	53	37	37	37	37	59	32	19	18	18
Exptab	67	37	22	18	18	105	91	91	91	91	162	49	43	24	24
Fit	86	63	62	62	62	88	73	70	70	70	249	150	148	148	148
Initarr	79	41	32	32	32	106	96	96	96	96	164	84	56	56	56
Initmat	89	51	29	28	28	150	110	110	110	110	310	77	74	74	74
Permute	33	19	15	15	15	33	25	24	24	24	74	40	29	29	29
Qsort	40	26	19	18	18	46	34	34	34	34	77	45	31	28	28
Remove	81	46	44	44	44	99	83	83	83	83	243	106	104	104	104
Try	109	62	44	42	41	163	137	137	137	137	290	86	85	79	79

Table 2: Comparison When Register Pressure Is Low (32 Registers Available)

	Our method					Method using a parallelized interference graph					Method using an interference graph				
Number of ALUs	1	2	4	8	∞	1	2	4	8	∞	1	2	4	8	∞
Bubble	39	22	14	11	11	39	26	24	23	23	39	22	14	11	11
Exptab	62	35	18	11	10	62	38	33	33	33	62	35	18	11	11
Fit	86	63	62	62	62	86	71	69	69	69	86	63	62	62	62
Initarr	75	39	32	32	32	75	51	49	49	49	75	39	32	32	32
Initmat	89	51	29	28	28	96	62	58	58	58	96	51	29	28	28
Permute	33	19	15	15	15	33	25	24	24	24	33	19	15	15	15
Qsort	38	25	19	17	17	38	25	23	23	23	38	24	19	17	17
Remove	81	46	44	44	44	81	59	57	57	57	81	46	44	44	44
Try	76	61	44	39	39	105	69	66	66	66	105	61	44	39	39

considering the ways of improving the leveling algorithm and allowing cooperation between a spill code generator and a code scheduler.

References

[1] G.J.Chaitin, M.A.Auslander, A.K.Chandra, J.Cocke, M.E.Hopplins and P.W.Markstein, "Register Allocation via Coloring," Computer Languages 6 (1981), pp.47-57.

[2] G.J.Chaitin, "Register Allocation & Spilling via Graph Coloring," Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction (Jun. 1982), pp.98-105.

[3] D.Callahan and B.Koblenz, "Register Allocation via Hierarchical Graph Coloring," Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (Jun. 1991), pp.192-203.

[4] A.Koseki, H.Komatsu and Y.Fukazawa "A Register Allocation Technique Using Guarded PDG," Proceedings of the International Conference on Supercomputing (May 1996), pp.270-277.

[5] J.R.Goodman and W.C.Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," International Conference on Supercomputing (Jul. 1988), pp.442-452.

[6] J.R.Ellis, "Bulldog: A Compiler for VLIW Architectures," The MIT Press (1985).

[7] 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 (Jan. 1989), pp.25-35.

[8] S.S.Pinter, "Register Allocation with Instruction Scheduling: A New Approach," Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation (1993), pp.248-257.

[9] C.Norris and L.L.Pollock, "A Scheduler-Sensitive Global Register Allocation," Proceedings of the ACM SIGPLAN '93 Conference on Supercomputing (1993), pp.804-813.

[10] E.B. Fernandez and B. Bussel, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," IEEE Trans. Computers, Vol.C22, No.8, pp.745-751 (1973).