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
1308 W. Main Street
Urbana, IL 61801

Correspondent: W. Kent Fuchs
TEL: (217)333-9731
FAX: (217)244-5686
Email: fuchs@crhc.uiuc.edu

Abstract

This paper describes a compiler-assisted approach to providing multiple instruction rollback capability for general purpose processor 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 have been implemented in the IMPACT C compiler. Experiments illustrating the effectiveness of this approach are described.

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

[1]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 in commercial machines, such as the IBM 4341 [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 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.

B. Error Model

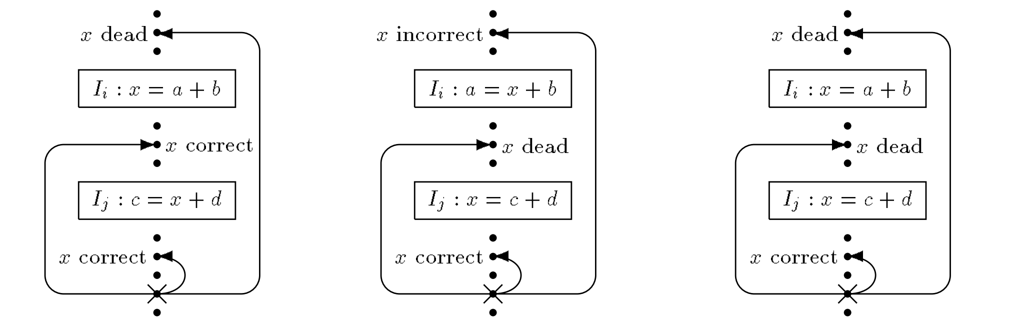
The design of a specific processor is not our focus, but rather the capability to perform fast recovery from transient processor faults by means of instruction retry. Our approach is particularly appropriate for RISC-type machines with large register files. To clarify which errors are considered in the multiple instruction retry scheme, we describe our assumptions:

1. The maximum error detection latency is N instructions.
2. 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.
3. There are delayed write buffers [3] for the memory system and I/O with capacity $C \geq N$. Otherwise, the memory system and I/O can not rollback to states consistent with CPU registers.
4. The CPU state can be restored by loading the correct contents of the register file and the program counter.
5. The register file contents do not spontaneously change and data is not written to an incorrect register location.

The scheme allows any error which does not result in an illegal path in the control flow graph. Such errors include CPU errors, incorrect data read from memory, I/O, and register locations, incorrect data writes to memory, I/O, and correct register locations, and incorrect branch decisions. As long as the instruction execution sequence forms a path in the control flow graph, data hazards can be classified into on-path hazards due to anti-dependencies and branch hazards due to incorrect branches [8]. Since all branch hazards can be treated as on-path hazards [8], in this paper we focus on techniques to remove all forms of N-instruction anti-dependencies.

C. Anti-Dependencies and Our Approach

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



(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.

write) [9]. 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. 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 figures. 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 it 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 removed.

Anti-dependencies come from two sources: 1) when the intermediate code generator assigns live values to *pseudo registers* (or symbolic registers) [10], 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). If variable x in both instructions I_i and I_j is assigned to pseudo register t_k by the intermediate code generator, it generates an anti-dependency on t_k. This type of anti-dependency is a *pseudo register anti-dependency*. The latter case may introduce machine register anti-dependencies 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 instructions I_i and I_j in r_k.

Compiler techniques have been used to assist in error recovery at the process level. For example, checkpoint decision [11] and multi-processor state compression [12] can be achieved by having the compiler insert appropriate code in the program. Also, algorithm-based error detection [13] can be implemented 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 instruction retry 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 [10, 14]. Our contribution is the formulation of the register state preservation for error recovery problem as an anti-dependency removal problem, the provision of a compiler-based solution, and an implementation with experimental results.

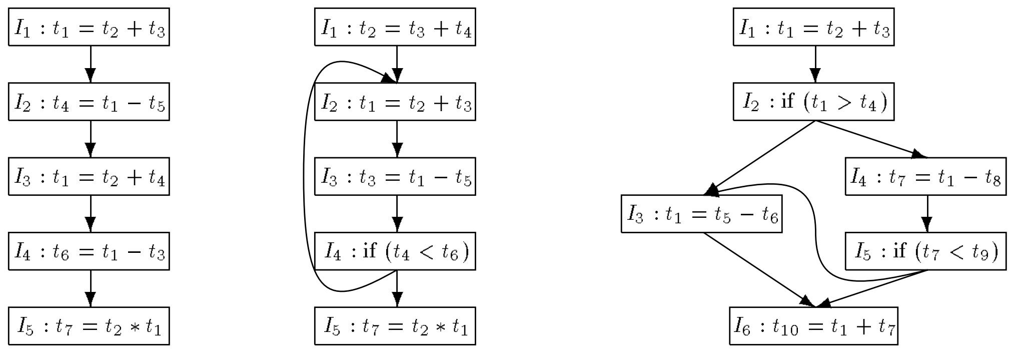
Section II describes the removal of N-instruction pseudo register anti-dependencies by *loop protection*, *node splitting*, and *loop expansion* techniques. Section III describes the prevention of N-instruction machine register anti-dependencies by introducing *anti-dependency constraints* in the interference graph used by the register allocator [15, 16]. 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 have been implemented in the MIPS code generator of the IMPACT C compiler [17]. Experiments evaluating the performance of our implementation are reported in Section V. Section VI describes techniques to enhance the run time performance of the code.

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* on variable x 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 [18], 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 instruction sequence.



(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 2. Resolvability of anti-dependencies.

B. Resolvability

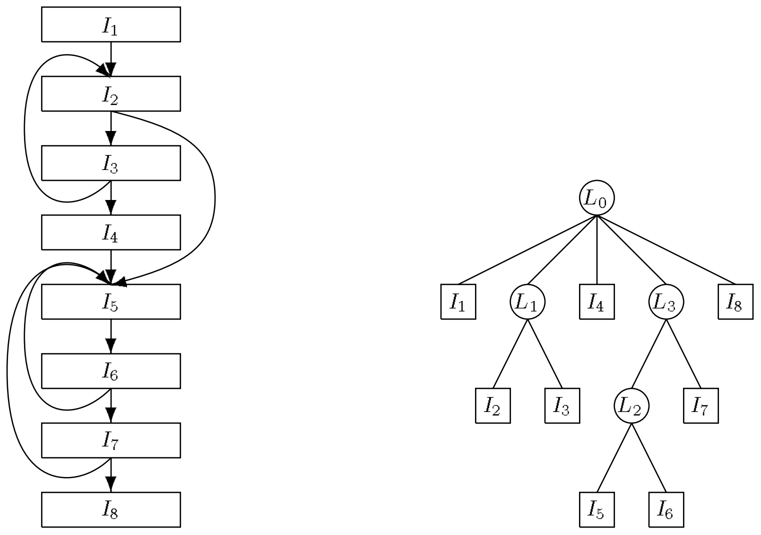
The basic approach to resolving an anti-dependency is to rename the pseudo registers. For example, in Figure 2(a), there is an anti-dependency $I_2\delta^a_{t_1}I_3$ that needs to be resolved if $N = 3$. This can be done by simply renaming the t_1 in I_3, I_4, and I_5 to t_8 since the value in t_1 is dead at the entry of I_3. However, some flow graphs do not allow proper renaming. For example, in Figure 2(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. This problem can occur even in acyclic graphs. For example, in Figure 2(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 2(b) and 2(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^d_i in Z_x; if it uses x, put a symbol I^u_i in Z_x. Define an equivalence relation $\equiv_x$ on Z_x as follows: if the renaming of x in I_i will result in the renaming of x in I_j, then $I^d_j \equiv_x I^u_i$. For example, 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 [10] 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 [19]. An anti-dependency $I_i \delta_x^a I_j$ 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 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 2(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 $I_4 \delta_{t_1}^a I_3$ 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 describe the loop structure of the program that guides the application of node splitting and loop expansion. We also 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) [10]. 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 [10]. 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



(a) nested loops (b) loop structure

Figure 3. Program loop structure.

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 represents the entire flow graph, each interior node indicates a loop, and each leaf node is an instruction. For example, the tree in Figure 3(b) describes the loop structure of the flow graph in Figure 3(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 highest 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, assuming that the root L_0 is at level 0. For example, in Figure 3, $I_5 \delta_x^a I_6$ is in level 2 (L_2), $I_6 \delta_x^a I_7$ is in level 1 (L_3), and $I_3 \delta_x^a I_4$ is in level 0 (L_0). 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 loops can then be sorted into an array by a topological sort algorithm [20]. 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 3 could be L_1, L_2, L_3, L_0, or L_2, L_1, L_3, L_0.

D. Loop Protection

Some anti-dependencies need to be resolved by node splitting or loop expansion, as described in the next two subsections. 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 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.

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 3(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* $\widetilde{L}_h(t_k)$ relative to pseudo register t_k consists of all nodes in L_h and all nodes I_i satisfying the following conditions: 1) $t_k \in \text{live\_in}(I_i)$, where $\text{live\_in}(I_i)$ is the set of live variables at the entry point of I_i [10], 2) I_i has only one successor, 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 3(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 $\widehat{E}_h = E - \{\text{all backedges associated with loops that are outer loops of } L_h\}$. The *hazard set* $H(G)$ of a graph G consists of all pseudo registers t_k such that $I_i \delta^a_{t_k} I_j$, $d(I_i, I_j) \leq N$, 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 *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.

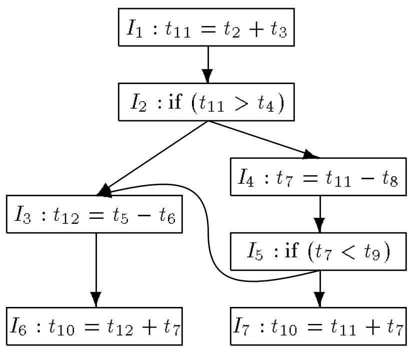
To prevent the loop header I_h from being split due to register t_k, L_h has to be protected by using the preheader and exit positions if $I_h \in S(\overline{G}, t_k)$. t_k may or may not be in $H(\widehat{G}_h)$. To provide the renaming capability after the loop is expanded for $t_k \in H(\widehat{G}_h)$, L_h has to be protected by using the header, tail, and exit positions, if 1) $\widetilde{L}_h(t_k)$ is not save; or 2) the header I_h of L_h is in $S(\widehat{G}_h, t_k)$. Every iteration of the expanded loop then has a unique set of save and restore nodes so that the pseudo registers can be renamed freely.

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 optimizing compilers [10]. 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.

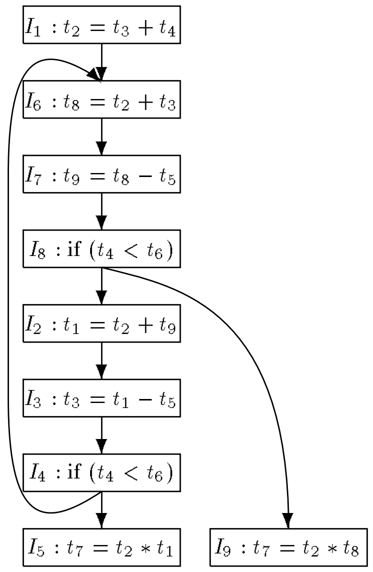
A node I_i will be in the split set $S(\overline{G}_h, t_k)$ if $t_k \in$ live_in(I_i) and there exists more than one definition of t_k that can reach I_i. After the splitting, two copies of the originally connected nodes are connected if they are *compatible*, i.e., they have the same reaching definition of t_k. For each $t_k \in H(\overline{G}_h)$, the algorithm splits all nodes in $S(\overline{G}_h, t_k)$ and matches the split nodes by a set of edges. It then renames all pseudo registers. Note that the header will not be in the split set since the loop has been protected.

Figure 4(a) shows the resulting flow graph after the code segment in Figure 2(c) is processed by 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 at I_6 resulting in the I_6 and I_7 in Figure 4(a). 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}.



$I_1 : t_{11} = t_2 + t_3$
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 $(t_7 < t_9)$
$I_6 : t_{10} = t_{12} + t_7$
$I_7 : t_{10} = t_{11} + t_7$

(a) node splitting for fig. 2(c)



(b) loop expansion for fig. 2(b)

Figure 4. Examples of node splitting and loop expansion.

The node splitting technique functions correctly due to 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 loop 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 5. 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. Usually, the restore nodes outside the loop body are in $\widetilde{L}_h - L_h$, and will be split due to expansion. The number of times the loop needs to be expanded, i.e., the constant T, is particularly important ($T = 1$ means no expansion).

There are two kinds of anti-dependencies that need to be considered. One is through the back edge (I_t, I_h) and occurs only when there is a flow dependency in the loop body; another does 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 5 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. For both cases, $T_f = T_a = 2$, if $d(I_i, I_t) + d(I_h, I_j) \geq N$. The final T is just the maximum of the two numbers T_f and T_a.

```
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 $T_f$:
            $T_f(I_i, I_j) = \begin{cases} 1 & \text{if } d(I_i, I_j) > N \\ 2 & \text{if } d(I_i, I_t) + d(I_h, I_j) + 1 > N \\ \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D+1} \right\rfloor + 2 & \text{otherwise} \end{cases}$
        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 $T_a$:
            $T_a(I_i, I_j) = \begin{cases} 1 & \text{if } x \text{ is dead at the entry of } I_h \\ 2 & \text{if } d(I_i, I_t) + d(I_h, I_j) + 1 > N \\ \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D+1} \right\rfloor + 3 & \text{otherwise} \end{cases}$
        $T = \max(T_f, T_a)$;
        expand the loop $\widetilde{L}_h$ to $T$ consecutive iterations;
        rename all pseudo registers;
}
```

Figure 5. The loop expansion algorithm.

The loop expansion technique is illustrated in Figure 4(b). The loop shown is an expanded loop of Figure 2(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 step, the distance can be increased indefinitely by increasing T. Therefore, the N-instruction anti-dependencies are resolved.

III. Machine Register Anti-Dependencies

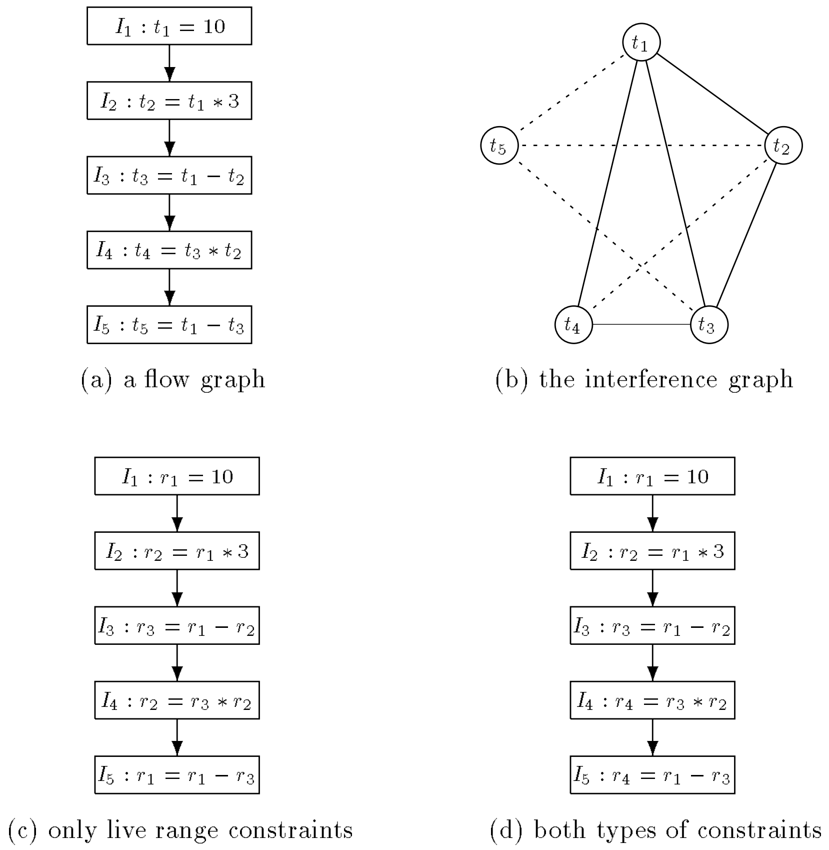
The machine register anti-dependencies may be resolved by register allocation, post-pass code scheduling [21], or a combination of both. The former approach modifies the existing register allocation algorithm by including an additional anti-dependency constraint. This approach may result in more register spillage and produce more memory accesses. The latter approach changes the order of instructions to improve performance. However, the inserted nop instructions needed to resolve remaining dependencies may slow down the code execution compared to the former approach. 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 detected error requires a rollback. The states of the program counter are preserved by an external recording device or by shadow registers such as described in the micro rollback scheme [3]. The program status word is either not used in user space or is preserved by shadowing registers. Depending on the specific micro architecture, the stack pointer 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 graph coloring [15, 22]. 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 6(b) represent the live range constraints for the flow graph in Figure 6(a). The edge between t_1 and t_2 indicates that they



(a) a flow graph (b) the interference graph

(c) only live range constraints (d) both types of constraints

Figure 6. Adding the anti-dependency constraints to the interference graph.

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 6(c) could be generated; otherwise, some values such as t_3 may need to be spilled.

However, Figure 6(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. The anti-dependency constraint is stated as follows:

Any value 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 6(a) are represented by the dashed lines in Figure 6(b). If both types of edges exist between two nodes, only the solid line is shown. The resulting code is shown in Figure 6(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, an anti-dependency may immediately follow. For example, in Figure 6(a), if all machine registers hold live values at instruction I_2, two register values have to be spilled to make room for registers t_2 and t_3. Suppose that the values of r_4 and r_5 are to be spilled, r_1 holds the value of t_1, and r_{15} is the spill register. The following spill code is generated for instructions I_2 and I_3:

```
$r_{15} = r_4$
save $r_{15}$ to memory
$r_4 = r_1 * 3$
$r_{15} = r_5$
save $r_{15}$ to memory
$r_5 = r_1 - r_4$
```

The anti-dependency on r_{15} between the second and the fourth instructions is easily seen if $N \leq 2$. We can increase the number of reserved spill registers (currently it is 3 in our implementation), so that consecutive spill instructions can use different spill registers. However, this reduces the number of usable registers which may cause more spilling. In this paper, we employ a simple approach by inserting nops between consecutive spill instructions to increase their 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.

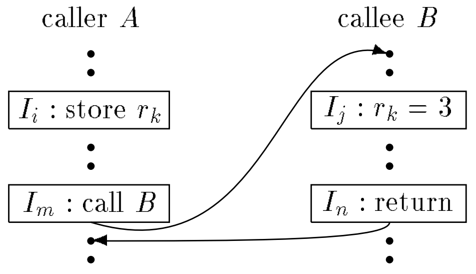


Figure 7. Inter-procedural anti-dependency $I_i \delta^a_{r_k} I_j$.

IV. Inter-Procedural Anti-Dependency Constraints

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

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 [22]: if every procedure is under the control of the current compiling session, registers may be allocated across procedure boundary.

The machine register anti-dependencies are not terminated even if the above methods are used. For example, in Figure 7, register r_k is used in both the caller procedure A and the callee procedure B. It is saved before the calling of B. However, 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 any necessary nops to maintain the following conditions:

1. at least N instruction between '\$r = \$sp $-\,a$' and '\$sp = \$r' at the entry;
2. at least N instructions between '\$sp = \$r' at the entry and any procedure call;
3. at least N instructions between any procedure call and '\$r = \$sp $+\,a$' at the exit;
4. at least N instructions between '\$r = \$sp $+\,a$' and '\$sp = \$r' at the exit;
5. at least N instructions between '\$sp = \$r' at the entry and '\$r = \$sp $+\,a$' at the exit; and
6. at least N instructions between any two procedure calls.

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 were implemented in the MIPS code generator of the IMPACT C compiler [17]. The algorithms for resolving pseudo register anti-dependencies (loop protection, node splitting, and

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

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.

We set a threshold for the maximum number of instructions 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 the experiments the threshold was set at 800 instructions.

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.

C. Performance Data

There are several potential 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 anti-dependency 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 compiled 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 8 to Figure 14. 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 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.

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 8(a), the $N = 5$ version is slightly faster than the $N = 4$ version. There are several sources for this irregularity: 1) measurement accuracy

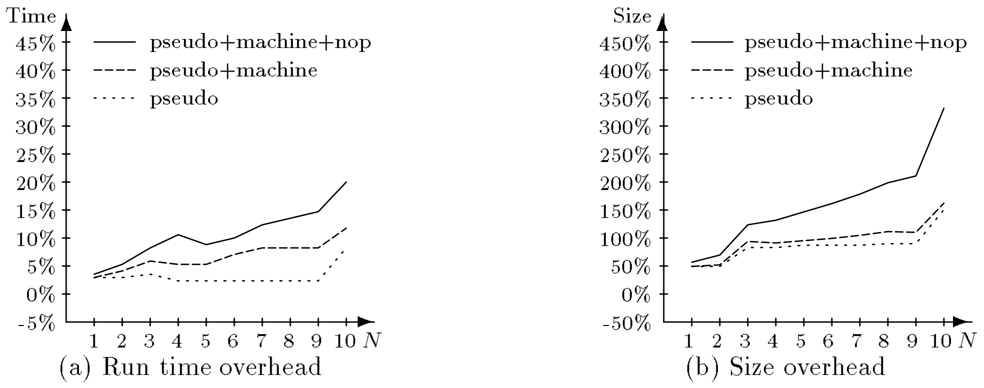


Figure 8. Run time and size overhead of QUEEN.

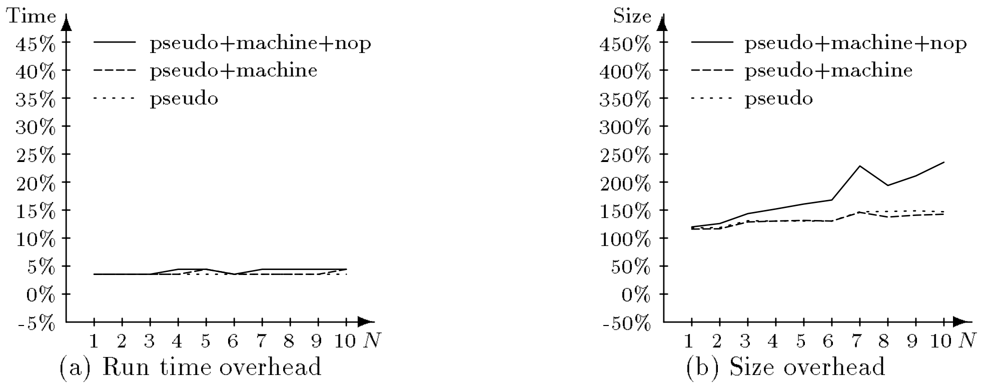


Figure 9. Run time and size overhead of WC.

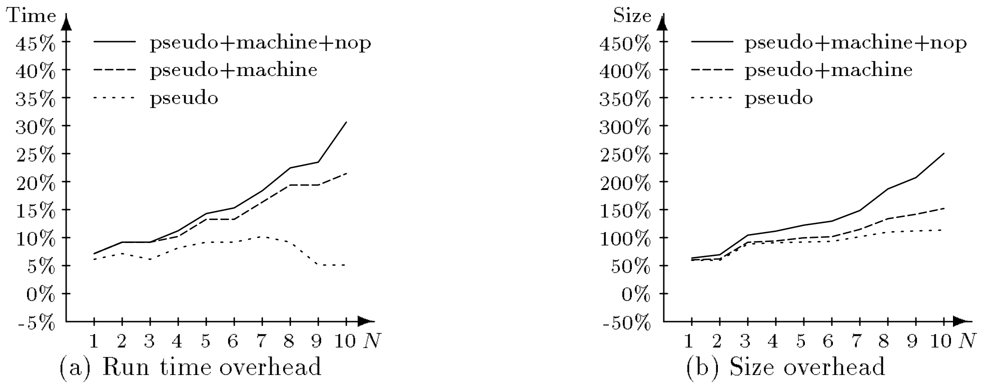
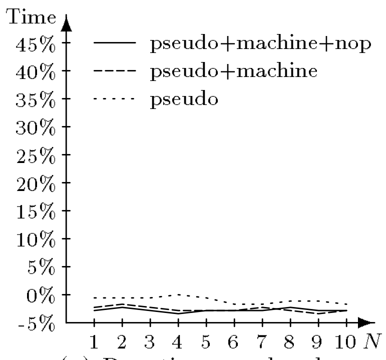
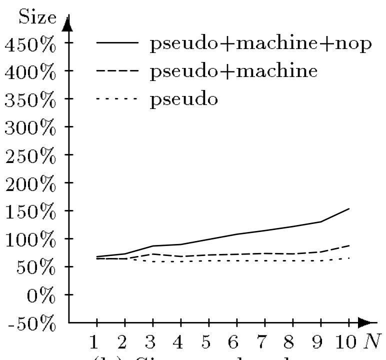


Figure 10. Run time and size overhead of QSORT.

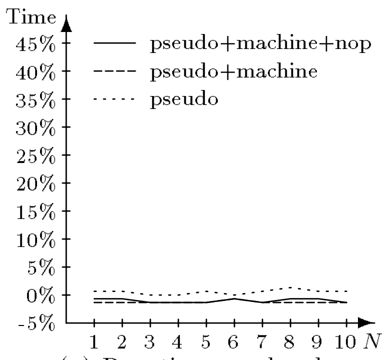


(a) Run time overhead

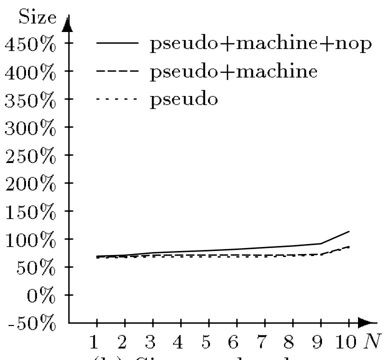


(b) Size overhead

Figure 11. Run time and size overhead of CMP.



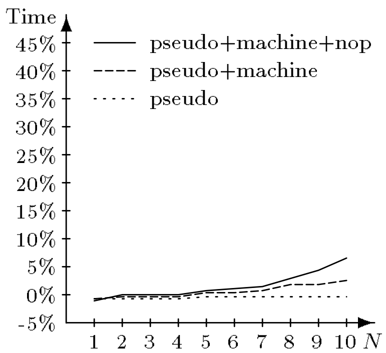
(a) Run time overhead



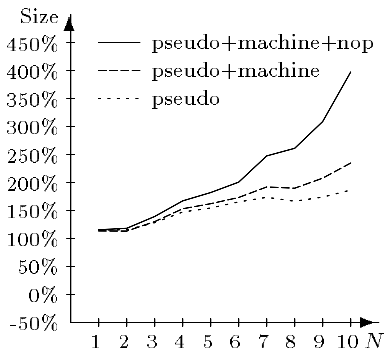
(b) Size overhead

Figure 12. Run time and size overhead of PUZZLE.

(about 0.1 to 0.2 seconds), 2) the post-pass code reorganizer of the MIPS machine sometimes changes the execution order for different N, 3) the register allocator is not always optimal, 4) the inherent jump optimizer in the pseudo register anti-dependency solver sometimes makes different decisions for different N. In Figure 11(a) and Figure 12(a), the versions with $N \geq 1$ run faster than the original version due to the latter three reasons mentioned above. Since register allocation is an NP-complete problem [23], the third reason may explain why for CMP and PUZZLE the versions generated by the pseudo register anti-dependency solver have larger run time overhead than those generated by applying one or two more phases. The heuristic algorithm may generate more spill code for the interference graph with only live-range constraints than for the interference graph with

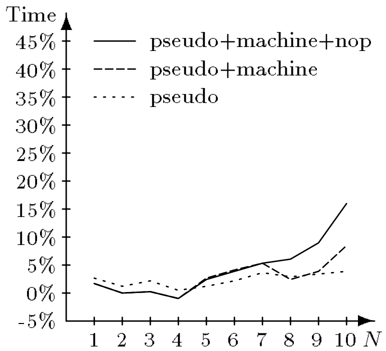


(a) Run time overhead

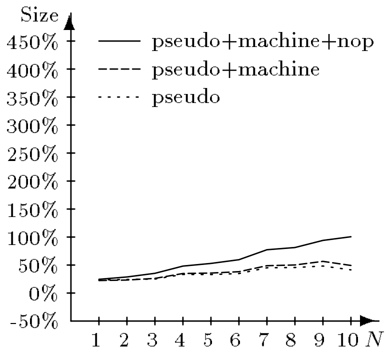


(b) Size overhead

Figure 13. Run time and size overhead of NOP.



(a) Run time overhead



(b) Size overhead

Figure 14. Run time and size overhead of COMPRESS.

both live-range and anti-dependency constraints.

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 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.

In summary, the run time overhead of this compiler-assisted approach is comparable to the write buffer 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 8(a)

– Figure 14(a) as lower bounds.

VI. Execution and Compile Time Performance Enhancements

The authors have observed that by modifying the processor to include a simple read buffer (not write buffer), compiler-assisted multiple instruction retry can be implemented with negligible run time performance degradation [8]. The enhancements described in this section provide for reduced performance degradation without utilizing a read buffer.

A. Save/Restore Node Ordering

During loop protection, the order of inserted nodes has an important impact on performance. If the save node is always inserted right before the header, and the restore node right after the tail or trailer, it produces immediate anti-dependent distances among the save/restore nodes, which may result in more copies of the current loop (or parent loops), during loop expansion.

We implemented the following techniques to resolve this problem : 1) when a save node is to be inserted at the preheader position, it is inserted right before the most recent save node, or before the header if there is no such node; and 2) when a restore node is to be inserted at the exit position, it is inserted right before the most recent restore node, or after the tail node if there is no such node.

For example, consider program WC, loop 1, $d(I_h, I_t) = 7$. For $N = 9$, the save/restore node pairs are $(S_1 : t_{87} = t_9; R_1 : t_9 = t_{87})$, and $(S_2 : t_{88} = t_{10}; R_2 : t_{10} = t_{88})$. If the order is S_1, S_2, $[L_1]$, R_2, R_1, then $d(S_1, R_1) = 11$, but $d(S_2, R_2) = 9 = N$, which creates a new hazard on register t_{10} for the outer loop. However, if we reverse the order of save nodes, i.e., S_2, S_1, $[L_1]$, R_2, R_1, then $d(S_1, R_1) = d(S_2, R_2) = 10 > N$.

B. Node Splitting

When performing node-splitting on a loop, only nodes inside the loop were considered due to loop protection. A straightforward approach would split a node according to the number of reaching definitions. If a hazard node is split into several copies, all of them are hazard nodes and become reaching definitions for later references. This may result in further splitting.

For any given node α, let L be the number of incoming edges, and M be the number of original reaching definitions that can reach α, among which K definitions are hazard. We have implemented an improved approach in which the number of copies, S, for node α after splitting satisfies the following formula : $S =$ 1) K, if $M = K$; or 2) $K + 1$, if $M > K$. This reduces the number of copies for splitting by viewing the definition in the hazard node and its split nodes as the same reaching definition.

C. Loop Expansion

If loop L is protected from inside on register t_k, every iteration of loop L executes a pair of extra instructions. Observing that if every path leading from loop header to loop tails has at least one instruction defining t_k, we can move the save/restore nodes for t_k out of loop L. This is because such instructions form new live ranges for t_k such that within the loop t_k can be renamed to the same new registers for different iterations.

D. Inter-Procedural Anti-Dependency

Condition 5 and condition 6 in Section IV can be removed by providing two registers $\$r$, $\$\hat{r}$, so that the stack pointer adjustment instruction at the entry can use $\$r$, and at the exit can use $\$\hat{r}$. In other words, the instruction '$\$sp = \$sp - a$' at the entry is split into two instructions

'$\$r = \$sp - a; \$sp = \$r$', and the instruction '$\$sp = \$sp + a$' at the exit is split into two instructions '$\$\hat{r} = \$sp + a; \$sp = \$\hat{r}$'.

Another performance improvement is based on condition 1 and condition 4 in Section IV relating to the prologue and epilogue segments in IMPACT C. The prologue segment includes code to adjust the stack pointer and to save the values of some local registers to memory, while the epilogue segment includes code to retrieve the original values of the same local registers from memory and to adjust the stack pointer. Figure 15(a) shows the prologue segment of the second function, *merge_sort*, of QSORT for $N = 10$. Figure 15(b) illustrates how the register assignment and code rescheduling are used to eliminate 19 nops in the prologue segment. Instruction 'move \$sp, \$30' has been moved after all instructions for saving local registers. The instructions to store local registers are rescheduled according to the first definitions of corresponding registers. For example, register \$16 is first defined after the prologue segment. The instruction to save \$16 is moved ahead. Register \$30 is used as a temporary stack pointer in the segment to reduce the number of nops between the last instruction for saving local registers and the instruction 'move \$sp, \$30'. Similar improvement is performed for the epilogue segment.

Furthermore, in addition to the three reserved spill registers, we can also utilize dead registers at specific locations for the purpose of spilling. A post-pass code rescheduling and register assignment is incorporated into the nop insertion phase. For example, among the test programs, QSORT has the worst performance degradation, 30.2% for $N = 10$ as shown in Figure 10(a). The second function, *merge_sort* , which can recursively call itself contributes to most of the run time overhead. By applying the additional post-pass code scheduling, the performance degradation has been reduced to 16.7%. The number of nops inserted is reduced from more than 150 to 40.

```
merge_sort :
$_merge_sort_1 :
prologue_begin :
          subu     $30,     $sp,     128

                    10  nops

          move     $sp,      $30
          sw       $31,     124($sp)
          sw       $23,     120($sp)
          sw       $22,     116($sp)
          sw       $21,     112($sp)
          sw       $20,     108($sp)
          sw       $19,     104($sp)
          sw       $18,     100($sp)
          sw       $17,     96($sp)
          sw       $16,     92($sp)

                    10  nops
prologue_end :
          move     $16,     $4
          move     $3,      $5
          sw       $3,       52($sp)
```

(a)

```
merge_sort :
$_merge_sort_1 :
prologue_begin :
          subu     $30,     $sp,     128
          sw       $16,     92($30)
          sw       $17,     96($30)
          sw       $31,     124($30)
          sw       $23,     120($30)
          sw       $22,     116($30)
          sw       $21,     112($30)
          sw       $20,     108($30)
          sw       $19,     104($30)
          sw       $18,     100($30)
          move     $0,      $0
          move     $sp,      $30
prologue_end :
          move     $16,     $4
          sw       $5,       52($sp)
```

(b)

Figure 15. Post-pass code rescheduling and register assignment for QSORT, $N = 10$.

VII. 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 anti-dependencies within N instructions. Our method uses loop protection, node splitting, and loop expansion algorithms to remove pseudo register anti-dependencies; the anti-dependency constraints are added to the interference graph to prevent machine register anti-dependencies; the remaining anti-dependencies are resolved by nop insertion. The algorithms have been implemented in the IMPACT C compiler. The experimental results indicate that the run time performance of this software approach is comparable to that of the write buffer hardware approach, with an additional benefit of changeable N.

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] N. J. Alewine, S.-K. Chen, C.-C. J. Li, W. K. Fuchs, and W.-M. W. Hwu, "Branch recovery with compiler-assisted multiple instruction retry," in *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 66–73, July 1992.

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness.* Freeman, 1979.