December 1991

UILU-ENG-91-2252
CRHC-91-31

Center for Reliable and High-Performance Computing

COMPILER-ASSISTED MULTIPLE INSTRUCTION RETRY

Chung-Chi Jim Li, Shyh-Kwei Chen
W. Kent Fuchs, and **Wen-Mei W. Hwu**

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CRHC-91-31	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois — 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014- 90-J-1270 and N00014-91-J-1283
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Compiler-assisted Multiple Instruction Retry

12. PERSONAL AUTHOR(S) Li, Chung-Chi Jim; Chen, S-K., Fuchs, W. Kent; Hwu, Wen-Mei

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	91-11-25	31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	rollback recovery, fault-tolerant computing, compilers

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This paper describes a compiler-assisted approach to providing multiple instruction rollback capability for general purpose registers. The objective is achieved by having the compiler remove all forms of N-instruction anti-dependencies. Pseudo register anti-dependencies are removed by loop protection, node splitting, and loop expansion techniques; machine register anti-dependencies are prevented by introducing anti-dependency constraints in the interference graph used by the register allocator. To support separate comilation, inter-procedural anti-dependency constraints are added to the code generator to guarantee the termination of machine register anti-dependencies across procedure boundaries. The algorithms are implemented in the IMPACT-C compiler and experiements are performed to evaluate the effectiveness of this approach.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

SUBMITTED TO: IEEE TRANSACTIONS ON COMPUTERS

Compiler-Assisted Multiple Instruction Retry

Chung-Chi Jim Li, *Shyh-Kwei Chen*, *W. Kent Fuchs*, and *Wen-Mei W. Hwu*

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, IL 61801

Correspondent: W. Kent Fuchs
Tel: (217)333-9731
FAX: (217)244-1764
Email: fuchs@crhc.uiuc.edu

Abstract

This paper describes a compiler-assisted approach to providing multiple instruction rollback capability for general purpose registers. The objective is achieved by having the compiler remove all forms of N-instruction anti-dependencies. *Pseudo register anti-dependencies* are removed by *loop protection*, *node splitting*, and *loop expansion* techniques; *machine register anti-dependencies* are prevented by introducing *anti-dependency constraints* in the interference graph used by the register allocator. To support separate compilation, *inter-procedural anti-dependency constraints* are added to the code generator to guarantee the termination of machine register anti-dependencies across procedure boundaries. The algorithms are implemented in the IMPACT C compiler and experiments are performed to evaluate the effectiveness of this approach.

Index Terms: rollback recovery, fault-tolerant computing, compilers

This research was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Grant N00014-90-J-1270, and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283.

I. Introduction

A. Multiple Instruction Retry

The capability of retrying a few instructions is desirable in situations requiring rapid recovery from transient processor failures. This involves preserving the state of memory locations and CPU registers. If all errors can be detected immediately, single instruction retry is sufficient. This has been successfully implemented on commercial machines, such as IBM 4341 processor [1] and VAX 8600 processor [2]. If the target position of the rollback is an established checkpoint rather than a point within a sliding window [3], the state of memory locations can be preserved by copying the old values of all updated locations to a push-down stack, and the state of CPU registers can be preserved by copying to a backup register file. When an error occurs, the contents of the backup register file is copied to the working register file and the contents of the push-down stack is applied to the memory system in reverse order. This approach is implemented in the IBM 3081 processor with a checkpoint interval of 10-20 instructions [4, 5].

If the target position of the rollback is anywhere within a sliding window, the general approach is to delay the effect of write operations by N instructions. The delayed writes to main memory can be achieved by providing a delayed write buffer [3] or by modifying the cache coherence protocol [6]; the delayed writes to CPU registers are usually achieved by replicating the entire register file [7] or by providing another delayed write buffer [3]. The basic assumption is that the usage pattern can not be predicted. However, if the program is written in a high level language, the usage of the general purpose registers is controlled by the compiler. This paper describes the use of compiler technology to preserve the state of CPU registers within a sliding window in order to facilitate multiple instruction retry.

Due to environmental variables, it may be difficult to determine the optimal value for N until the system is in operation. Also, a new device may have a higher N than originally expected. Therefore, it may be desirable to have a recovery mechanism that can adapt to different N after the system is installed.

Our approach is to let N be a compile-time parameter. The resulting executable code will not destroy the content of a register until it is voided for more than N instructions. This property is obtained by prohibiting all anti-dependencies [8] within N instructions.

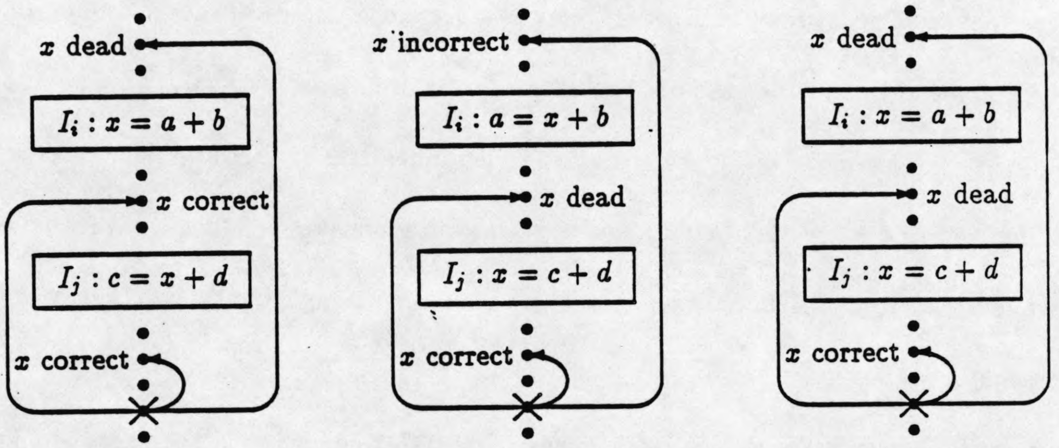
B. Error Model

To clarify which errors are considered in this multiple instruction retry scheme, we have made the following assumptions:

1. CPU errors and memory errors are detected before the register contents can be contaminated. Otherwise, incorrectly fetched instructions can nullify any flow information recognized by the compiler.
2. The maximum error detection latency is N instructions.
3. There is an external device or a buffer inside the CPU that records the executed instructions with capacity $C \geq N$. This is to facilitate the rollback of the program counter.
4. There is a delayed write buffer [3] for the memory system with capacity $C \geq N$. Otherwise, the memory system can not rollback to a state consistent with CPU registers.
5. The CPU state can be restored by loading the correct contents of the register file and the program counter.

C. Anti-Dependencies

There are generally three types of dependencies between instructions: 1) flow dependency (read after write), 2) anti-dependency (write after read), and 3) output dependency(write after write) [8]. The flow dependency and the output dependency do not impair rollback capability, but the anti-dependency does. These situations are illustrated by the simple sequential code in Figure 1.



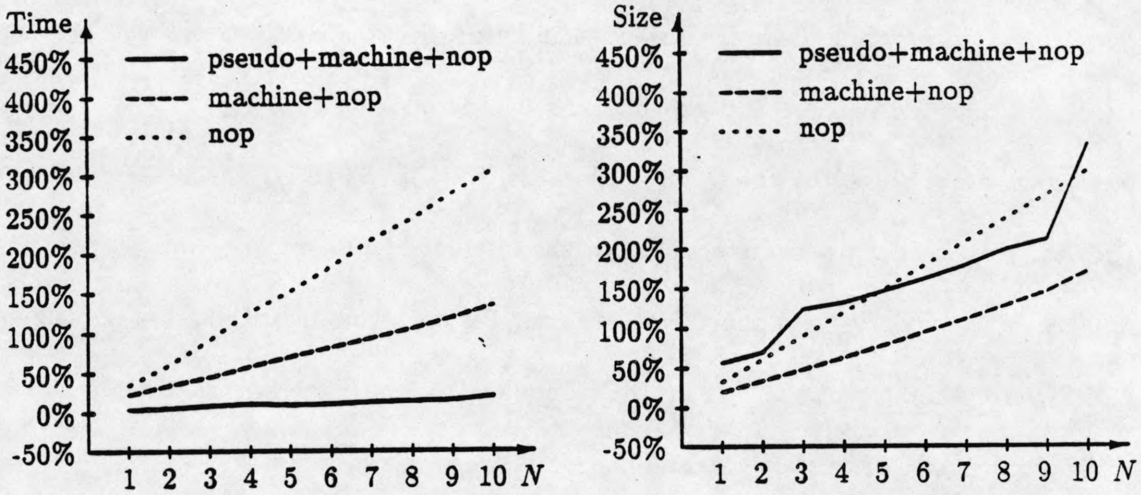
(a) flow dependency $I_i \delta_x^f I_j$ (b) anti-dependency $I_i \delta_x^a I_j$ (c) output dependency $I_i \delta_x^o I_j$

Figure 1. Types of dependencies and their impact on rollback capability

Assume that an error requiring multiple instruction rollback is detected at the cross mark and there are no other instructions containing variable x except those shown in the figure. In Figure 1(a), there is a flow dependency from instruction I_i to I_j based on variable x (denoted by $I_i \delta_x^f I_j$). If the program counter is rolled back to a point before the execution of instruction I_i, the program will produce the correct result since variable x is dead and will be reloaded in instruction I_i. If the program counter is rolled back to a point after the execution of I_i, the program will also produce the correct result since x now contains the correct value. Similar arguments hold for the points after I_i in Figure 1(b) and all points in Figure 1(c). However, for the points before I_i in Figure 1(b), x now contains the incorrect value $c + d$ rather than its expected value. Therefore, to achieve complete rollback capability, the anti-dependencies within N instructions must be prohibited.

Anti-dependencies come from two sources: 1) when the intermediate code generator assigns live values to *pseudo registers* (or symbolic registers) [9], and 2) when the register allocator assigns pseudo registers to machine registers. An example of the former case is the x variable in Figure 1(b). The intermediate code generator will assign a pseudo register, say t_k, to variable x and generate an anti-dependency. This type of anti-dependency is a *pseudo register anti-dependency*. The latter case may introduce anti-dependencies on machine registers even when two values reside in different pseudo registers. For example, in Figure 1(a), if the pseudo register t_m for variable a and the pseudo register t_n for variable c are assigned to the same machine register r_k, then an anti-dependency occurs between instruction I_i and I_j on r_k. This type of anti-dependency is a *machine register anti-dependency*.

One simple approach to resolve both types of anti-dependencies is to insert enough nops (or other redundant operations that will not change the state of the register file) between the use and definition that cause the anti-dependency. However, the execution time will be increased dramatically. Figure 2 shows the effectiveness of applying compiler techniques to resolve the pseudo register and machine register anti-dependencies compared with the simple nop insertion approach. The program under test is the 12 queen problem which is one of the benchmarks described in Section V. Figure 2(a) shows the run time overhead compared with the original run time of 17.0 seconds on a DECstation 3100. The x-axis is the intended anti-dependency distance N. The y-axis is the percentage overhead. The dotted line is for the version that utilizes only nop insertion. The dashed line is for the version that resolves machine register anti-dependencies and then applies nop insertion to resolve the remaining anti-dependencies. The solid line is for the version that resolves both pseudo register and machine register anti-dependencies and then applies nop insertion to resolve the inter-procedural anti-dependencies. From the figure, it is clear that applying compiler techniques to resolve anti-dependencies can significantly reduce the run time overhead compared with just inserting nops. The size overhead, measured by the number of machine instructions, is shown in Figure 2(b). It is not improved or, in some cases, it is even worse than just inserting nops. However, this is of less importance unless the cache miss problem becomes serious for very



(a) Run time overhead (b) Size overhead

Figure 2. Effectiveness of applying compiler techniques on the QUEEN benchmark

large programs. Our primary goal is to minimize the run time overhead.

D. Approach

Compiler techniques have been used to assist error recovery at the process level. For example, checkpoint decision [10] and multi-processor state compression [11] can be achieved by having the compiler insert code in the program. Also, algorithm-based error detection [12] can be assisted by having the compiler analyze the source code. This paper is different in that it introduces a coherent method to provide a particular property of programs for purposes of error recovery. Most of the compiler techniques used in this paper, such as *node splitting* and *loop expansion*, are variations of well known techniques that have been applied for other purposes [9, 13]. Our contribution is the formulation of the register state preservation problem as an anti-dependency removal problem, the provision of a practical solution that uses well-developed compiler techniques, and an implementation with experimental results.

Section II describes the removal of N-instruction pseudo register anti-dependencies by *loop* *protection*, *node splitting*, and *loop expansion* techniques. Section III describes the prevention of N-instruction machine register anti-dependencies by introducing *anti-dependency constraints* in the interference graph used by the register allocator [14, 15]. Since the machine register anti-dependency can exist across procedure boundaries, the *inter-procedural anti-dependency constraints* are introduced in Section IV to support separate compilation. The algorithms are implemented in the MIPS code generator of the IMPACT C compiler [16] and experiments are conducted to evaluate the performance of this approach. The results are reported in Section V.

II. Pseudo Register Anti-Dependencies

A. The Problem

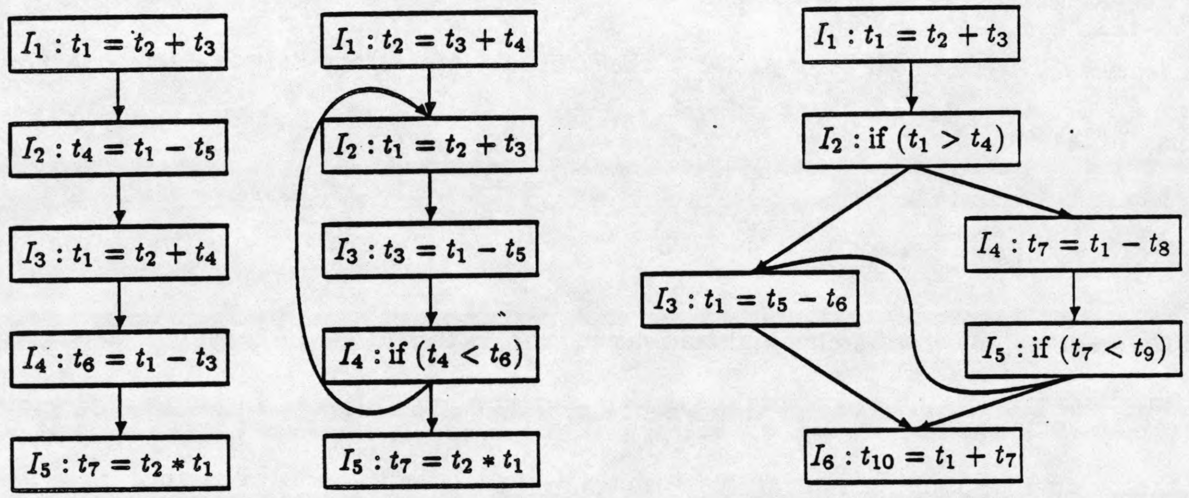
The input we consider is a flow graph $G(V, E)$ where V is the set of nodes and E the set of edges. Each node $I_i \in V$ represents an instruction. If there is a direct control flow from instruction I_i to instruction I_j, then there is an edge $(I_i, I_j) \in E$. Define the *distance* $d(I_i, I_j)$ to be the smallest number of instructions on any path from I_i to I_j. The distance from a node to itself is 0. An instruction I_i is called *self-anti-dependent* if $I_i \delta_x^a I_i$, e.g., $I_i : x = x + a$. The objective is to remove all pseudo register anti-dependencies within distance N (i.e., $I_i \delta_x^a I_j$ and $d(I_i, I_j) \leq N$) while still maintaining the semantics of the code.

The pseudo register anti-dependencies can be resolved by code transformation, pre-pass code scheduling [17], or a combination of both. The former approach renames pseudo registers but maintains the relative order of instructions; the latter approach changes the order of instructions but does not rename pseudo registers. Both approaches require the insertion of extra code. This paper utilizes the code transformation approach. After the transformation, only register allocation and code emission as described in the next section are allowed; otherwise, the N-instruction anti-dependencies may reemerge if other phases of the compiler, such as loop optimization, change the sequence of the code.

B. Resolvability

The basic approach to resolving an anti-dependency is to rename the pseudo registers. For example, in Figure 3(a), there is an anti-dependency $I_2\delta^a_{t_1}I_3$ that needs to be resolved if $N = 3$. This can be done by simply renaming the t_1 in I_3, I_4, and I_5 to t_8 since the value in t_1 is dead at the entry of I_3. However, some flow graphs do not allow proper renaming. For example, in Figure 3(b), the anti-dependency $I_3\delta^a_{t_1}I_2$ can not be resolved since any renaming of the t_1 in I_2 will result in a renaming of t_1 in I_3 to the same new pseudo register in order to maintain the semantics. Similarly, $I_2\delta^a_{t_3}I_3$ can not be resolved either. This problem can occur even in acyclic graphs. For example, in Figure 3(c), the anti-dependency $I_4\delta^a_{t_1}I_3$ can not be resolved since any renaming of t_1 in I_3 will result in the same renaming of t_1 in I_6. If the t_1 in I_6 is renamed, so is the t_1 in I_1 and hence the t_1 in I_2 and I_4.

The problems presented in Figure 3(b) and 3(c) are formally described as follows. For each pseudo register x, initialize the set of symbols $Z_x = \phi$. If an instruction I_i defines x, put a symbol I_i^d in Z_x; if it uses x, put a symbol I_i^u in Z_x. Then define an equivalence relation $\equiv_x$ on Z_x as follows: if x is defined in I_j, x is used in I_i, and the definition of x in I_j belongs to the set of reaching definitions [9] of I_i (i.e., all definitions that can reach I_i without being redefined along the path), then we have $I_j^d \equiv_x I_i^u$. Naturally, the equivalence relation $\equiv_x$ is reflexive, symmetric, transitive, and can partition the set Z_x into disjoint subsets [18]. An anti-dependency $I_i\delta^a_x I_j$ is *unresolvable* if and only if $I_i^u \equiv_x I_j^d$ since the renaming of x in one instruction requires all occurrences of x in all



(a) resolvable $I_2 \delta^a_{t_1} I_3$ (b) unresolvable $I_3 \delta^a_{t_1} I_2$ (c) unresolvable $I_4 \delta^a_{t_1} I_3$

Figure 3. Resolvability of anti-dependencies

the other elements belonging to the same subset to be renamed to the same new pseudo register in order to maintain the correct semantics. This is exactly what happened in Figure 3(c). Since $I_1^d \equiv_{t_1} I_4^u$, $I_1^d \equiv_{t_1} I_6^u$ and $I_3^d \equiv_{t_1} I_6^u$, by symmetry and transitivity, we obtain $I_4^u \equiv_{t_1} I_3^d$. Therefore, the anti-dependency $I_4 \delta^a_{t_1} I_3$ is unresolvable.

To handle the unresolvable N-instruction anti-dependencies, we can transform the original code by the following two methods: 1) *node splitting*, and 2) *loop expansion*. The former breaks the $\equiv_x$ relation between the nodes; the latter effectively increases the distance between the two instructions that cause the anti-dependency. Before presenting the two methods, we need to describe the loop structure of the program that guides the application of node splitting and loop expansion. Also, we need to describe a preparation step called *loop protection* that inserts code in the program to prevent the loop structure from being destroyed by node splitting.

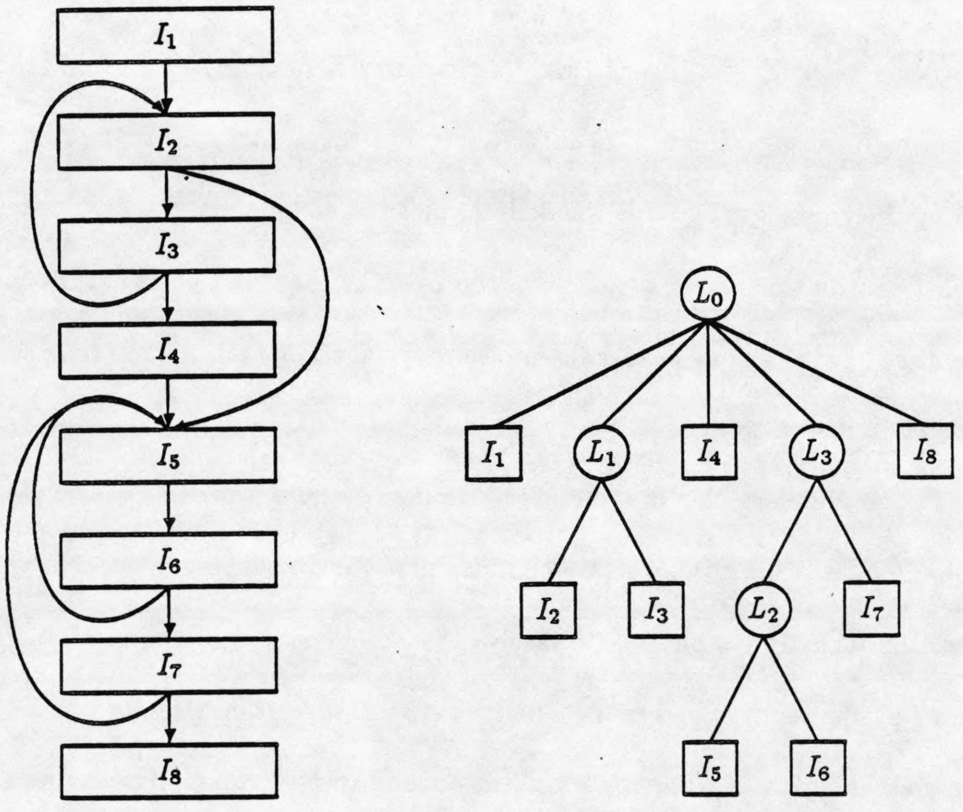
C. Loop Structure

A *backedge* is an edge (I_t, I_h) such that I_h *dominates* I_t (i.e., any path from the initial node of the program to I_t must go through I_h) [9]. I_h is called the *header* and I_t the *tail*. The *natural loop* induced by the backedge (I_t, I_h) is the node I_h plus the set of nodes that can reach I_t without going through I_h [9]. In this paper, we define a *loop* L_h to be the union of all natural loops induced by backedges that have the same header h. In other words, a loop has a single header and at least one backedge associated with it.

Most of the programs written in structured high level languages use nested iteration constructs such as the while loop. Therefore, we only consider programs with nested loops. If this is not the case, nop insertion can always be used to resolve the anti-dependencies. The relationship among the loops can be represented by a tree. The root of this tree stands for the entire flow graph, each interior node indicates a loop, and each leaf node is an instruction. For example, the tree in Figure 4(b) describes the loop structure of the flow graph in Figure 4(a). Instruction I_6 belongs to loop L_2 (the inner loop) which in turn belongs to loop L_3 (the outer loop). Obviously, loop L_3 belongs to the entire flow graph represented by the node L_0.

The *level* of an anti-dependency $I_i \delta_x^a I_j$ is the lowest level of the tree such that the paths causing $d(I_i, I_j) \leq N$ are entirely contained in a loop of that level. Our general approach is to successively reduce the levels of the N-instruction anti-dependencies until all of them occur at the top level and get resolved.

To determine the actual processing sequence of the loops, we define a relation $\prec$ on loops as follows: $L_i \prec L_j$ if the nodes in L_i is a proper subset of L_j. L_i is called an *inner loop* of L_j and L_j an *outer loop* of L_i. The relation $\prec$ is transitive and defines a partial ordering of the loops. The



(a) nested loops (b) loop structure

Figure 4. Program loop structure

loops can then be sorted into an array by a topological sort algorithm [19]. The generated array gives the processing sequence of the loops, which is not unique. However, as long as we process from the beginning to the end of the array, inner loops must be processed before their corresponding outer loops. For example, the processing sequence of the loops in Figure 4 could be L_1, L_2, L_3, L_0, or it could be L_2, L_1, L_3, L_0.

D. Loop Protection

An anti-dependency is to be resolved by node splitting or loop expansion. However, if the anti-dependency is to be resolved by node splitting and a loop header is one of the nodes to split, more loops and anti-dependencies will be generated which in turn requires more splitting. To prevent this abnormal situation, the loop should be *protected* relative to the pseudo register that causes the anti-dependency. Also, when we use loop expansion to resolve an anti-dependency, the targeted pseudo register may not be able to be renamed freely because it is used outside the current loop. This situation also requires the loop to be protected. The loop protection technique described in this subsection is actually a preparation step for node splitting and loop expansion.

If a pseudo register t_k causes an anti-dependency in a loop, the protection is done by renaming every t_k in the loop to a newly generated pseudo register t_i, and inserting nodes at one or more of the following positions:

1. Header position: right before the loop header and inside the loop, performing $t_i = t_k$.
2. Preheader position: right before the loop header but outside the loop, performing $t_i = t_k$.
3. Tail position: between each tail node and header, performing $t_k = t_i$.
4. Exit position: between each exit node and its target, performing $t_k = t_i$.

The nodes inserted at the header or preheader positions are called *save nodes* and the nodes inserted at the tail or exit positions are called *restore nodes*. The insertion is performed only if t_k is live at that point. For example, for loop L_1 in Figure 4(a), the header and preheader positions are both between I_1 and I_2, but the former is inside the loop receiving all incoming edges and the latter is outside the loop receiving only the incoming edge from I_1. The tail position is between I_3 and I_2, and the exit positions are between I_2 and I_5, and between I_3 and I_4.

To determine which positions require node insertion, the following definitions should first be understood. The *extended loop* $\tilde{L}_h(t_k)$ relative to pseudo register t_k consists of all nodes in L_h and all nodes I_i satisfying the following conditions: 1) $t_k \in$ live_in(I_i), where live_in(I_i) is the set of live variables at the entry point of I_i [9], 2) I_i has only one successor, and 3) I_i has only one predecessor I_j, and 4) I_j is in $\widetilde{L}_h$. For example, the extended loop of L_1 in Figure 4(a) consists of I_2, I_3, and I_4, if t_k is live at the entry point of I_4. If t_k is dead at every exit point of $\widetilde{L}_h(t_k)$, the extended loop is *safe*. The ***stripped graph*** $\overline{G}_h(\overline{V}_h, \overline{E}_h)$ is a subgraph of $G(V, E)$ such that $\overline{V}_h = V$ and $\overline{E}_h = E - \{\text{all backedges}\}$. The ***outer-stripped graph*** $\widehat{G}_h(\widehat{V}_h, \widehat{E}_h)$ is a subgraph of $G(V, E)$ such that $\widehat{V}_h = V$ and $\widehat{E}_h = E - \{$all backedges associated with loops that are outer loops of $L_h\}$. The ***hazard set*** $H(G)$ of a graph G consists of all pseudo registers t_k such that $I_i \delta^a_{t_k} I_j$, $d(I_i, I_j) < N$, and $I_i^u \equiv_{t_k} I_j^d$, using only nodes and edges in G. In other words, the hazard set is the set of pseudo registers that result in unresolvable anti-dependencies. The *exclusive hazard set* $X(G, L_h)$ of a graph G is the set $H(G)$ excluding all pseudo registers that do not result in anti-dependencies if the inner loops of L_h do not have anti-dependencies. The *split set* $S(G, t_k)$ of a graph G consists of all nodes in G that need to be split relative to pseudo register t_k using the node splitting algorithm to be described in the next subsection.

The loop protection algorithm is outlined in Figure 5. The outer most **if** statement checks the hazard set of $\widehat{G}_h$ rather than G because the anti-dependencies in outer loops should be resolved at the outer loop level rather than the current level. The first condition in the **for** loop is for the node splitting step to prevent the loop structure from being destroyed. The insertion is at the preheader and exit positions because all backedges have been disabled in $\overline{G}_h$ and the multiple definitions that result in the node splitting must come from outside the loop (the criteria for node splitting is described in next subsection). The second and third conditions are for the loop expansion step to provide the renaming capability after the loop is expanded. The insertion is at the header, tail, and exit positions because we want every iteration of the expanded loop to have unique set of save and restore nodes in order to rename the pseudo registers freely. The last **for** loop is used to protect the inner loops if the loop structure is to be destroyed due to the anti-dependencies of the current

```
if ($H(\widehat{G}_h) \neq \phi$) {
        for (each $t_k \in H(\widehat{G}_h)$) {
                if (the header node $I_h$ is in $S(\overline{G}_h, t_k)$)
                        protect $L_h$ by using the preheader and exit positions;
                else if ($\tilde{L}_h(t_k)$ is not safe)
                        protect $L_h$ by using the header, tail, and exit positions;
                else if (any tail node $I_t$ of $L_h$ is in $S(\widehat{G}_h, t_k)$)
                        protect $L_h$ by using the header, tail, and exit positions;
                for (each inner loop $L_{h'}$ of $L_h$)
                        if ($t_k$ is live at the entry of $L_{h'}$ and $t_k \in X(\overline{G}_h)$)
                                protect $L_{h'}$ by using the preheader and exit positions;
        }
}
```

Figure 5. The loop protection algorithm

loop. The insertion is at the preheader and exit positions due to the same reason for the first if statement in the for loop.

E. Node Splitting

Since the loop body must be made resolvable before the loop can be considered, we describe the node splitting technique before loop expansion. Various forms of the node splitting technique have been used in other parts of optimizing compilers [9]. In our approach, the purpose of node splitting is to break the $I_i^u \equiv_{t_k} I_j^d$ relation if t_k is in the current hazard set.

A node I_i will be in the split set $S(\overline{G}_h, t_k)$ if $t_k \in$ live_in(I_i) and there are more than one definition of t_k that can reach I_i. After the splitting, two copies of the originally connected nodes are connected if they are *compatible*, i.e., they have the same reaching definition of t_k. The algorithm is outlined in Figure 6. Note that the header will not be in the split set since the loop has been protected.

Figure 7 shows the resulting flow graph after the code segment in Figure 3(c) is processed by

```
if ($H(\overline{G}_h) \neq \phi$)
        for (each $t_k \in H(\overline{G}_h)$)
                if ($S(\overline{G}_h) \neq \phi$) {
                        split all nodes in $S(\overline{G}_h, t_k)$;
                        match the split nodes by a set of edges;
                }
rename the pseudo registers;
```

Figure 6. The node splitting algorithm

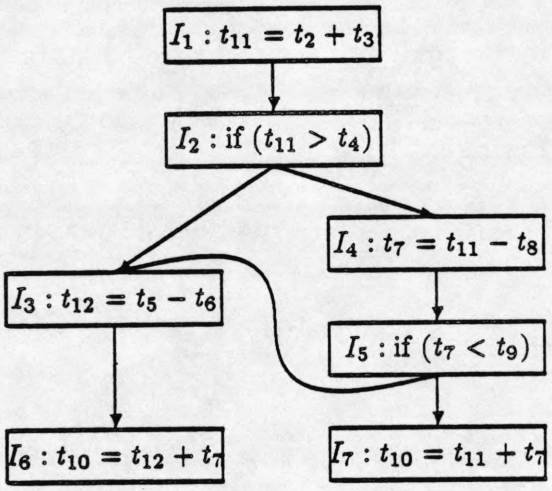

$I_1 : t_{11} = t_2 + t_3$
I_2 : if $(t_{11} > t_4)$
$I_4 : t_7 = t_{11} - t_8$
$I_3 : t_{12} = t_5 - t_6$
I_5 : if $(t_7 < t_9)$
$I_6 : t_{10} = t_{12} + t_7$
$I_7 : t_{10} = t_{11} + t_7$

Figure 7. Application of the node splitting algorithm

the node splitting algorithm relative to the pseudo register t_1. The use of t_1 in I_2 has a unique reaching definition from I_1; therefore, I_2 is not to be split. The situation is the same in node I_4. However, both definitions in I_1 and I_3 can reach I_6. Therefore, we have a non-trivial node splitting on I_6 resulting in the I_6 and I_7 in Figure 7. The final pseudo register renaming is done by changing the t_1 in I_1, I_2, I_4, and I_7 to t_{11}, and the t_1 in I_3 and I_6 to t_{12}.

The node splitting technique works because of the following three reasons: 1) all the N-instruction anti-dependencies in the inner loops have been resolved since we process the loops from inside out, 2) the live ranges of the variations of t_k (i.e., the definitions of t_k before the renaming) do not intersect, and 3) the definition always occurs before its use unless there is an unresolved inner loop, which is impossible because it contradicts the first condition. Since an anti-dependency requires a read before write, they must belong to different live ranges and can be renamed to different pseudo registers.

If there are no back edges outside the current loop body, i.e., at the root level of the loop structure, the anti-dependency can simply be resolved by removing all unnecessary save and restore nodes (usually, too many are generated by loop protection and node splitting). However, if there is a back edge outside this body, anti-dependencies may occur in the following cases: 1) between the use of a variation of t_k and its definition, going through the back edge, or 2) between the nodes at an upper level. The latter will eventually be resolved since we are working from inside out. The former is the subject of loop expansion.

F. Loop Expansion

Loop expansion is used to increase the distance between the nodes that cause an anti-dependency. The algorithm is outlined in Figure 8. The expansion itself is simply done by replicating all nodes and internal edges, connecting the tail of each iteration to the header of the next iteration, and connecting the tail of the last iteration to the header of the first iteration. Notice that the loop to be expanded is the extended loop $\widetilde{L}_h$ rather than L_h. Otherwise, the uses of t_k outside the loop may prevent the definitions of t_k in the loop to be freely renamed. The most important thing is to determine the constant T, i.e., the number of times the loop needs to be expanded ($T = 1$ means no expansion).

There are two kinds of anti-dependencies that need to be considered. One goes through the back edge (I_t, I_h) and occurs only when there is a flow dependency in the loop body; another does

```
if ($H(\widehat{G}_h) \neq \phi$) {
    define a set of flow dependencies $F = \{I_j \delta_x^f I_i | I_i \in L_h, I_j \in L_h\}$;
    for all flow dependencies $I_j \delta_x^f I_i \in F$, find the maximum $T_f(I_i, I_j)$ and denote it $T_f$:
        $T_f(I_i, I_j) = \begin{cases} 1 & \text{if } d(I_i, I_j) > N \\ \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D+1} \right\rfloor + 2 & \text{if } d(I_i, I_j) \leq N \end{cases}$
    define a set of anti-dependencies $A = \{I_i \delta_x^a I_j | I_i \in L_h, I_j \in L_h\}$;
    for all anti-dependencies $I_i \delta_x^a I_j \in A$, find the maximum $T_a(I_i, I_j)$ and denote it $T_a$:
        $T_a(I_i, I_j) = \begin{cases} 1 & \text{if } x \text{ is dead at the entry of } I_h \\ \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D+1} \right\rfloor + 3 & \text{if } x \text{ is live at the entry of } I_h \end{cases}$
    $T = \max(T_f, T_a)$;
    expand the loop $\widetilde{L}_h$ to $T$ consecutive iterations;
    rename all pseudo registers;
}
```

Figure 8. The loop expansion algorithm

not go through the back edge and occurs when there is an anti-dependency in the loop body itself. Therefore, we have two formulas shown in Figure 8 to calculate the number of times to expand. The constant D is the shortest distance from I_h to any tail node I_t. The formula for $T_f(I_i, I_j)$ is derived from the fact that, after the expansion, the distance between I_i and I_j is increased to $d(I_i, I_t) + (T_f(I_i, I_j) - 1) \times (D + 1) + 1 + d(I_h, I_j)$ which should be greater than N. Similarly, the formula for $T_a(I_i, I_j)$ is derived from the fact that, after the expansion, the distance between I_i and I_j is increased to $d(I_i, I_t) + (T_a(I_i, I_j) - 2) \times (D + 1) + 1 + d(I_h, I_j)$ which should be greater than N. The final T is just the maximum of the two numbers T_f and T_a.

The loop expansion technique is illustrated in Figure 9. The loop shown is an expanded loop of Figure 3(b) with $T = 2$, assuming the last use of t_1 is in I_5. Instructions I_6, I_7, I_8, and I_9 are copied from I_2, I_3, I_4, I_5 with t_8 replacing the t_1 in I_6, I_7, and I_9, and t_9 replacing the t_3 in I_2 and I_7. The distance for the anti-dependency $I_3 \delta_{t_1}^a I_2$ has been increased by 3, i.e., the length of the loop body. Since all anti-dependencies go through the iteration boundary after the node splitting

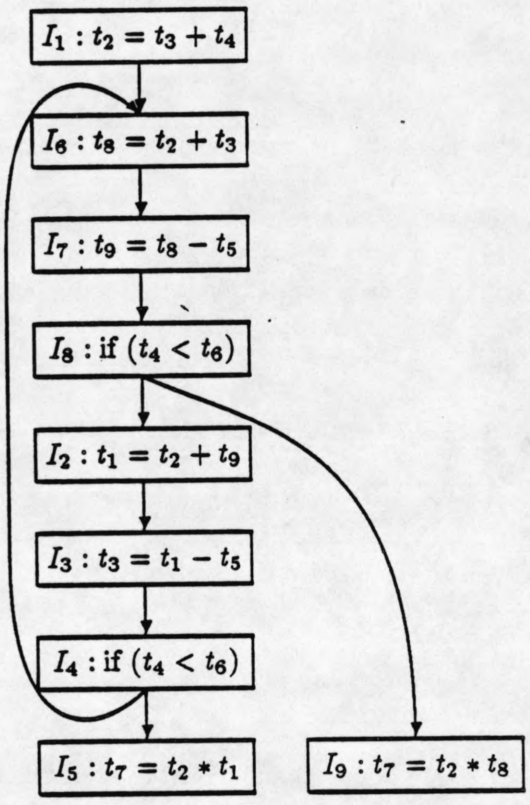


Figure 9. Application of the loop expansion algorithm

step, the distance can be increased indefinitely by increasing T. Therefore, the N-instruction anti-dependencies are resolved.

III. Machine Register Anti-Dependencies

The machine register anti-dependencies may be resolved by register allocation, post-pass code scheduling [20], or a combination of both. This paper examines the former approach.

A. Machine Model

The CPU model we consider in this paper does not have out-of-order execution, multiple instruction issuing, run-time register reordering, or register windows. Pipelining is allowed as long as the hardware can guarantee a precise instruction boundary when the error being detected requires a rollback.

The state of the Program Counter (PC) is preserved by an external recording device or by shadowing registers such as described in the micro rollback scheme [3]. The Program Status Word (PSW) is either not used in user space or is preserved by shadowing registers. Depending on the specific micro architecture, the Stack Pointer (SP) may be considered a special register (e.g., many 16-bit CPUs) or a general purpose register (e.g., most of the 32-bit CPUs). Our objective is to assign the general purpose registers such that the final code does not have any N-instruction machine register anti-dependency on the general purpose registers.

B. Register Allocation

Most register allocators that can handle global register assignment use the graph coloring method [14, 21]. By way of an interference graph, the register allocator guarantees that two values that may be simultaneously live do not occupy the same machine register. This type of constraint is called a *live range constraint*. If there are not enough registers available, spill code is generated to put aside some live values to main memory. For example, the solid lines in Figure 10(b) represent the live range constraints for the flow graph in Figure 10(a). The edge between t_1 and t_2 indicates that they may be live simultaneously, i.e., in instructions I_2, I_3, and I_4. If we have no less than 3 registers available, the code in Figure 10(c) could be generated; otherwise, some values such as t_3 may need to be spilled.

However, Figure 10(c) is not free of N-instruction machine register anti-dependencies if $N = 2$. Registers r_1 and r_2 are defined right after their use. Therefore, another type of constraint, called an *anti-dependency constraint*, is incorporated in the interference graph to prevent this situation.

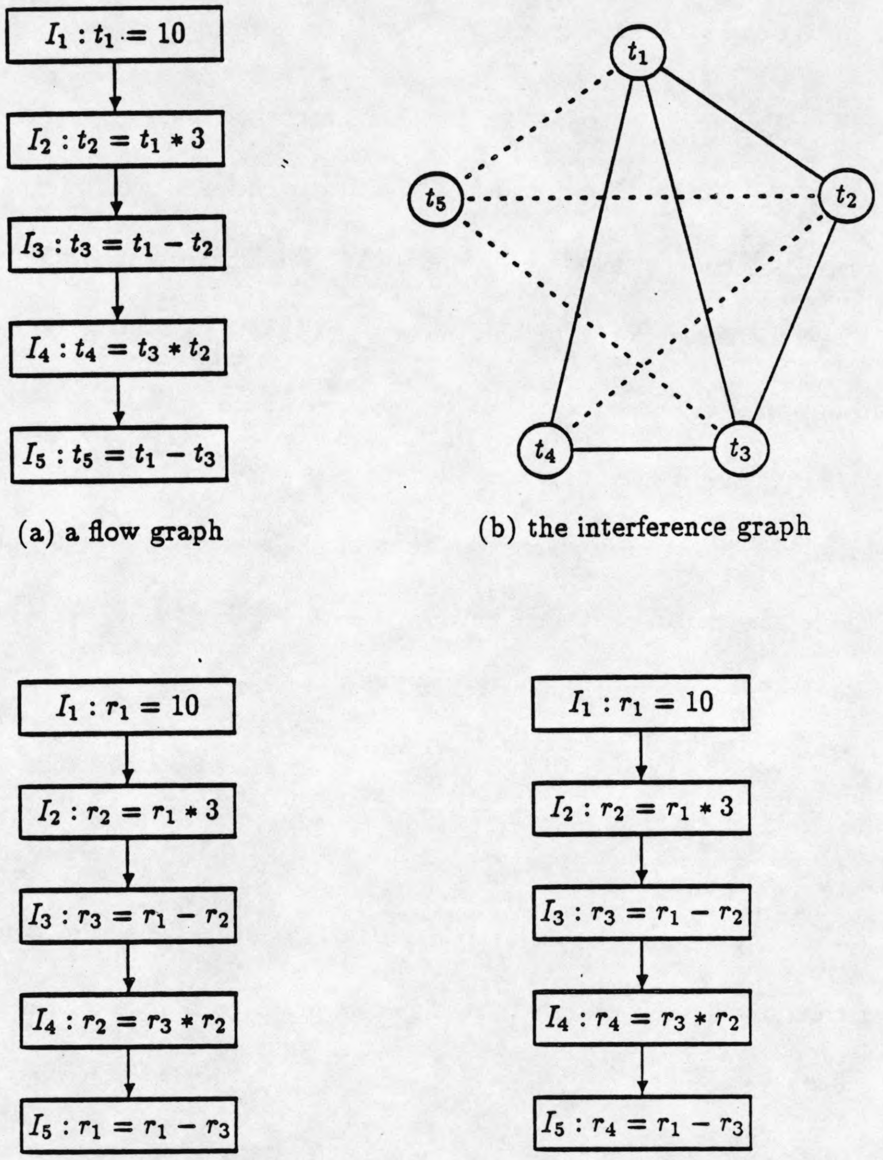


Figure 10. Adding the anti-dependency constraints to the interference graph

The anti-dependency constraint is stated as follows:

Any value being defined in the current instruction can not occupy a register that has been assigned to some value used within the previous N instructions.

The anti-dependency constraints for the flow graph in Figure 10(a) are represented by the dashed lines in Figure 10(b). If both types of edges exist between two nodes, only the solid line is shown. The resulting code is shown in Figure 10(d). Note that the minimum number of registers required has been increased from 3 to 4. If we have less than 4 registers available, some values such as t_3 need to be spilled.

Spill code may result in another problem: if two values in two consecutive instructions are both spilled and use the same spill register, then an N-instruction anti-dependency immediately follows if N is larger than the distance between the two spill code. For example, in Figure 10(a), if t_1 is spilled, the following spill code is generated for instructions I_2 and I_3:

load r_{15} by the value of t_1 from memory
$r_2 = r_{15} * 3$
load r_{15} by the value of t_1 from memory
$r_3 = r_{15} - r_2$

where register r_{15} is the spill register for operand 1. The anti-dependency between the second and the third instructions is easily seen. To resolve this problem, nops are inserted between them to increase the distance. Similar situations exist for the stack pointer and frame pointer adjustment at the beginning and end of a procedure or before and after a procedure call.

IV. Inter-Procedural Anti-Dependency Constraints

In ordinary register allocation algorithm, the live range constraint is maintained across procedure boundaries by one of the following methods:

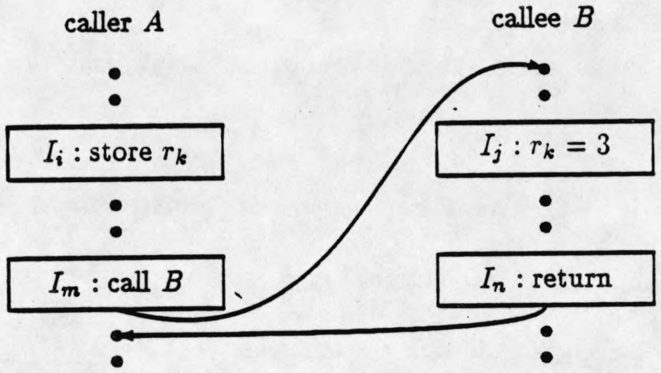


Figure 11. Inter-procedural anti-dependency $I_i \delta^a_{r_k} I_j$

1. Caller-saved registers: the registers containing live values are saved before a procedure call and restored after the call.
2. Callee-saved registers: the registers that may be changed in the callee are saved by the callee at the entry point and restored at the exit point.
3. Inter-procedural register allocation [21]: if every procedure is under the control of the current compiling session, registers may be allocated across procedure boundary.

However, the machine register anti-dependencies are not terminated even if the above methods are used. For example, in Figure 11, register r_k is used in both the caller procedure A and the callee procedure B. It is saved before the calling of B. But the initialization of r_k at the beginning of B results in an immediate anti-dependency if N is large enough.

To handle this problem, extra constraints are added to the following four regions:

1. Before a procedure call: the pseudo registers that are used within N instructions before the procedure call can only be assigned to register set R''.
2. Entry point of a procedure: the pseudo registers that are defined within N instructions after the entry of the procedure can only be assigned to register set R'.
3. Exit point of a procedure: the pseudo registers are used within N instructions before the return statement can only be assigned to register set R''.
4. After a procedure call: the pseudo registers that are defined within N instructions after the procedure call can only be assigned to register set R'.

As long as $R' \bigcap R'' = \phi$, no anti-dependency will occur across the procedure boundary. If an instruction belongs to more than one of the above regions, it should follow all the rules that apply.

Since the IMPACT C compiler always adjusts the stack pointer at the entry and exit points of a procedure, the above *inter-procedural anti-dependency constraints* are implemented by first splitting the stack pointer adjustment instruction '\$sp = \$sp $- a$' into two instructions '\$r = \$sp $- a$; \$sp = \$r' and then inserting nops to maintain the following conditions:

1. There should be at least N instruction between '\$r = \$sp $- a$' and '\$sp = \$r' at the entry.
2. There should be at least N instructions between the '\$sp = \$r' at the entry to any procedure call.
3. There should be at least N instructions between any procedure call to the '\$r = \$sp $+ a$' at the exit.
4. There should be at least N instructions between '\$r = \$sp $+ a$' and '\$sp = \$r' at the exit.

Machine register $\$r$ is a reserved register just for the stack handling purpose. Other unresolved anti-dependencies, such as between the preservation of a callee-saved register and its first assignment, are also solved by nop insertion.

V. PERFORMANCE EVALUATION

A. Implementation

The algorithms are implemented in the MIPS code generator of the IMPACT C compiler. The algorithms for resolving pseudo register anti-dependencies (loop protection, node splitting, and loop expansion) are called right before the register allocation phase. The machine register anti-dependency constraints are added after the live range constraints have been generated but before graph coloring. The nop insertion algorithm is called right before the assembly code output routine. The actual 'nop' inserted is the assembly code 'move \$0,\$0' to avoid the assembler complaining the 'nop' should be in a special block. Since register \$0 is hard-wired to the value 0, it can serve the purpose of 'nop'.

For simplicity, the data flow information (e.g., live variable, reaching definition. du chain) is not incrementally maintained. In other words, once a node is inserted or deleted, the entire graph needs to be processed again. This results in an intolerable compilation time for large procedures. To overcome this drawback, we set a threshold for the size of the procedure. Once the number of nodes in the graph exceeds this threshold, the algorithm enters the *simplified mode* which bypasses the rest of pseudo register anti-dependency processing except the breaking of self-anti-dependent instructions. In other words, the simplified mode transfers the responsibility of resolving anti-dependencies to the nop insertion phase. Currently, the threshold is set at 800 instructions. Both the threshold and the parameter N are supplied as a compiler switch to the code generator.

B. Benchmarks

Seven programs were cross-compiled on a SPARCserver 490 and run on a DECstation 3100. The original run time and size are listed in Table 1. The size is the number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead. QUEEN is based on the eight-queen program but with 12 queens as input. QSORT implements the quick sort algorithm to process a randomly generated array. Both QUEEN and QSORT use recursive calls. WC, CMP, and COMPRESS are well-known UNIX utilities. PUZZLE is a game. Finally, NOP is the nop insertion routine mentioned above.

Table 1. Original run time and size of benchmarks

program	run time (seconds)	number of static instructions
QUEEN	17.0	148
WC	11.3	181
QSORT	9.8	252
CMP	17.7	262
PUZZLE	15.0	932
NOP	27.5	2307
COMPRESS	41.3	1853

C. Performance data

There are several sources of performance degradation in our code transformation approach: 1) loop protection inserts save and restore nodes in the flow graph, 2) the machine register antidependency constraints result in the inefficiency in register usage and hence more spill code, 3) the nops inserted for consecutive spill code, stack pointer updates, and inter-procedural anti-dependency constraints will degrade the performance, and 4) the increased code size may increase the cache miss ratio.

We compile each benchmark program for $N = 1$ to 10, and selectively disabled the machine register anti-dependency solver and the nop inserter to generate a total of 31 versions (including the original version). Run time and size information for each benchmark are shown in Figure 12 to Figure 18. The x-axis is the parameter N. The y-axis is the percentage overhead. The dotted line is for the versions with machine register anti-dependency solver and nop inserter disabled, i.e., it shows the overhead that should be attributed to the pseudo register anti-dependency solver. The dashed line is for the versions with nop inserter disabled, i.e., it shows the combined overhead of pseudo register and machine register anti-dependency solver. The solid line gives the complete overhead figures. Note that the $N \geq 3$ versions for NOP and the $N \geq 1$ versions for COMPRESS

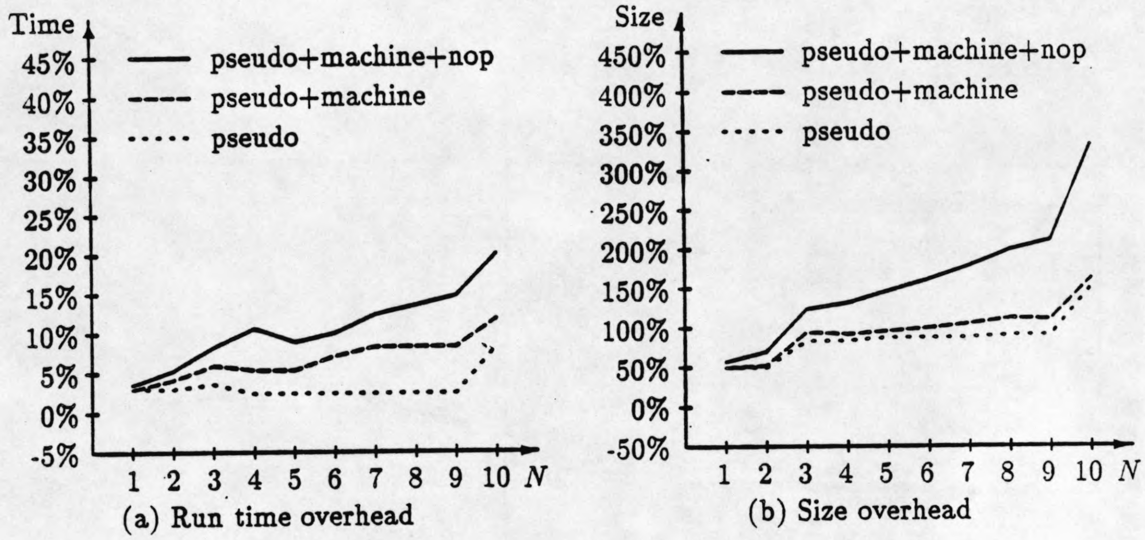


Figure 12. Run time and size overhead of QUEEN

have some functions compiled in simplified mode. That is, the run time shown is usually an over-estimate of the true number, and the size shown is usually an under-estimate. Also note that the libraries have not been recompiled by our compiler and the effect of the increased cache miss ratio is not separately measured.

For most of the benchmarks, the time and size overhead tends to increase with N as expected. However, this is not strictly true. For example, in Figure 12(a), the $N = 5$ version is a little faster than the $N = 4$ version. There are several sources for this irregularity: 1) the measurement error (about 0.1 to 0.2 seconds), 2) the postpass code reorganizer of the MIPS machine changed the execution order, 3) the register allocator is not optimal, 4) the inherent jump optimizer in the pseudo register anti-dependency solver made different decision for different N. In Figure 15(a) and Figure 16(a), the versions with $N \geq 1$ even run faster than the original version due to the latter three reasons mentioned above.

Notice that, in general, the difference between the dotted and the dashed lines of the run time figures increases with N. This is because larger N requires a register to hold a value longer

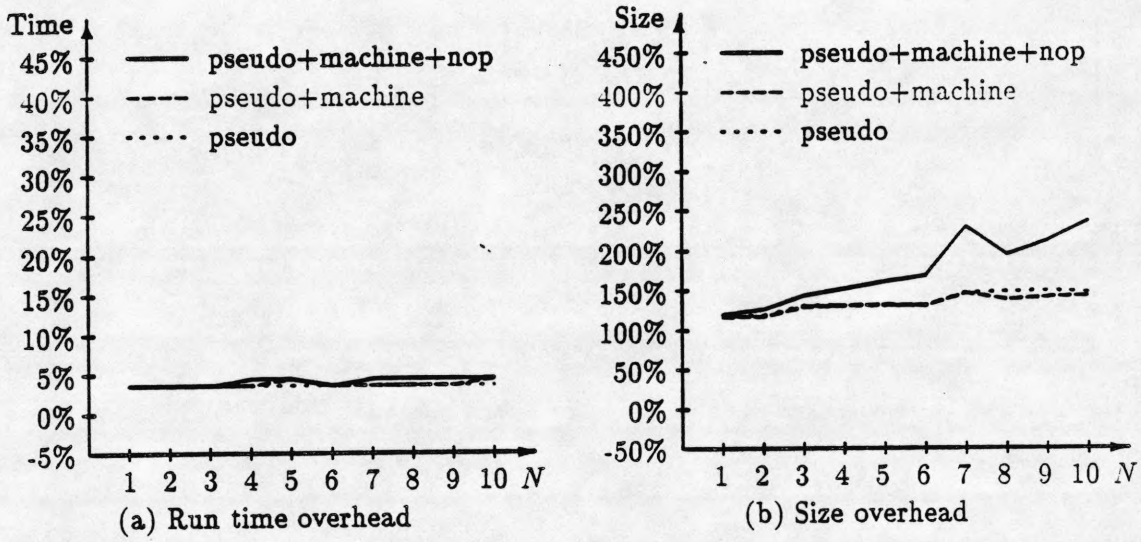


Figure 13. Run time and size overhead of WC

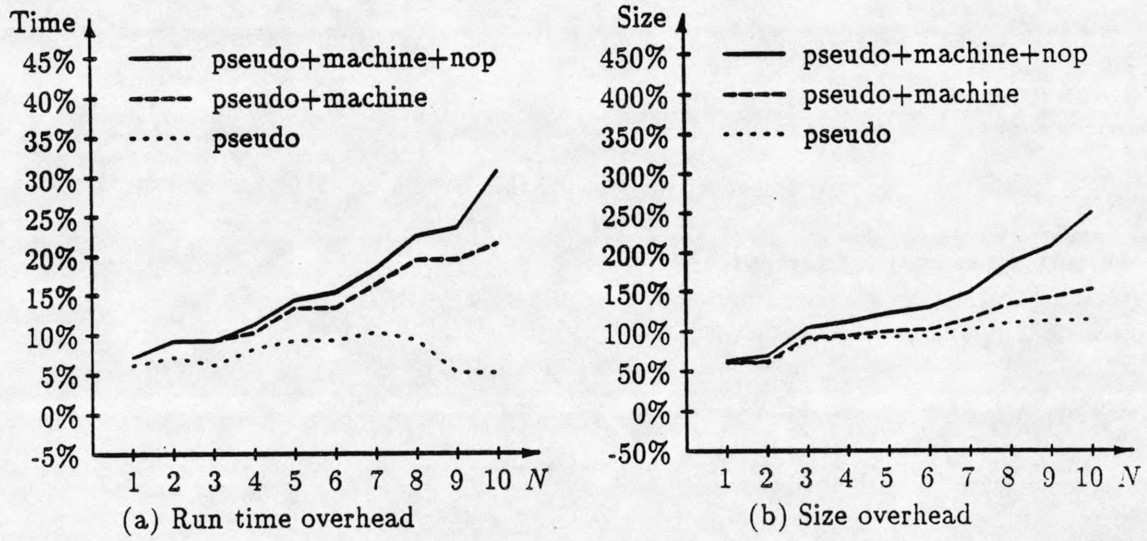


Figure 14. Run time and size overhead of QSORT

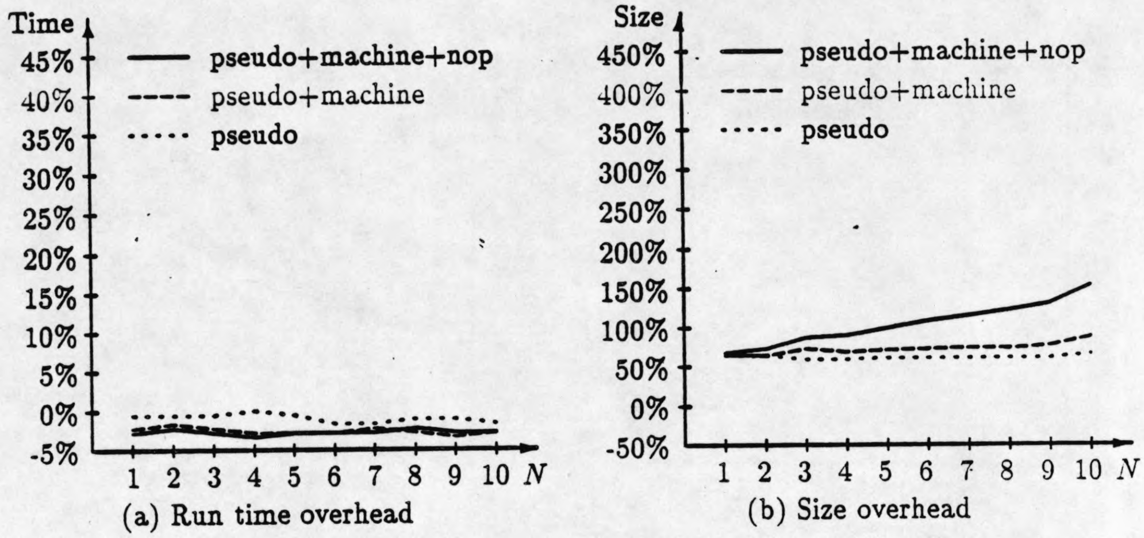


Figure 15. Run time and size overhead of CMP

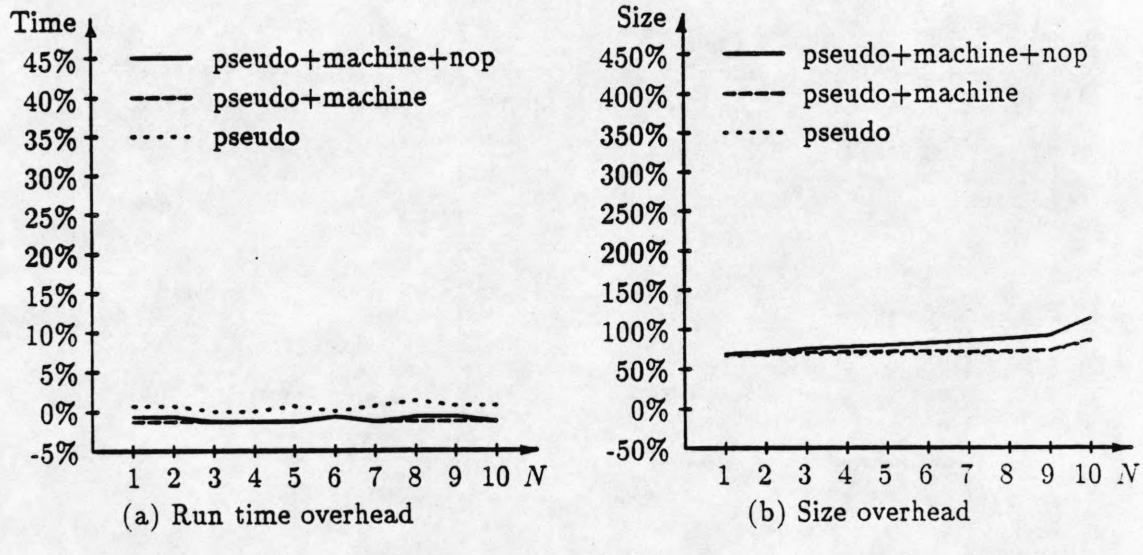
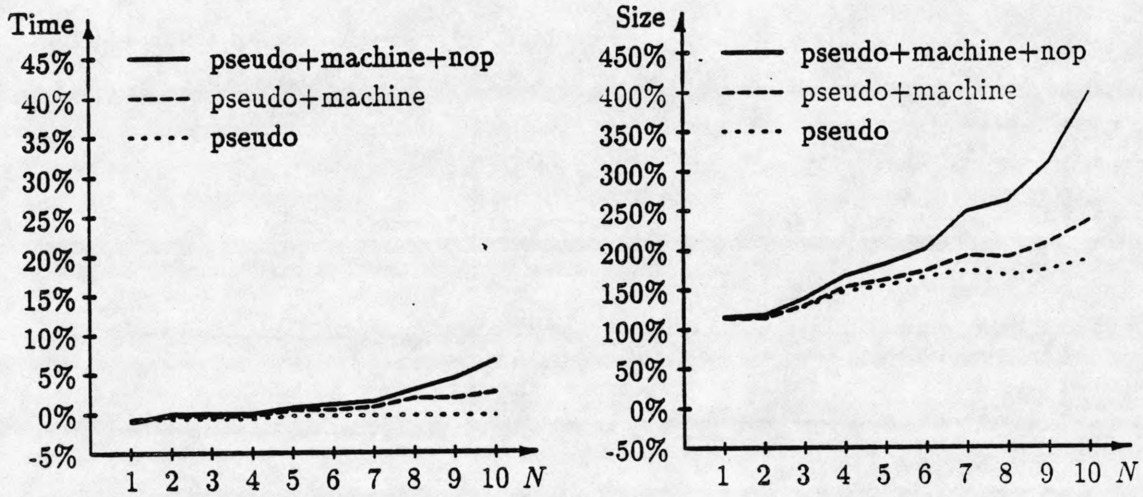
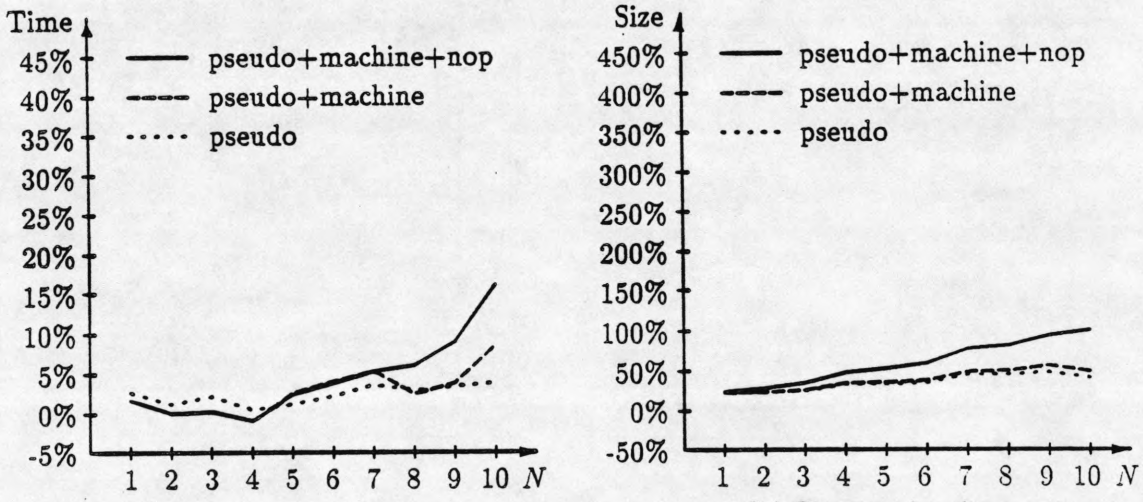


Figure 16. Run time and size overhead of PUZZLE



(a) Run time overhead (b) Size overhead

Figure 17. Run time and size overhead of NOP



(a) Run time overhead (b) Size overhead

Figure 18. Run time and size overhead of COMPRESS

before it can be used again. In other words, providing a larger register file can reduce the run time overhead attributed to the machine register anti-dependency constraints.

If the number of reserved spill registers is increased (currently it is 3), the nops inserted due to consecutive spills can be reduced. This in turn reduces the overhead shown by the solid lines. However, it is a conflicting goal with the one described in the previous paragraph since the total number of registers is fixed. There must be a compromise between these two goals. Code scheduling could further decrease the nop insertion overhead.

In summary, the run time overhead of this compiler-assisted approach is comparable to the hardware approach [3] for the examples examined with an additional benefit of changeable N. However, the cost is the increased compilation time and the larger executable code size. If more registers are provided, the performance will improve, with the dotted lines in Figure 12(a) – Figure 18(a) as lower bounds.

VI. Conclusion

This paper described a compiler-based alternative to a hardware delayed write buffer to preserve the state of the register file for N instructions. This objective is achieved by having the compiler remove all forms of anti-dependencies within N instructions. Our method used loop protection, node splitting, and loop expansion algorithms to remove pseudo register anti-dependencies: the anti-dependency constraints were added to the interference graph to prevent machine register anti-dependencies; the remaining anti-dependencies were resolved by nop insertion. The algorithms have been implemented in the IMPACT C compiler. The experimental results indicated that the run time performance of this software approach is comparable to that of the hardware approach, with an additional benefit of changeable N. The trade-off is the increased compilation time and the larger executable code size. The results also showed that a larger register file can further reduce the run time overhead.

References

[1] M. L. Ciacelli, "Fault handling on the IBM 4341 processor," in *The Eleventh International Symposium on Fault-Tolerant Computing*, pp. 9–12, June 1981.

[2] W. F. Bruckert and R. E. Josephson, "Designing reliability into the VAX 8600 system," *Digital Technical Journal of Digital Equipment Corporation*, pp. 71–77, Aug. 1985.

[3] Y. Tamir and M. Tremblay, "High-performance fault-tolerant vlsi systems using micro rollback," *IEEE Transactions on Computers*, vol. 39, pp. 548–554, Apr. 1990.

[4] M. S. Pittler, D. M. Powers, and D. L. Schnabel, "System development and technology aspects of the IBM 3081 processor complex," *IBM Journal of Research and Development*, vol. 26, pp. 2–11, Jan. 1982.

[5] R. N. Gustafson and F. J. Sparacio, "IBM 3081 processor unit: Design considerations and design process," *IBM Journal of Research and Development*, vol. 26, pp. 12–21, Jan. 1982.

[6] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 231–240, Apr. 1990.

[7] W.-M. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. 36, pp. 1496–1514, Dec. 1987.

[8] D. A. Padua and M. J. Wolfe, "Advanced computer optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] C.-C. J. Li and W. K. Fuchs, "CATCH - Compiler-Assisted Techniques for Checkpointing," in *The Twentieth International Symposium on Fault-Tolerant Computing*, pp. 74–81, June 1990.

[11] C.-C. J. Li and W. K. Fuchs, "Maintaining scalable checkpoints on hypercubes," in *The 1990 International Conference on Parallel Processing*, pp. II.98–II.104, Aug. 1990.

[12] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors," *IEEE Transactions on Computers*, vol. 39, pp. 436–446, Apr. 1990.

[13] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981.

[15] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *The ACM SIGPLAN'82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[16] W.-M. W. Hwu and P. P. Chang, "Inline function expansion for compiling c programs," in *The ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[17] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[18] C. L. Liu, *Elements of Discrete Mathematics.* McGraw-Hill, second ed., 1985.

[19] N. Wirth, *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

[20] J. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 422–448, July 1983.

[21] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *The ACM SIGPLAN'84 Symposium on Compiler Construction*, pp. 222–232, 1984.