SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(9), 1–20(September 1994)

Incremental Compiler Transformations For Multiple Instruction Retry

SHYH-KWEI CHEN, NEAL J. ALEWINE*, W. KENT FUCHS, AND WEN-MEI W. HWU

Center for Reliable and High-performance Computing, Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, 1308 W. Main St., Urbana, Illinois 61801, U.S.A.

SUMMARY

Previous work on compiler-based multiple instruction retry has utilized a series of compiler transformations, loop protection, node splitting, and loop expansion, to eliminate anti-dependencies of length $\leq N$ in the pseudo register, the machine register, and the post-pass resolver phases of compilation [1]. The results have provided a means of rapidly recovering from transient processor failures by rolling back N instructions. This paper presents techniques for improving compilation and run-time performance in compiler-based multiple instruction retry. Incremental updating enhances compilation time when new instructions are added to the program. Post-pass code rescheduling and spill register reassignment algorithms improve the run-time performance and decrease the code growth across the application programs studied. Branch hazards are shown to be resolvable by simple modifications to the incremental updating schemes during the pseudo register phase and to the spill register reassignment algorithm during the post-pass phase.

KEY WORDS Rollback recovery Fault-tolerant computing Instruction retry

INTRODUCTION

To achieve uninterrupted operation, fault-tolerant computer systems usually possess the ability of detecting errors and either correcting the errors immediately or recovering the systems to previous consistent states prior to the occurrences of the errors. There is evidence that hardware transient faults, due mainly to temporary changes of electric, electromagnetic and radioactive conditions, occur more often than permanent faults [2,3]. In an environment with a high transient fault rate, it is desirable for a system to recover rapidly without resorting to a global restart whenever a fault occurs.

Software based checkpointing provides for rollback recovery when transient system faults occur. In such schemes, a checkpoint of the system state is captured and recorded at regular intervals [4,5,6], or predetermined positions in the application program [7]. In the event of a fault, the system can be rolled back to one of the previously recorded checkpoints, returning the system to a consistent state [8]. Software checkpointing can accommodate long error detection latencies at the cost of potentially long recovery time.

In contrast to full software checkpointing, multiple instruction retry schemes aid in rollback of just a few instructions. Instruction retry schemes have traditionally been implemented in hardware, both in full checkpointing [9,10,11,12,13], and in incremental checkpointing (sliding window) [14,15,16,17,18] formats.

* Current address is IBM Corp., Dept. 8C9A Internal Zip 2111, 1798 N.W. 48th St. Boca Raton, FL 33431, U.S.A.

0038–0644/94/090001–20$15.00

Received 9 August 1993
Revised 25 May 1994

Several multiple instruction retry schemes maintain at least two copies of the data for N cycles, which is the maximum error detection latency (or exception latency), before the old values can be destroyed. For example, IBM has utilized a shadow register file for backup of the entire register file [13], and single instruction retry has been implemented by twin shadow register files [14]. The DEC VAX 8600 [11] and VAX 9000 [12] implement single instruction retry without the need for complex redundant data storage. The IBM 3081 [10] supports full checkpointed rollback recovery with a checkpoint interval of 10 to 20 instructions. The shadow file serves as a history buffer, recording the old system state. The IBM ES/9000 [19] utilizes the virtual register management system to dynamically map 16 architectural registers to 32 physical registers. Data redundancy in the physical registers is used to assist in instruction retry. Similar techniques have been developed for exception handling in out-of-order architectures, including history buffers, reorder buffers, and future file structures [15].

The micro-rollback scheme [16] employs a delayed write buffer to sustain rollback capability. A write to the register file is written to the buffer instead. The delayed write buffer has N entries, where each entry corresponds to an instruction of the last N cycles and consists of two fields, the name of the destination register and the new value. The new value and the information regarding its destination register are held in the buffer for N cycles before updating the real register. A prioritized by-pass circuitry is needed to retrieve the most recent copy of the register. During recovery, the buffer contents are invalidated and the program counter(PC) and the program status word(PSW) are reloaded with the rollback values.

A compiler-based multiple instruction retry scheme has been developed, in which compiler-driven data flow manipulation is used to resolve data hazards associated with rollback recovery [1] by removing anti-dependencies of length $\leq N$ instructions. If an error is detected, the execution is recovered by loading the correct values of the PC and the PSW. The delayed write buffer for the register file is removed in this approach. However, the original implementation suffered from long compilation times. An alternative to the compiler-based technique is the combined compiler-hardware scheme [20], which can remove one type of hazard using a hardware read buffer, while allowing the compiler transformations to resolve the remaining hazards.

This paper addresses the compile-time limitations of the original compiler-based hazard removal approach to multiple instruction retry [1]. The techniques described include incremental updating, post-pass code rescheduling, spill register reassignment and branch hazard resolution.

ERROR MODEL AND HAZARD TYPES

Targeted processor errors are described as follows [20]. Error detection latency is $\leq N$ instructions. Units external to the CPU, such as memory and I/O, have their own rollback capability (e.g., delayed write buffers of depth N and appropriate bypass logic). The PC and the PSW contents at each instruction are preserved by an external recording device or by shadow registers [16]. A restartable CPU state can be restored by loading the correct contents of the PC and the PSW.

Given the above assumptions, a permissible error is one which does not result in a path inconsistent with the control flow graph (CFG) of the target application program provided that the register file contents do not spontaneously change and data is not written to an incorrect register location. Errors targeted for recovery via multiple instruction retry are summarized as follows: 1) CPU errors such as those caused by a faulty ALU; 2) incorrect values read from memory, the register file, or external functional units such as the floating point unit; 3) correct/incorrect operands read from incorrect locations within the I/O, memory, or register file; and 4) incorrect branch decisions resulting from errors 1 through 3.

The code can be represented as a CFG, $G(V, E)$, where V is the set of nodes denoting instructions and E the set of edges denoting flow information. If there is a direct control flow from instructions I_i to I_j, where $I_i \in V$ and $I_j \in V$, then there is an edge $(I_i, I_j) \in E$.

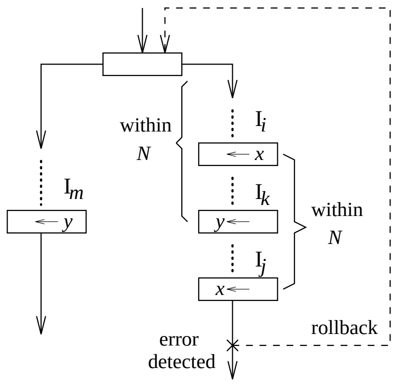


Figure 1. On-path and branch hazards

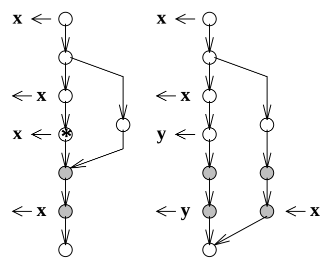
Within the general error model above, data hazards resulting from instruction retry are of two types [20]. *On-path hazards* are those encountered when the instruction path after rollback is the same as the initial instruction path and *branch hazards* are those encountered when the instruction path after rollback is different from the initial instruction path. On-path hazards can be described as anti-dependencies of length $\leq N$ in $G(V, E)$ [21]. As shown in Figure 1, register x of node I_j represents an on-path hazard and register y of node I_k represents a branch hazard.

OVERVIEW OF SCHEMES IMPLEMENTED

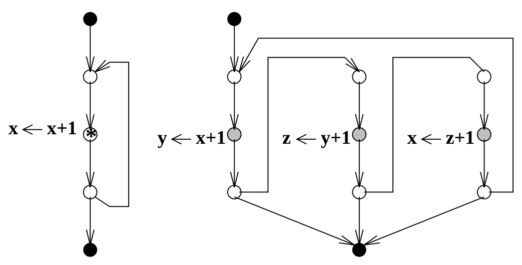
Our implementation is based on the intermediate code generated by the IMPACT C compiler [22] after optimization but before register allocation. Data hazards are resolved in three different phases, the *pseudo register* phase, the *machine register* phase, and the *nop insertion* phase. In order to compare compile time and run time efficiency, alternative schemes for each of the phases were implemented, as shown in Table I.

Table I. Schemes implemented

	Pseudo register	machine register	Nop insertion
Scheme L	on-path	on-path	on-path
Scheme A	on-path + branch[*]	on-path + branch	on-path + branch
Scheme 0	on-path[i]	on-path	on-path[cr]
Scheme 1	on-path[i]	on-path	on-path + branch[cr]
Scheme 2	on-path + branch[i]	on-path	on-path + branch[cr]
Scheme 3	on-path + branch[i]	on-path + branch	on-path + branch[cr]



(a) Node splitting , *N = 5.*



(b) Loop expansion , *N = 5.*

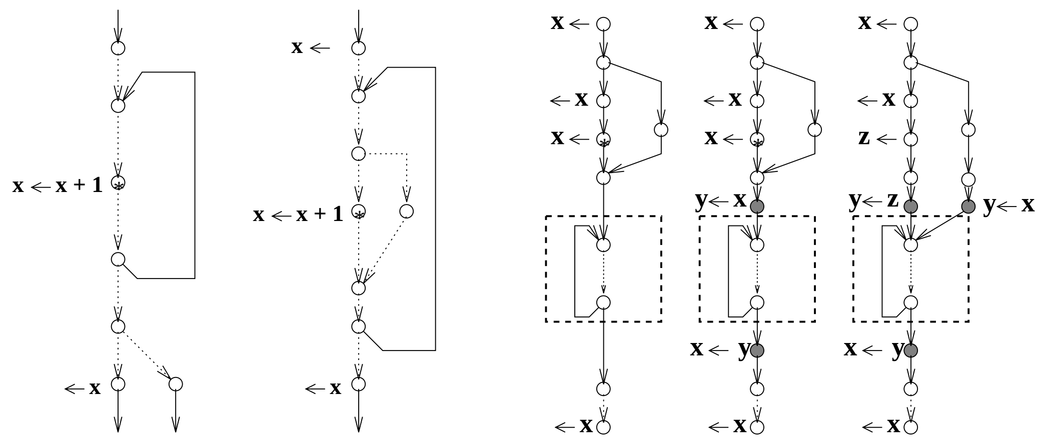
Figure 2. Node splitting, loop expansion, and renaming

Scheme L [1] resolves on-path hazards only. Scheme A [20] resolves both on-path and branch hazards at the latter two phases, but does not resolve all pseudo register branch hazards at the first phase, as marked “[*]”. The dominant fraction of compile time in the previous Schemes L and A is devoted to resolving pseudo register hazards. Both schemes implement a simple pseudo register phase, and the data structure updating is not incrementally maintained. Therefore, four alternative schemes that exploit incremental compilation techniques were implemented and compared. Scheme 0 uses incremental updating in the pseudo register phase for resolving on-path hazards. Compilation time has been enhanced with respect to Scheme L. Scheme 0 also employs post-pass code rescheduling and spill register reassignment algorithms to enhance the run-time performance and decrease the code growth across the application programs studied. The marker “[i]” denotes incremental updating, while “[cr]” denotes code rescheduling. Modifications to the post-pass algorithms can resolve both types of hazards during the nop insertion phase (Schemes 1, 2, and 3). We also show that a slightly modified incremental updating scheme can resolve branch hazards as well in the pseudo register phase (Schemes 2 and 3), though experimental results favor Scheme 1 in code run-time, code growth and compilation speed.

REVIEW OF THE PSEUDO REGISTER PHASE IN SCHEME L

The following notation is for on-path hazards, while those for branch hazards can be similarly defined. A node I_d is a *hazard node* if I_d defines a register x, another node I_u uses x, and there is a directed path of length less than or equal to N from I_u to I_d. Register x is called a *hazard register* or a *hazard* that causes data inconsistency. A loop header is the beginning of the loop, and a loop tail is the node that is within the loop and has a directed connection to the loop header. $Live\_in(I)$ and $live\_out(I)$ are the sets of registers whose values have later uses immediately before and after node I respectively [23].

Scheme L resolves pseudo register hazards in three sequential stages, loop protection, node splitting, and loop expansion. All three stages may insert new nodes, which change the CFG, loop structure, and data flow information. Renaming is the primary technique for hazard resolution. Figure 2 illustrates how node splitting and loop expansion resolve hazards. A hazard node is denoted by a circle with a “*”. In Scheme L, node I_j will be split due to hazard register x if $x \in live\_in(I_j)$ and there is more than one definition of x that can reach I_j. Nodes are scanned sequentially in the node splitting process. Scheme L also derives the number of times a loop should be expanded to resolve hazards. To prevent some loop headers



(a) Condition C1. (b) Condition C2. (c) Condition C3.

Figure 3. Conditions for loop protection

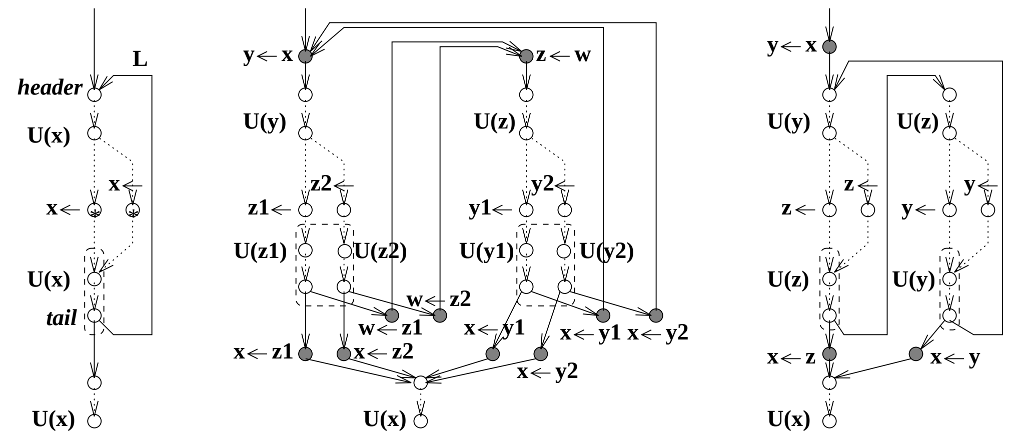
from being split, and to allow the targeted hazards to be renamed freely after loop expansion, save and restore nodes are inserted around loop headers, tails, and exit nodes. A loop can be protected either from outside or from inside. The following conditions are used to determine if a loop L should be protected for register x: C1. x is a hazard register which is live after the extended loop $\tilde{L}$ for register x; C2. L's header will be split due to its hazard register x; and C3. L's header will be split due to out of loop hazard register x.

The extended loop $\tilde{L}$ for register x consists of all nodes in L and all nodes I_1 satisfying the following rules: 1) $x \in$ live_in(I_1), 2) I_1 has only one successor, 3) I_1 has only one predecessor I_0, and 4) I_0 is in $\tilde{L}$. C2 may occur since some tail has more than one reaching definition and at least one is a hazard node defining x. If C1 or C2 is true, L is protected from inside. If C3 is true, L is protected from outside. C1 is for $\tilde{L}$ instead of L since L may not have to be protected if all nodes in which x is live after L are in $\tilde{L} - L$. Checking C3 prevents L's header from being split, while observing C1 and C2 can confine x's live range to within each iteration of L, so that after loop expansion, x can be renamed correctly within each new loop copy.

Example Figure 3 illustrates the three conditions for loop protection. In Figure 3(c), to limit the code growth, the loop on the splitting path needs to be protected from outside for x. Dotted lines denote that there may be some nodes in between as long as they do not redefine register x.

PERFORMANCE ENHANCEMENT TECHNIQUES – PSEUDO REGISTER PHASE

Let $d(I_i, I_j)$ denote the minimum number of edges on any path from I_i to I_j, and $d_L(I_i, I_j)$ is similar to $d(I_i, I_j)$ except that all the nodes in the minimum length path must be within loop L. Let D_L denote the minimum number of edges from L's loop header to any of L's tail. $\{I_u, I_d\}$ is a *hazard pair* within loop L on register x if I_u uses x, I_d defines x, and $d_L(I_u, I_d) \leq N$. Register x is a *cut hazard register* in loop L if 1) there is a hazard node in L, defining x; and 2) any header to tail path within loop L has at least one node defining x.



(a) A loop segment. (b) Scheme L. (c) The new scheme.

Figure 4. Loop protection and cut hazard register

Loop protection

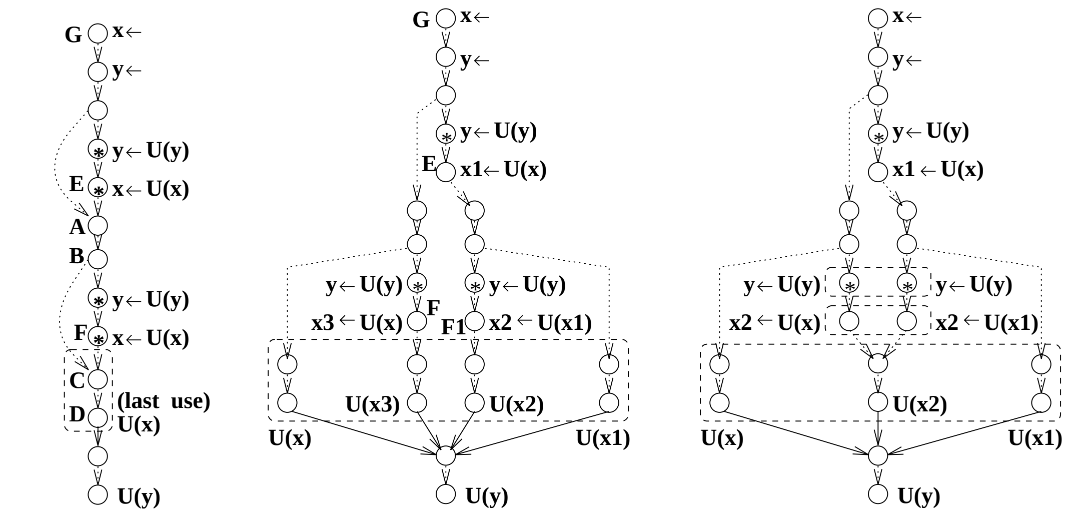
In Scheme L, if x is a hazard register inside loop L, and condition C1 or C2 is true, then L should be protected from inside for x. However, if x is also a cut hazard register, L can be renamed correctly after being protected from outside for x and expanded a sufficient number of times.

Example As shown in Figure 4(a), register x is such a cut hazard register. According to Scheme L, L is protected from inside since both conditions C1 and C2 are true. $U(x)$ represents using register x. Figure 4(b) illustrates the program segment after applying node splitting, loop expansion twice, and renaming. The shaded circles denote save and restore nodes for register x. Observing that every iteration of the loop redefines x, we can protect L from outside and still get the correct renaming after expanding the loop twice, as shown in Figure 4(c). The number of nodes is reduced, and the run time is improved since every iteration of loop L dose not execute save and restore nodes. Similarly register x within the loop in Figure 3(a) is also a cut hazard register.

Node splitting

A hazard node can be split if it is on the splitting path of some other hazards. Such new hazard nodes may cause redundant splittings in Scheme L. We have implemented a scheme in which the number of copies for node I after splitting equals the number of original hazard reaching definitions (plus 1 if there is at least one non-hazard reaching definition). This can be done by using a stamp heap data structure [24], so that if a hazard node I is split into I, I_1, $I_2, \ldots, I_{S-1}$, then the stamp field of I_i, $i = 1, 2, \ldots, S-1$, points to I. The hazard nodes in the same heap will be assigned to the same new destination register if renaming is required. During the sequential scanning process, node I should be split due to hazard register x if 1) x is in *live_in(I)*, and 2) all reaching definitions of x that can reach I do not belong to the same stamp heap, assuming that all non-hazard nodes defining x belong to the same stamp heap.

Example Consider the program segment shown in Figure 5(a). We process hazard register



(a) A program segment. (b) Split and rename based on x. (c) The new node splitting strategy.

Figure 5. Node splitting enhancement

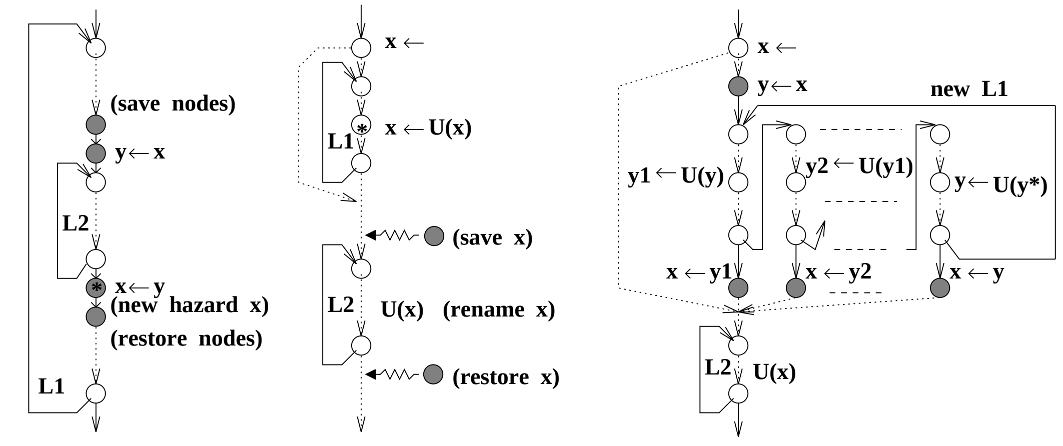
x first. The nodes from A to B are split into two copies in Scheme L. However, the hazard node F, defining x, is also split since x is still live, resulting in two hazard nodes, F and F_1. Therefore, the nodes from C to D are split into four copies due to the hazard reaching definitions E, F, F_1, and the reaching definition G. By applying the stamp heap strategy, we can view F and F_1 as the same reaching definition. Only three copies are required from nodes C to D, as shown in Figure 5(c).

Loop and node processing order

Node splitting transforms all the hazards within the current loop across its backedges, while loop expansion resolves all such hazards. In this manner, when we process a given loop, there is no data hazard across the backedges of its inner loops. Therefore it is natural to process the loops from inside out so that the levels of data hazards can be successively reduced until all of them occur at the root level. The hazards at the root level then can be resolved by node splitting and renaming.

In addition to the inner loop first rule, we have to enforce the sequential order rule (top-down) to smoothly check condition C3 for parent hazard registers and to further eliminate extra save/restore nodes. Figure 6(a) illustrates the inner loop first rule that new hazards due to loop protection are propagated to the outer loop. The program segment in Figure 6(b) illustrates the sequential order rule. Suppose L_2 is processed first. Without enforcing the sequential order rule, L_2 may need to be protected from outside for register x. However, such protection is redundant if we process L_1 first and remove hazards that might affect L_2, as shown in Figure 6(c).

Breadth First Search (BFS) is used to determine the processing order of nodes within loops or nodes of the entire program. The starting nodes may be the headers of loops or the root of the program. For some procedures, we have to modify the BFS algorithm by enforcing the following rules : 1) a node can be processed if and only if all of its parents have been



(a) The inner loop first rule. (b) Process L2 first. (c) Process L1 first.

Figure 6. Loop processing order

processed, MBFS; and 2) reverse the direction of searching, RBFS.

Loop expansion

Our formula for the number of copies of L needed to resolve all on-path hazards within L is the same as the formula in Scheme L [1], with a slight modification. To simplify the analysis, we assume that loop L has a header I_h, and a single tail I_t. It can be easily extended to loops with multiple tails. Let $D_L = d(I_h, I_t)$. Assume that $\{I_u, I_d\}$ is a hazard pair within loop L for register x. The new formula includes the following cases: Case 1. The backedge (I_t, I_h) is not counted in $d_L(I_u, I_d)$; Case 2. The backedge (I_t, I_h) is counted in $d_L(I_u, I_d)$, and within L there exists a directed path that does not include (I_t, I_h) from I_d to I_u; and Case 3. The backedge (I_t, I_h) is counted in $d_L(I_u, I_d)$, and within L not considering (I_t, I_h), there is no directed path from I_d to I_u.

Suppose it takes K_1, K_2 and K_3 copies to resolve the hazard pair $\{I_u, I_d\}$ for each case respectively, where a value 1 denotes no replication. For case 1, since the use of x in I_u and the definition of x in I_d are renamed to different registers in the same loop iteration after expanding the loop K_1 times, the potential hazard distance would be from I_u through ($K_1 - 2$) copies of L to the K_1th copy of I_d, which is $d(I_u, I_t) + (K_1 - 2)D_L + d(I_h, I_d) + K_1 - 1$, as shown in Figure 7(a). On the other hand, for case 2, both x's can be renamed to the same register, and the potential distance is from I_u through ($K_2 - 1$) copies of L to the first copy of I_d, which is $d(I_u, I_t) + (K_2 - 1)D_L + d(I_h, I_d) + K_2$, as shown in Figure 7(b). These terms must be greater than N. Solving both inequalities, we have $K_2 = \left\lfloor \frac{N - d(I_u, I_t) - d(I_h, I_d) - 1}{D_L + 1} \right\rfloor + 2$, and

$$K_1 = \begin{cases} 2 & \text{, if } d(I_u, I_t) + d(I_h, I_d) + 1 > N \\ \left\lfloor \frac{N - d(I_u, I_t) - d(I_h, I_d) - 1}{D_L + 1} \right\rfloor + 3 & \text{, otherwise.} \end{cases}$$

For case 3, due to our observation concerning cut hazard registers, K_3 may be either K_1

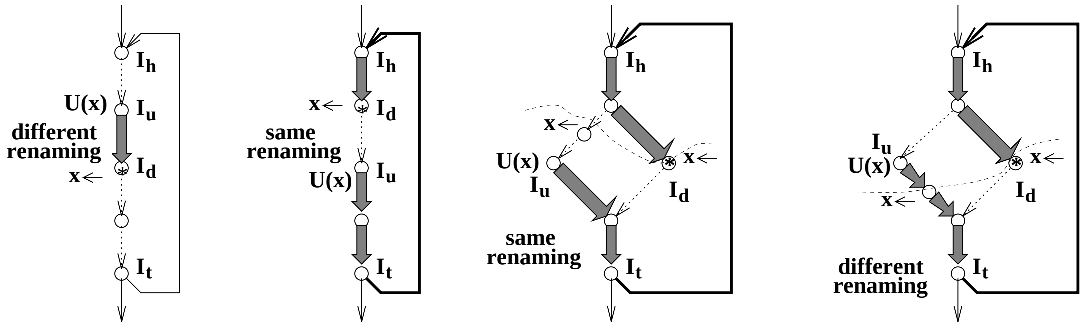


Figure 7. Cases for loop expansion

or K_2, depending on if both x's in I_u and I_d can be renamed to different registers, as shown in Figure 7(c) and (d) respectively. For fixed $d(I_u, I_t)$ and $d(I_h, I_d)$, we have $K_1 = K_2 + 1$. Since case 3 rarely occurs, we choose $K_3 = K_1$ in our implementation. The number of copies of L needed to resolve all hazards within L is the maximum of all such K's.

Self-anti-dependency

A node I is *self-anti-dependent* if I defines what it uses. For example, $x \leftarrow x + a$ is a self-anti-dependent node that uses and defines pseudo register x. This type of anti-dependency can be resolved by splitting I into two nodes : ($I_1 : y \leftarrow x + a$, $I_2 : x \leftarrow y$), and then inserting N nops between them [1,20]. However, using renaming with the aid of node splitting and loop protection, we can rename the definition of x to a new pseudo register without introducing a new node.

THE INCREMENTAL UPDATING SCHEME

For on-path hazards – Scheme 0

Figure 8 shows the flowchart of the incremental scheme for resolving on-path hazards during the pseudo register phase. Three subroutines *loop-protection*, *node-splitting*, and *replicate-loop*, marked by "*", may insert new nodes to loops. Information associated with each node, including register live range, stamp heap and loop structure, is updated locally whenever a node is inserted.

Assume that L_i's immediate parent loop is L_j, which may be the entire program, and $\{I_u, I_d\}$ is a hazard pair for register x. The two cases in which we consider protecting L_i from outside for x are shown in Figure 9(a) and (b). In Figure 9(a), since the x in I_d will be renamed, we only need to check if there is any other definition of x, I_ω, that can reach I_h, and is not in the same stamp heap as I_d. The search for I_d is restricted to the shaded area, denoting the definitions within L_j that can reach I_h without going through backedges, but I_ω can be nodes in the upper levels that can reach I_h. The hazard in Figure 9(b) can also be resolved by expanding L_i a sufficient number of times and renaming registers within L_i. For simplicity, we protect L_i from outside for register x instead, so that the hazard is automatically resolved.

Subroutines *renaming*, *live-analysis*, *record-loop-structure*, and *sort-loop* are executed only

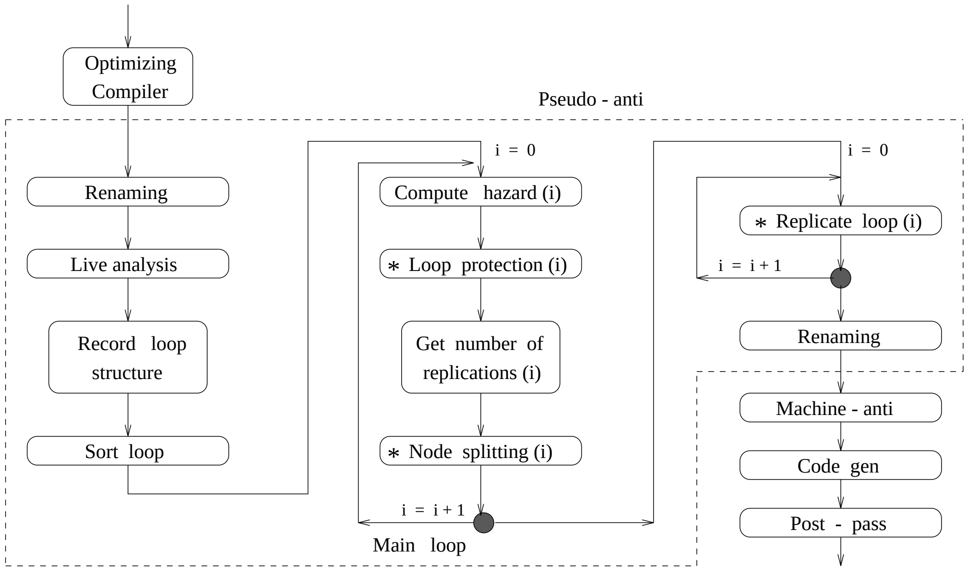


Figure 8. Incremental updating for resolving on-path hazards

once. The incremental scheme does not perform global DU-chain and global reaching definition analysis as Scheme L does, but rather performs a global live range analysis [23]. Loop structure and dataflow information $live\_in$ and $live\_out$ are maintained and updated locally throughout the computation.

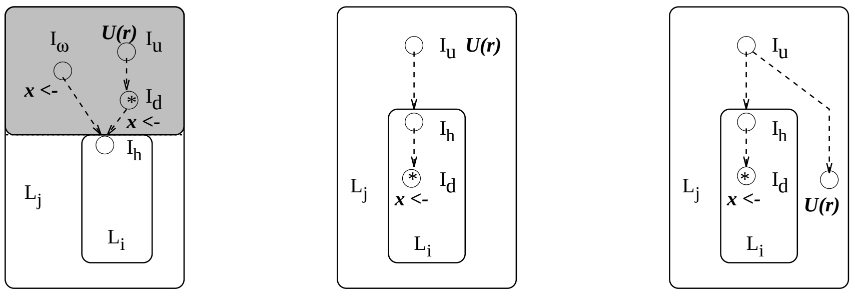
Subroutine *compute-hazard* computes all hazard registers and hazard nodes within the current loop, bypassing inner loop hazards. It traverses nodes within L_i from the loop header in a BFS order. If node I defines x, it performs an RBFS traversal from node I up to distance N, but the search never leaves L_i. If there is a use of x within distance N, it records x a hazard register, and I a hazard node. Subroutine *loop-protection* protects loop L_i according to conditions C1, C2, and C3. Subroutine *get-number-of-replications* performs a BFS traversal to compute $d(I_h, I_\beta)$ and an RBFS traversal to compute $d(I_\alpha, I_t)$ for all nodes I_α, I_β in L_i. It then computes K using the new formula, for every hazard pair $\{I_u, I_d\}$ in L_i. The maximum of all such values is the number of replications needed for L_i to resolve its hazards.

Subroutine *node-splitting* executes the criterion mentioned in the previous section, and scans the loop nodes in an MBFS order, bypassing the inner loops. Subroutine *replicate-loop* first marks the extended loop $\tilde{L}_i$ for all hazard registers, and then applies a BFS traversal to replicate $\tilde{L}_i$. The number of copies is obtained from *get-number-of-replications* subroutine.

As shown in Figure 8, each program loop is examined once. The actual code growth occurs after all loops have been inspected.

Incorporating branch hazards – Schemes 2 and 3

Branch hazards occur at branch boundaries when an error results in a wrong branch decision. The following criterion can be used to locate all branch hazards : Register x is a branch hazard if there exists a branch node I_{BR}, such that the distance from I_{BR} to a definition of x along

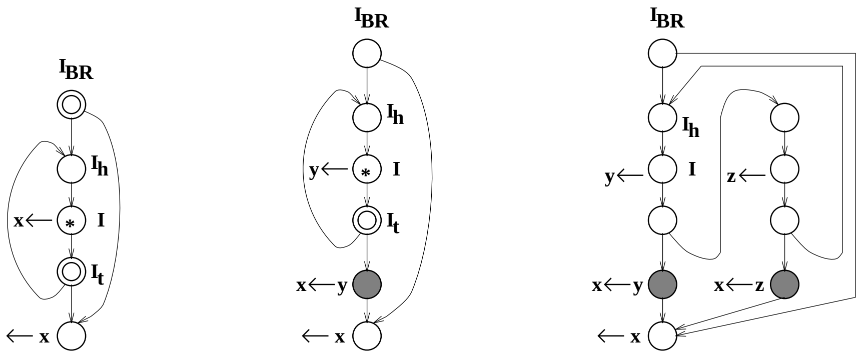


(a) Hazard splitting the loop header. (b) Across loop on-path hazard. (c) Across loop branch hazard.

Figure 9. The confinement of search nodes outside the loop

one branch path of I_{BR} is within N, and x is *live* at the other branch paths of I_{BR}. By viewing x as if it is used at I_{BR}, renaming can resolve branch hazards as well as on-path hazards. Similar to the case shown in Figure 9(b), we modify the loop protection conditions. As shown in Figure 9(c), I_u is a branch node that does not use register x, and x is live along one branch path of I_u. Loop L_i is protected from outside for register x, as if branch node I_u uses register x.

Example Consider the partial segment shown in Figure 10(a), and $N = 3$. Register x at node I is a branch hazard due to branch nodes I_{BR} and I_t, denoted by double circles. After loop protection as in Figure 9(c), and renaming x to y, the new register y at node I is a branch hazard due to branch node I_t, as shown in Figure 10(b). Note that the save instruction $y \leftarrow x$ before the loop header I_h is removed since x is not live at I_h. In Figure 10(c), by expanding the loop twice and renaming, the branch hazard is resolved. The formula for the number of loop replications can also be modified by viewing the branch node as using the hazard register x.



(a) A program segment. (b) After loop protection on x. (c) After expanding twice and renaming.

Figure 10. The loop expansion for branch hazards, $N = 3$

POST-PASS CODE RESCHEDULING AND SPILL REGISTER REASSIGNMENT

On-path hazards – Scheme 0

Although the pseudo register phase aims at removing on-path hazards within a function, new hazards may emerge after the machine register phase. First, the stack pointer adjustment instructions within the prologue segment and the epilogue segment create immediate self-anti-dependencies. Second, before calling a procedure, the registers used as parameters need to be saved before the new values can be loaded. Register spilling may also create on-path hazards. When a register is to be spilled, most likely it will be loaded with new values, thus creating a use-before-definition scenario. A straightforward post-pass nop insertion algorithm was employed in Scheme L to resolve these new hazards. Sufficient nops are inserted before the hazard definitions to force all anti-dependency distances exceeding N.

In this section, we apply a code rescheduling technique within the prologue and the epilogue segments, and a register reassignment algorithm for rearranging spill registers, so that the total number of nops inserted is greatly reduced. The post-pass algorithm includes the following steps : 1) reassign spill registers; 2) reschedule code and insert nops in the prologue segment; 3) reschedule code and insert nops in the epilogue segment; and 4) insert remaining nops.

The IMPACT C compiler [22] reserves three registers, \$3, \$24, and \$25, as spill registers. The spill registers perform *load* and *store* to access memory. The compiler generates instructions of the following groups for load and store functions respectively, where \$$r_1$ and \$$r_2$ are different spill registers, and are dead after the second (or the third) instruction :

```
load $r_1, memory;        load $r_1, memory_1;        operation defining $r_1;
use $r_1;                 load $r_2, memory_2;        store $r_1, memory;
                          use $r_1, $r_2;
```

Spill registers serve as temporaries and have very short live ranges, i.e., 2 or 3. On-path hazards occur when two groups of spill code use the same spill register and their distance, from the use of the first group to the definition of the second group, is less than or equal to N. The goal is to minimize the number of nops needed to resolve all hazards. Our approach is to utilize dead registers as substitutes within groups so that the sum of all the anti-dependency distances for spill registers and substitutes is maximized, considering the anti-dependency distance between groups of different spill registers and substitutes $N + 1$. In general, this problem is NP-hard, which includes as a special case the following NP-complete problem after determining that only spill registers are dead registers, and $N = 1$:

Given K colors, an undirected graph G and an integer n, is there a node coloring such that the number of edges with the same colors at both ends is at most n?

This can be proven by restricting n to 0, and it becomes the K-colorability problem [25]. However, we propose a simple heuristic algorithm to reassign spill registers within groups in a BFS traversal of the entire program. We always choose as a substitute the register which is dead before and after the group, and whose sum of the distance backward to the first use and the distance forward to the first definition is maximum.

The prologue segment includes code to adjust the stack pointer and to save the values of local registers to memory. The epilogue segment includes code to retrieve the original register values from memory and to adjust the stack pointer. The last step simply performs a BFS traversal, and inserts nops to resolve all remaining on-path hazards.

```
merge_sort :       :
                   :
                   :
$_merge_sort_3 :
epilogue_begin :

        10   nops

     lw      $16,     92($sp)
     lw      $17,     96($sp)
     lw      $18,     100($sp)
     lw      $19,     104($sp)
     lw      $20,     108($sp)
     lw      $21,     112($sp)
     lw      $22,     116($sp)
     lw      $23,     120($sp)
     lw      $31,     124($sp)
     addu     $30,     $sp,     128

        10   nops

     move    $sp,     $30
epilogue_end :     :
                   :
     beq     $16,     $20,     $_merge_sort_3
                   :
     beq     $17,     $23,     $_merge_sort_3
                   :
```

(a) Scheme L.

```
merge_sort :       :
                   :
                   :
$_merge_sort_3 :
epilogue_begin :
     addu     $30,     $sp,     128
     move    $0,      $0
     move    $0,      $0
     move    $0,      $0
     move    $0,      $0
     lw      $21,     -16($30)
     lw      $31,     -4($30)
     lw      $22,     -12($30)
     lw      $19,     -24($30)
     lw      $18,     -28($30)
     lw      $23,     -8($30)
     lw      $20,     -20($30)
     lw      $17,     -32($30)
     lw      $16,     -36($30)
     move    $sp,     $30
epilogue_end :     :
                   :
     beq     $16,     $20,     $_merge_sort_3
                   :
     beq     $17,     $23,     $_merge_sort_3
                   :
```

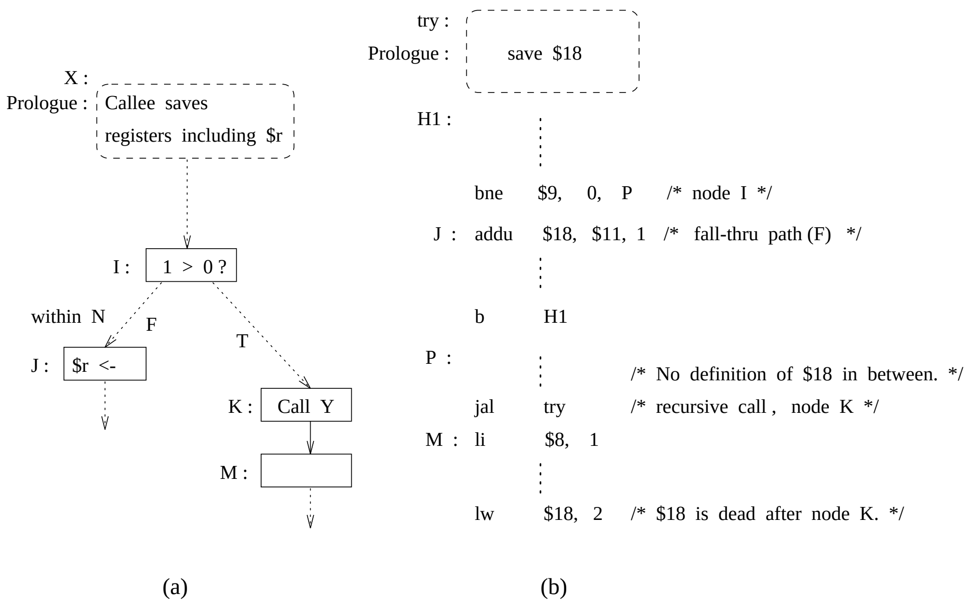
(b) The new scheme.

Figure 11. Post-pass code rescheduling for an epilogue segment, $N = 10$

Example Figure 11(a) shows the epilogue segment processed by Scheme L in post pass, for $N = 10$. Figure 11(b) illustrates how the register assignment and code rescheduling are used to eliminate 16 nops in the epilogue segment. Instruction 'addu $30, $sp, 128' has been moved backward up to before all instructions of loading local registers, with the base register being replaced by $30. The instructions to load local registers are rescheduled according to their distances from the first uses of corresponding registers. The four instructions loading registers $16, $17, $20, and $23 are thus moved to the end of the load instructions. Four more nops are needed to resolve the hazard register $23.

Both types of hazards – Schemes 1, 2, and 3

Post-pass nop insertion can also resolve extra branch hazards generated by the machine register allocator. The branch hazard check can be incorporated in the original on-path hazard check. The heuristic to reassign spill registers has to be modified as follows. The register we choose to replace the reserved spill register at a specific group G of spill instructions must be not only dead before and after G, but also requires as few nops as possible to resolve the new branch hazard induced by the substitute register.



```
try :
Prologue :    save  $18

H1 :
              ⋮

        bne     $9,   0,   P     /* node I */
  J :   addu    $18,  $11, 1     /* fall-thru path (F) */
              ⋮

        b       H1
  P :
              ⋮                  /* No definition of $18 in between. */
        jal     try              /* recursive call , node K */
  M :   li      $8,   1
              ⋮

        lw      $18,  2          /* $18 is dead after node K. */
```

Figure 12. Register live range across procedure boundaries

The above schemes for incorporating branch hazard resolution do not create extra hazards across procedural boundaries. However, depending on implementations, the callee-saved registers may have a performance impact due to separate compilations. As shown in Figure 12(a), suppose at branch node I, a wrong decision is made. After rollback and a correct decision at I, register $\$r$ has a wrong value. If $\$r$ is in Y's callee-saved register set, then $\$r$ is live along I's target (T) branch. Several nops should be inserted between I and J to resolve such branch hazard. However, since Y's callee-saved register set are unknown at current procedure X, a conservative scheme may assume that the registers are all in the set, e.g., \$16, \$17, $\cdots$, \$23 in IMPACT C. By viewing K as a node that uses such set, we can incorporate it in the initial global live range analysis.

For library routines, a built-in table holding corresponding saved register sets can be attached to the compiler to relieve the situation described above. The following checking can determine $\$r$'s live range before the procedure call, regardless of whether $\$r$ belongs to the callee-saved register set. $\$r \in live\_in(M)$ *iff* $\$r$ is live at node K, where M is the next instruction following the subroutine call node K. Such live range checking starting from M should skip any subroutine call encountered.

Example Figure 12(b) is an assembly code segment for the recursive function *try* . Without checking the additional condition, N nops are inserted between node I and node J to eliminate the hazard \$18. None is required by observing \$18 is dead after node K. Code run time performance is improved since such N nops are within a loop.

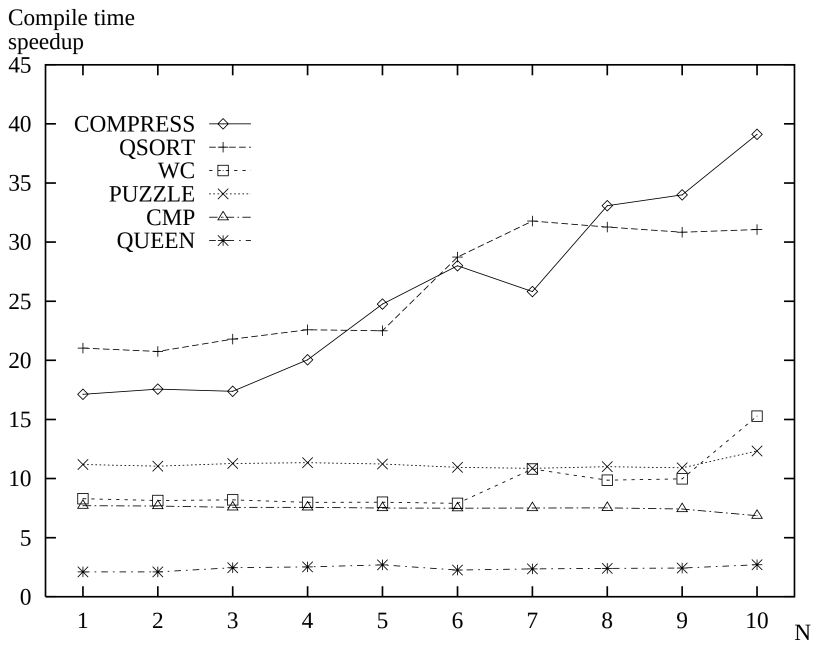


Figure 13. Compile time speedup

PERFORMANCE EVALUATION

Implementation and performance benefits of the schemes are evaluated on a set of twelve programs cross-compiled on a SPARC server 490 by the IMPACT C compiler with the hazard removal schemes, and executed on a DEC station 3100. The benchmarks and descriptions are as follows: QUEEN(148), 8-queen program; QSORT(261), recursive quick sort algorithm; PUZZLE(877), a game; WC(181), CMP(251), GREP(926), COMPRESS(1828), UNIX utilities; EQN(6251), mathematics typesetting program; LEX(6873), lexical analyzer; YACC(8099), parser generator; CCCP(8775), preprocessor for gnu C compiler; and TBL(9191), table formatter. The number within the parentheses is the number of instructions generated by the IMPACT C compiler without removing hazards. The chosen benchmarks consist of a variety of typical program constructs including sequential single loops, highly nested loops, and recursive functions.

Resolving on-path hazards – Scheme 0 v.s. Scheme L

The incremental updating scheme and the post-pass code rescheduler improve application compile time, run-time performance, and reduce code growth for most applications studied. In this section we compare the performance impact of Scheme 0 and Scheme L with respect to the compile time, code run time and code size. We investigate the same set of benchmarks used in [1]: CMP, COMPRESS, PUZZLE, QSORT, QUEEN, and WC.

For $N = 10$, Scheme L requires more than 8 minutes, 15 seconds, 1.5 minutes, 3.5 minutes, and 9.5 minutes compiling QSORT, QUEEN, CMP, WC, and PUZZLE respectively, while Scheme 0 takes compile time less than 16 seconds, 8 seconds, 15 seconds, 15 seconds and 50

Table II. Code run time overhead for Schemes L and 0

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	6.2%	8.3%	8.3%	10.4%	11.5%	13.5%	14.6%	26.0%	22.9%	30.2%
	0	5.2%	6.2%	6.2%	8.3%	8.3%	10.4%	10.4%	13.5%	15.6%	16.7%
QUEEN	L	3.0%	5.3%	7.2%	7.2%	9.0%	9.8%	11.5%	15.8%	16.3%	20.9%
	0	2.9%	3.5%	3.9%	4.9%	5.1%	5.5%	6.0%	8.0%	10.2%	16.3%
CMP	L	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%
	0	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%
WC	L	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	4.4%
	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.3%	1.3%
PUZZLE	L	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%
	0	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	0.0%	0.0%
COMPRESS	L	-0.6%	0.0%	0.0%	0.0%	1.2%	2.5%	5.6%	6.2%	11.2%	18.8%
	0	-0.6%	-0.6%	-0.6%	-0.6%	1.2%	1.2%	5.0%	5.6%	10.6%	16.9%

seconds respectively. COMPRESS has the best compile time improvement. Scheme L spends more than an hour for $N = 7$, 8, and 9, and almost two hours for $N = 10$ on compilation, while Scheme 0 compiles in less than 3 minutes. Figure 13 shows the compile time speedup which is the compile time ratio between Scheme L and Scheme 0.

Table II lists code run time overhead for both Schemes L and 0 respectively. The base of comparison is the original code run time. Some benchmarks, e.g., CMP and PUZZLE, have improved performance, as shown by negative numbers. The register allocator, nop inserter, and spill register reassignment involve heuristic algorithms that in some cases provide improved performance under loop expansion. The MIPS post-pass code reorganizer also sometimes changes the execution order for different N. Two benchmarks, QSORT, and QUEEN, include recursive functions and have among the largest run-time enhancements, for $N > 5$. Post-pass code rescheduling is a significant contributor to these two benchmarks.

Table III lists the code size overhead for Schemes L and 0. The base of comparison is the number of instructions in the original code. COMPRESS has larger code growth in Scheme 0 due to the removal of the 800 instruction threshold [1] and the change in the number of functions compiled in simplified mode which bypasses the rest of pseudo register hazard resolution except the breaking of self-anti-dependent instructions, after exceeding the threshold. In

Table III. Code size overhead for Schemes L and 0

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	63%	70%	105%	115%	123%	136%	154%	199%	219%	274%
	0	101%	104%	105%	110%	118%	130%	138%	146%	169%	191%
QUEEN	L	57%	69%	124%	134%	152%	164%	176%	208%	219%	310%
	0	48%	53%	58%	68%	78%	127%	132%	147%	151%	179%
CMP	L	75%	80%	92%	107%	120%	141%	158%	179%	200%	228%
	0	60%	63%	67%	76%	82%	84%	88%	90%	94%	122%
WC	L	133%	138%	160%	167%	179%	216%	245%	249%	257%	290%
	0	153%	155%	160%	163%	164%	165%	187%	205%	209%	244%
PUZZLE	L	80%	80%	87%	89%	91%	94%	96%	101%	106%	126%
	0	79%	79%	81%	84%	85%	87%	96%	99%	101%	111%
COMPRESS	L	28%	32%	38%	52%	60%	69%	80%	94%	107%	129%
	0	70%	73%	74%	78%	82%	87%	108%	122%	152%	156%

Table IV. Run time overhead for Scheme 1

N	1	2	3	4	5	6	7	8	9	10
QSORT	6.2%	6.2%	7.3%	9.4%	9.4%	12.5%	12.5%	16.7%	18.7%	18.7%
QUEEN	2.8%	3.1%	4.1%	5.7%	6.3%	6.7%	7.4%	11.1%	11.2%	18.0%
CMP	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-2.4%	-1.8%	-1.2%	-1.2%
WC	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%
PUZZLE	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	0.7%	0.7%
COMPRESS	1.3%	2.0%	2.6%	4.0%	7.3%	9.3%	9.9%	11.3%	13.9%	17.9%
GREP	11.1%	11.1%	11.1%	11.1%	11.1%	13.0%	13.0%	13.0%	14.8%	24.1%
LEX	10.5%	11.6%	11.6%	11.6%	11.6%	11.6%	12.8%	14.0%	14.0%	18.6%
EQN	7.8%	11.3%	12.2%	12.2%	12.2%	12.2%	13.9%	13.9%	13.9%	13.9%
YACC	0.0%	0.0%	2.4%	2.4%	2.4%	*7.1%	*11.9%	*16.7%	*23.8%	*28.6%
CCCP	8.5%	9.3%	10.1%	11.6%	11.6%	*17.1%	*17.1%	*19.4%	*20.9%	*26.4%
TBL	5.3%	7.9%	7.9%	7.9%	7.9%	7.9%	14.5%	14.5%	14.5%	15.8%

Scheme 0, QSORT and WC have larger code growth when $N = 1$ and 2. Loop expansion is the major stage that results in most of the code growth. In Scheme L proper renaming after protecting the loop from inside and node splitting for small N may prevent the loop from being expanded. Also, if the loop is protected for several registers from inside, the hazards can be removed after arranging the order of the save/restore nodes, and renaming without actually expanding the loop. However, using the cut hazard register technique, as in Scheme 0, to move save/restore nodes out of the loop L requires L to be expanded at least once.

Resolving on-path and branch hazards – Schemes 1, 2, and 3

Schemes 1, 2, and 3 deal with removing both types of hazards during three separate phases. Scheme 1 has the fastest compilation speed since it postpones the branch hazard resolution to the last phase.

All three schemes perform relatively the same for the twelve benchmarks studied. Reasons for this behavior include 1) the occurrences of branch hazards are infrequent; 2) both machine register and nop insertion phases employ heuristics, and the spill register reassignment heuristic may be efficient enough to resolve branch hazards in the post-pass; and 3) resolving branch hazards at the pseudo register phase or the machine register phase is likely to have larger code growth, due to the extra node splitting and loop expansion. In most benchmarks, Scheme 1 even outperforms the other two schemes in both code run-time and code growth.

The performance overhead of Scheme 1 is tabulated in Table IV. Due to the heuristic algorithm employed in the post-pass phase, the performance overhead observed is not monotonically increasing according to N. However, the code generated to allow N instruction rollback is correct for $N - 1$ instruction rollback as well. Therefore, the overhead can be recorded as non-decreasing. Several functions generate more than 15,000 nodes, which increases the computation time for the machine register assignment phase, when $N > 6$. YACC has two such functions, and CCCP has one. For these three functions, we resolve the rollback hazards of distance 5 in the pseudo register phase, and then resolve the rollback hazards of distance $N > 5$ in the post-pass phase, as marked by "*" in Table IV.

Figure 14 depicts the percentage of hazard nodes that are branch hazard nodes but are not on-path hazard nodes, for various rollback distances. Benchmarks QUEEN and QSORT have 0 percentage for N within 10 because either they have no branch hazards, or all of their branch hazards are also on-path hazards. PUZZLE has the highest percentage of branch hazard

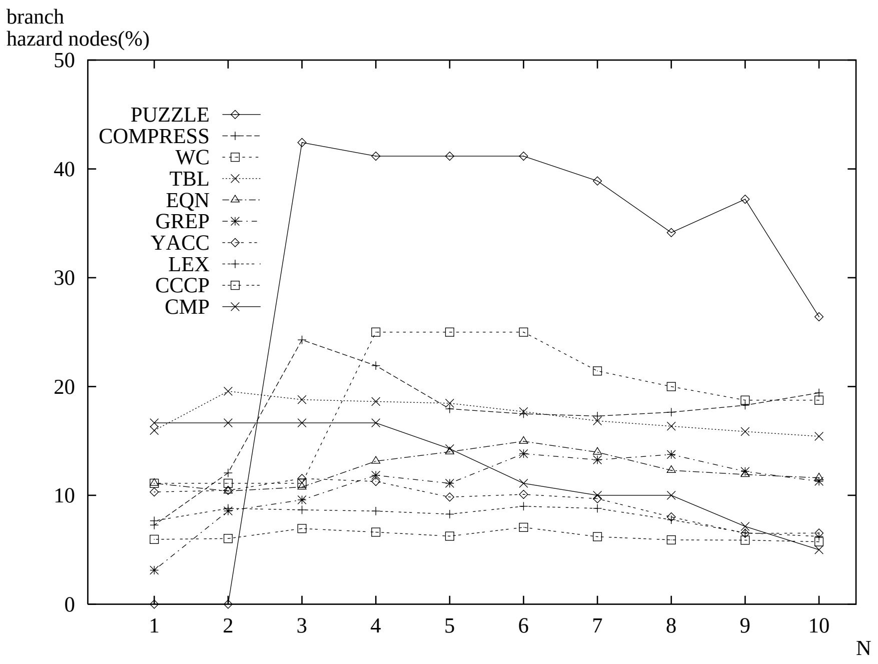


Figure 14. Percentage of the hazard nodes that are branch hazard nodes

nodes, 42.42% when $N = 3$. There is a significant rise from $N = 2$ to $N = 3$ due to the relative distances between branch nodes and hazard nodes. This can explain why in Scheme A, PUZZLE has the highest run-time overhead 10% when $N = 10$ [20]. The post-pass algorithms apparently reduce the overhead to 0.7%, as shown in Table IV. All the other benchmarks have less than a quarter of the hazard nodes that are branch hazard nodes but not on-path hazard nodes.

CONCLUSION

The previous compiler-based multiple instruction retry resolves data hazards associated with rollback recovery [1]. The proposed approach eliminates the delayed write buffer for the register file, but suffers from long compilation times. An incremental updating scheme has been incorporated in the compiler-based scheme, resulting in significantly reduced compile times. The code in the prologue and the epilogue segments was rescheduled, and the spill registers were reassigned to reduce the total number of nops inserted. The threshold for the number of nodes increased from 800 to 15,000. Branch hazards were shown to be resolvable by simple modifications to the proposed approaches. Three schemes were implemented and compared. Postponing the resolution of branch hazards to the last phase was shown to provide the fastest compilation. This approach also typically generated code as good as the alternatives in both code run time and code growth.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for the time and effort they spent on reviewing this paper. This research was supported in part by the National Aeronautics and Space Administration (NASA) under grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under grant N00014-91-J-1283.

REFERENCES

1. C-C. J. Li, S-K. Chen, W. K. Fuchs, and W-M. W. Hwu, 'Compiler-assisted multiple instruction retry', Technical Report CRHC-91-31, Coordinated Science Laboratory, University of Illinois, 1991, to appear in *IEEE Transactions on Computers*.
2. X. Castillo, S. R. McConnel, and D. P. Siewiorek, 'Derivation and calibration of a transient error reliability model', *IEEE Transactions on Computers*, **C-31**(7), 658–671 (1982).
3. R. Iyer and D. Rossetti, 'A measurement-based model for workload dependence of CPU errors', *IEEE Transactions on Computers*, **C-35**(6), 511–519 (1986).
4. L. Svobodova, 'Resilient distributed computing', *IEEE Transactions on Software Engineering*, **SE-10**(3), 257–268 (1984).
5. L. Lin and M. Ahamad, 'Checkpointing and rollback-recovery in distributed object based systems', *The Twentieth International Symposium on Fault-Tolerant Computing*, 1990, pp. 97–104.
6. K. Tsuruoka, A. Kaneko, and Y. Nishihara, 'Dynamic recovery schemes for distributed processes', *IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, 1981, pp. 124–130.
7. C.-C. J. Li and W. K. Fuchs, 'CATCH - Compiler-assisted techniques for checkpointing', *The Twentieth International Symposium on Fault-Tolerant Computing*, June 1990, pp. 74–81.
8. W-M. W. Hwu and Y. N. Patt, 'Checkpoint repair for high-performance out-of-order execution machines', *IEEE Transactions on Computers*, **C-36**(12), 1496–1514 (1987).
9. M. L. Ciacelli, 'Fault handling on the IBM 4341 processor', *The Eleventh International Symposium on Fault-Tolerant Computing*, June 1981, pp. 9–12.
10. 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*, **26**(1), 2–11 (1982).
11. W. F. Bruckert and R. E. Josephson, 'Designing reliability into the VAX 8600 System', *Digital Technical Journal of Digital Equipment Corporation*, **8**, 71–77 (1985).
12. D. B. Fite, T. Fossum, and D. Manley, 'Design strategy for the VAX 9000 system', *Digital Technical Journal of Digital Equipment Corporation*, **8**, 13–24 (1990).
13. P. M. Kogge, K. T. Truong, D. A. Richard, and R. L. Schoenike, 'Checkpoint retry mechanism'. United States Patent, no. 4912707, Mar. 1990, Assignee: International Business Machines Corporation, Armonk, N.Y.
14. G. L. Hicks, D. Howe Jr., and A. Zurla Jr., 'Instruction retry mechanism for a data processing system'. United States Patent, no. 4044337, Aug. 1977, Assignee: International Business Machines Corporation, Armonk, N.Y.
15. J. E. Smith and A. R. Pleszkun, 'Implementing precise interupts in pipelined processors', *IEEE Transactions on Computers*, **C-37**(5), 562–573 (1988).
16. Y. Tamir and M. Tremblay, 'High-performance fault-tolerant VLSI systems using micro rollback', *IEEE Transactions on Computers*, **C-39**(4), 548–554 (1990).
17. Y. Tamir, M. Liang, T. Lai, and M. Tremblay, 'The UCLA mirror processor: A building block for self-checking self-repairing computing nodes', *The Twenty-First International Symposium on Fault-Tolerant Computing*, June 1991, pp. 178–185.
18. L. Spainhower, J. Isenberg, R. Chillarege, and J. Berding, 'Design for fault-tolerance in system ES/9000 model 900', *The Twenty-Second International Symposium on Fault-Tolerant Computing*, July 1992, pp. 38–47.
19. J. S. Liptay, 'Design of the IBM enterprise system/9000 high end processor', *IBM Journal of Research and Development*, **36**(4), 713–731 (1992).
20. 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', *The Twenty-Second International Symposium on Fault-Tolerant Computing*, July 1992, pp. 66–73.

21. D. A. Padua and M. J. Wolfe, ‘Advanced compiler optimizations for supercomputers’, *Communications of the ACM*, **29**(12), 1184–1201 (1986).
22. P.P Chang, W.Y. Chen, N.J. Warter, and W-M. W. Hwu, ‘IMPACT: An architecture framework for multiple-instruction-issue processors’, *The 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 266–275.
23. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
24. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.
25. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, 1979.