A Study of Control Independence in Superscalar Processors

Eric Rotenberg, Quinn Jacobson, Jim Smith
University of Wisconsin - Madison
ericro@cs.wisc.edu, {qjacobso, jes}@ece.wisc.edu

Abstract

An instruction is control independent *of a preceding conditional branch if the decision to execute the instruction does not depend on the outcome of the branch -- this typically occurs if the two paths following the branch re-converge prior to the control independent instruction. A speculative instruction that is control independent of an earlier predicted branch does not necessarily have to be squashed and re-executed if the branch is predicted incorrectly. Consequently, control independence has been put forward as a significant new source of instruction level parallelism in future generation processors. However, its performance potential under practical hardware constraints is not known, and even less is understood about the factors that contribute to or limit the performance of control independence.*

A study of control independence in the context of superscalar processors is presented. First, important aspects of control independence are identified and singled out for study, and a series of idealized machine models are used to isolate and evaluate these aspects. It is shown that much of the performance potential of control independence is lost due to data dependences and wasted resources consumed by incorrect control dependent instructions. Even so, control independence can close the performance gap between real and perfect branch prediction by as much as half.

Next, important implementation issues are discussed and some design alternatives are given. This is followed by a more detailed set of simulations, where the key implementation features are realistically modeled. These simulations show typical performance improvements of 10 to 30 percent over a baseline superscalar processor.

Keywords: control dependences, selective squashing, branch prediction, speculation, ILP

1. Introduction

In order to expose instruction-level parallelism in sequential programs, dynamically scheduled superscalar processors form a “window” of fetched instructions. Each cycle, the processor selects and issues a group of independent instructions from this window. Maintaining a sufficiently large window of instructions is essential for high instruction-level parallelism -- the more instructions in the window, the greater the chance of finding independent ones for parallel execution.

Branch instructions are a major obstacle to maintaining a large window of useful instructions because they introduce *control dependences* -- the next group of instructions to be fetched following a branch instruction depends on the outcome of the branch. Typically, high performance processors deal with control dependences by using branch prediction. Then instruction fetching and speculative issue can proceed despite unresolved branches in the window. Unfortunately, branch mispredictions still occur, and current superscalar implementations squash all instructions after a mispredicted branch, thereby limiting the effective window size. Following a squash, the window is often empty and several cycles are required to re-fill it before instruction issuing proceeds at full efficiency. Furthermore, we are fast approaching the point where the hardware window that can be constructed exceeds the average number of instructions between mispredictions.

There are three ways of dealing with the conditional branch problem. The first, and most widely studied, is to improve branch prediction. This approach has received considerable (successful) research effort for nearly two decades. The second is to fetch and execute both paths following a branch, and keep only the computation of the correct path. Of course this can lead to exponential growth in hardware, so recently, more selective approaches have been advocated, where multi-path execution is only used for hard-to-predict branches [2, 3, 4, 5, 6, 7]. Predicated execution is a software method for achieving a similar effect [8, 9]. The third approach is aimed at reducing the penalty after a misprediction occurs. This approach exploits the fact that not all instructions following a mispredicted branch have performed useless computation.

The third approach is probably less well understood than the other two, and in this paper we explore its potential. The key point is that only a subset of dynamic instructions immediately following the branch may truly depend on the branch outcome. These instructions are *control dependent* on the branch. Other instructions deeper in the window may be *control independent* of the mispredicted branch: they will be fetched regardless of the branch outcome, and do not necessarily have to be squashed and re-executed [10, 11]. This can be illustrated with a simple example.

Figure 1 shows a control flow graph (CFG) containing four basic blocks. (Basic blocks are used for simplicity and, in general, may be substituted with arbitrary control flow.) The conditional branch terminating block 1 is mispredicted, with dashed arrows indicating the mispredicted path 1, 2, and 4. Two data dependences, through registers r4 and r5, are also shown.

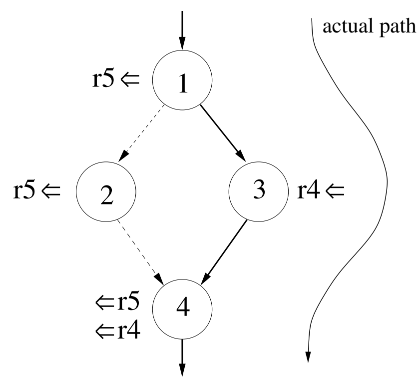


FIGURE 1. An example of control independence.

At the time the misprediction is detected, blocks 1, 2, and 4 have already been speculatively fetched and some of their instructions may have already started executing. Because only block 2 is control dependent on the misprediction, it is the only block whose instructions must be squashed. Immediately after the misprediction is found, the fetch unit goes back and fetches block 3 to replace the squashed instructions of block 2.

Control independent instructions following the mispredicted branch, specifically block 4, are not squashed, but they do need to be inspected for data dependence violations caused by the mispredicted control flow, and some instructions may have to be re-executed. The value identified with r5 must be corrected so that block 4 uses the value produced earlier in block 1 instead of the one incorrectly produced in block 2. Likewise, when block 3 is eventually inserted into the window, the data dependence through register r4 must also be established. Note that data dependences through memory must similarly be repaired. After the instructions using r4 and r5 in block 4 correct their data dependences and reissue, all subsequent data dependent instructions must also reissue. Hence, selective instruction reissue [12, 1] in some form is necessary.

Lam and Wilson's limit study on control independence [10] showed that substantial performance improvements may be possible. However, as a limit study, most implementation constraints were not considered. Further, important aspects of programs themselves were not modeled; in particular, a significant subset of data dependences were ignored due to the trace-driven nature of the study. Several microarchitecture implementations have since been proposed that incorporate control independence in some form [11, 13, 14, 15, 16, 17, 18, 1]. In these studies, however, either the impact of control independence is not isolated, or insight into the reported performance gains is limited and obscured by artifacts of the particular design.

In this paper we have three primary objectives and contributions. The first objective is to *establish new bounds on the performance potential of control independence under implementation constraints*. The study focuses on two fundamental constraints that characterize superscalar processors: instruction window size and instruction fetch/issue bandwidth. Other aspects of the study remain ideal and aggressive to avoid design artifacts that might obscure the analysis.

The second objective is to *provide insight into the factors that contribute to or limit the performance of control independence*. Data dependences between control dependent and control independent instructions play an important role. In Figure 1, there is a **true data dependence** (register r4) between the **correct control dependent instructions** in block 3 and subsequent control independent instructions in block 4. Similarly, there is a **false data dependence** (register r5) produced by the **incorrect control dependent instructions** in block 2. Resolving both types of data dependences is delayed by the branch misprediction in spite of control independence. Another important factor is the waste of fetch and execution resources by incorrect control dependent instructions. Having to first fetch the misspeculated instructions delays filling the instruction window with correct, control independent instructions. Also, if there are more incorrect control dependent instructions than correct ones, e.g. block 2 is larger than block 3, window space is wasted that might have gone to more control independent instructions.

The third objective is to *assess the complexity of implementing aggressive control independence mechanisms in superscalar processors*. Although it is beyond the scope of this paper to put forth detailed designs, implementation requirements are identified and hardware/software alternatives for meeting the requirements are proposed. We have also developed a detailed execution-driven simulator that implements the outlined requirements.

Several conclusions emerge from our study. First, the performance gap between branch prediction with conventional speculation and oracle branch prediction is quite large, but control independence holds the potential for closing the gap by as much as half. Second, the effects of incorrect control dependent instructions -- both wasted resources and false data dependences -- significantly limit the benefits of control independence, with wasted resources being the chief problem. The impact of true data dependences is slightly smaller than that of false data dependences. Third, for the chosen design alternatives in the detailed execution-driven model, performance improvements ranging from 10% to 30% are measured.

In order to keep the study manageable, we limit our scope to one of two major schemes for exploiting control independence. In particular, the study targets processors that use a single flow of control, i.e. a single fetch unit, as in today's superscalar processors. Other schemes, using multiple flows of control, are not studied here, although extending the study of control independence to multiple (yet finite) fetch units is an interesting problem to be explored.

1.1 Prior work

Lam and Wilson's limit study [10] demonstrates that control independence exposes a large amount of instruction-level parallelism, on the order of 10 to 100, for control-intensive integer benchmarks. Although these results are important, full interpretation is obscured for both technical and practical reasons. As pointed out in an analysis by Sundararaman and Franklin [19], the limit study makes certain assumptions that may inflate the apparent benefits of control independence. Static branch prediction based on profiling is used, as opposed to higher accuracy dynamic branch predictors. More importantly, because the simulation is fully trace-driven, it does not account for false data dependences created on mispredicted paths (as discussed previously), thus allowing incorrect-data dependent instructions to be scheduled earlier than they would be in practice. Furthermore, limit studies, by definition, are unconstrained in order to measure *inherent parallelism* in programs, and do not consider practical implementation issues. In the Lam and Wilson limit study, several fundamental features of processors are not modeled. In particular, there is no concept of a limited instruction window or instruction fetch bandwidth, whether considering a single or multiple flows of control. The limit study schedules the entire dynamic instruction stream at once; exposing the observed parallelism may require buffering speculative state for thousands of instructions and using an impractical number of parallel fetch units.

Another unconstrained limit study by Uht and Sindagi [2] uses a similar simulation approach, but in addition to studying "minimal control dependences", a form of selective eager execution called disjoint eager execution is also studied.

Multiscalar processors [11,13] and other multithreaded architectures [16, 17, 14, 15] exploit control independence by pursuing multiple flows of control. In the case of multiscalar, the compiler partitions the program into tasks, or subgraphs of the CFG. Arbitrary control flow may exist within a task, and the compiler need not guarantee that tasks be control and data independent. At run-time, a task sequencer predicts and allocates tasks to run on distributed processing elements, each capable of pursuing its own flow of control. In this way, branch mispredictions within a task may not cause subsequent tasks to squash if they are control independent of the branch. To date, however, there has been no study that separates the impact of control independence and determines its contribution to performance in the multiscalar paradigm.

Trace processors [20,1] are in some sense a variant of multiscalar processors where the dynamic instruction stream is divided into traces -- frequently executed dynamic instruction sequences. An internal mispredicted conditional branch causes its trace to be squashed, but subsequent traces are not squashed if, after repairing the mispredicted branch and predicting a new sequence of traces, the new traces match those already residing in the processing elements [1]. Only modest improvements are reported because no optimization in trace selection or processor assignment was done to enhance performance benefits of control independence.

The instruction reuse buffer [18] provides another way of exploiting control independence. It saves instruction input and output operands in a buffer -- recurring inputs can be used to index the buffer and determine the matching output; i.e. the instruction outputs are "reused". In the proposed superscalar processor with instruction reuse, there is complete squashing after a branch is mispredicted. However, control independent instructions after the squash can be quickly evaluated via the reuse buffer. Overall speedups due to reuse are on the order of 10%, over half of which is due to squash reuse.

1.2 Paper organization

In Section 2, we consider a series of idealized machine models in order to better understand the relative importance of some of the bigger issues affecting control independence. Section 3 lists the key features in a superscalar processor for exploiting control independence and discusses implementation alternatives for each of the features. Next, in Section 4, we study performance considering timing constraints imposed by practical implementations.

2. The potential of control independence

In this section we begin evaluating the performance potential of control independence in superscalar processors. It is an idealized study in the sense that some of the models have oracle knowledge so that (1) performance bounds can be established and (2) aspects that limit the performance of control independence can be isolated. The latter has important implications: by understanding the limiting aspects, techniques may be developed to overcome them. On the other hand, the study is *not* an unconstrained "parallelism limit study" -- a particular class of implementations is targeted, and some of the basic resources are limited.

2.1 Control independence models

In the models given below, the performance impact of three important aspects of a control independent design are singled out for study.

- The first aspect concerns true data dependences between correct control dependent instructions and control independent instructions. In such cases, issuing the control independent instructions is delayed until after the misprediction is resolved and the correct control dependent instructions are fetched/issued.
- The second aspect is the handling of false data dependences created by incorrect control dependent instructions. As discussed earlier, these cause the selective reissue of some control independent instructions. Delays brought on by this repair and selective reissue can inhibit performance gains.
- The third aspect is the use of machine resources by instructions on an incorrect path that are eventually squashed. Even if control independence is ideally implemented otherwise, this waste of resources and time will reduce performance.

Six different models are evaluated. Figure 2 illustrates the differences among these six models, using the example CFG in Figure 1. Only two resources, instruction fetch and issue, are shown. Time progresses downward in the fetch/issue schedules. Fetching each basic block consumes fetch bandwidth; this is shown using basic block labels within their respective fetch slots. Likewise, instructions consume issue bandwidth, and are labeled first with the corresponding basic block, followed by the production/consumption of a value. For clarity, only instructions that ultimately retire (i.e. correct instructions) are shown; for these, only the final issue time is shown. The labels "M" and "D" in the diagrams indicate the time of the branch misprediction (M) and the time that the misprediction is detected (D).

The *oracle* model (Figure 2(a)) uses oracle branch prediction and therefore the branch terminating block 1 is not mispredicted. Blocks 1, 3, and 4 are fetched in correct dynamic program order.

The next four models use real branch prediction coupled with complete knowledge of control dependences to exploit control independence. The following notations are used.

- *WR* (“Wasted Resources”): Misspeculated instructions consume window resources and bandwidth, thus delaying other, correct instructions.
- *FD* (“False Data Dependences”): The effects of false data dependences between incorrect control dependent instructions and control independent instructions are modeled.

The inverse notations, *nWR* and *nFD*, indicate the corresponding factor is *not* modeled. Thus, there are four possible models: *nWR-nFD*, *nWR-FD*, *WR-nFD*, and *WR-FD*.

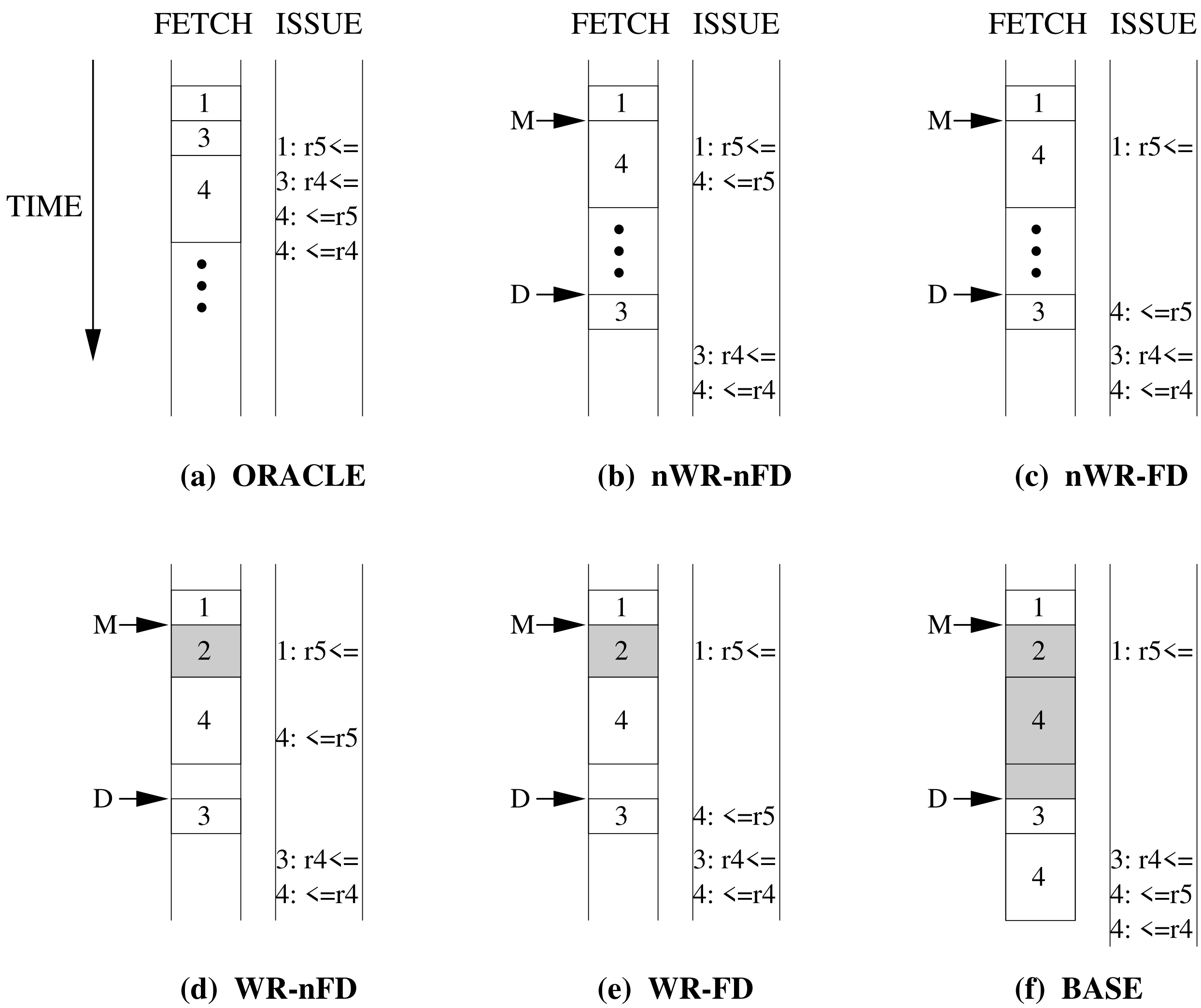


FIGURE 2. Fetch and issue timing for the six models, corresponding to the example CFG in Figure 1.

In the *nWR-nFD* model (Figure 2(b)), mispredicted branches delay fetching the correct control dependent instructions. But between the time that a branch is mispredicted and the misprediction is detected, fetch and window resources are kept busy with control independent instructions. Incorrect control dependent instructions are not considered (for example, block 2 is not fetched into the window), thereby eliminating false dependences and devoting resources solely to control independent work while the misprediction is resolved.

The only difference between this model and *oracle* is that instructions are fetched in a different order following mispredicted branches. This has a negative performance impact only when true data dependences are delayed with respect to *oracle*. For example, instruction “4: <=r4” issues later because the producer instruction in block 3 is delayed by the misprediction.

Interestingly, there are situations where performance of *nWR-nFD* may actually exceed that of *oracle*. For example, instruction “4: <=r5” issues slightly earlier with respect to *oracle*, because block 4 is fetched out-of-order and earlier. If this instruction is on the critical path, scheduling it earlier may improve overall performance.

The *nWR-FD* model, shown in Figure 2(c), also does not waste time with misspeculated instructions, however their effects on data dependences are felt. For example, we do not know the true producer of “r5” until the misprediction is resolved, delaying instruction “4: <=r5” until that time. The repair of false data dependences is assumed to occur in a single cycle, at the time a misprediction is resolved -- this is the best that can be achieved.

The dual of this model is *WR-nFD* (Figure 2(d)): misspeculated instructions take up time and resources (indicated by shaded regions), but false dependences are hidden. Performance degradation with respect to *nWR-nFD* is caused by an underutilized window and delayed fetching of correct (control independent) instructions.

The *WR-FD* model (Figure 2(e)) uses no oracle knowledge regarding misspeculated instructions -- they waste both time and resources, and interfere with data dependences. This model represents an upper bound on the performance of superscalar processors exploiting basic control independence.

Finally, the *base* model (Figure 2(f)) squashes all instructions after a branch misprediction.

2.2 Hardware constraints and assumptions

We are interested in the performance impact of instruction window size and machine width (peak fetch, issue, and retire rate) on control independence. In our study, the machine width is 16 instructions per cycle for all simulations, and window size is varied. This is wider than current processors, but may be suitable for a future generation when control independence is seriously considered for implementation [21,22,23].

We implement the following additional hardware constraints and assumptions:

- An ideal fetch unit is assumed. That is, all instructions hit in the cache, and fetching can proceed past any number of branches, taken or not taken, in a single cycle (up to 16 instructions).
- A 5-stage pipeline is modeled: instruction fetch, dispatch, issue, execute, and retire. Fetch and dispatch take 1 cycle each. Issue takes at least 1 cycle, possibly more if the instruction must stall for operands. An instruction is in the execution stage for some fixed latency based on its type, plus any time spent waiting for a result bus. Address generation takes 1 cycle, and all cache accesses are 1 cycle, i.e. a perfect data cache is assumed. Instructions retire in order.
- Any 16 instructions may issue in a cycle because fully symmetric functional units are assumed.
- Output and anti-dependences are eliminated by assuming an unlimited number of physical registers for register renaming and unlimited speculative store buffering for memory renaming.
- Oracle memory disambiguation is used. However, stores fetched down the wrong control path may still interfere with subsequent, control independent loads -- as with register values, false memory dependences may be created in this case.

- A 2^{16} entry *gshare* predictor [24] is implemented for predicting the direction of conditional branches. All direct target addresses are assumed to be predicted correctly since they can be computed at the time of instruction fetch. For indirect calls and jumps, a 2^{16} entry correlated target buffer [25] is used. Returns are predicted using a perfect return address stack [26].

2.3 Benchmarks

Dynamic instruction traces, including both correctly speculated and misspeculated instructions, are generated by the Simplescalar simulator [27]. Five of the integer SPEC95 benchmarks, *gcc*, *go*, *compress*, *jpeg*, and *vortex* were simulated to completion. These benchmarks were chosen to reflect a variety of prediction accuracies, ranging from very predictable (*vortex*) to difficult-to-predict (*go*). Input datasets, dynamic instruction counts, and branch misprediction rates are shown in Table 1. The misprediction rates include both conditional branches and indirect jumps.

TABLE 1. Benchmark information.

benchmark	input dataset	instruction count	misprediction rate
gcc	-O3 genrecog.i	117 million	8.3%
go	9 9	133 million	16.7%
compress	400000 e 2231	104 million	9.1%
ijpeg	vigo.ppm	166 million	6.8%
vortex	modified train input	101 million	1.4%

2.4 Results

Results of simulating the six machine models are in Figure 3. Performance is measured in instructions per cycle (IPC) and is shown as a function of window size.

First of all, a performance upper bound is established with the *oracle* results. These results, assuming perfect branch prediction, are typically over 10 IPC for window sizes of 256 to 512. The machine width upper bound is 16, and most of the benchmarks come close to this mark. Comparing the *oracle* and *base* results indicates a large performance loss due to branch mispredictions with a complete squash (but otherwise ideal) model. For a 512 instruction window, the loss is between 40% and 70% for four of the five benchmarks. The benchmark that has the least performance loss is *vortex* -- but its branch prediction accuracy is quite high. Performance for the *base* model typically saturates at a window size of 128 or 256 instructions. There is no such saturation point for the *oracle* model. These results are consistent with those produced by others and indicate the importance of branch mispredictions on overall performance.

The difference between *oracle* and *nWR-nFD* illustrates performance losses from deferring instructions on a correct control dependent path until after a mispredicted branch is resolved. In *nWR-nFD*, however, machine resources do not sit idle while the mispredicted branch is resolved -- all machine resources are kept as busy as possible fetching and executing the control independent path. The performance loss is typically only 1 to 2 IPC for the medium to large windows.

The *base* model also defers execution of the correct control path following a misprediction, but it gets no benefit from the machine resources before the mispredicted branch is resolved -- any work done after the branch is squashed. When viewed in this way, *nWR-nFD* indicates that the otherwise wasted resources in *base* can lead to large performance benefits. In terms of the way control flow is managed, *nWR-nFD* is most similar to Lam and Wilson's model [10], because misspeculated instructions are ignored.

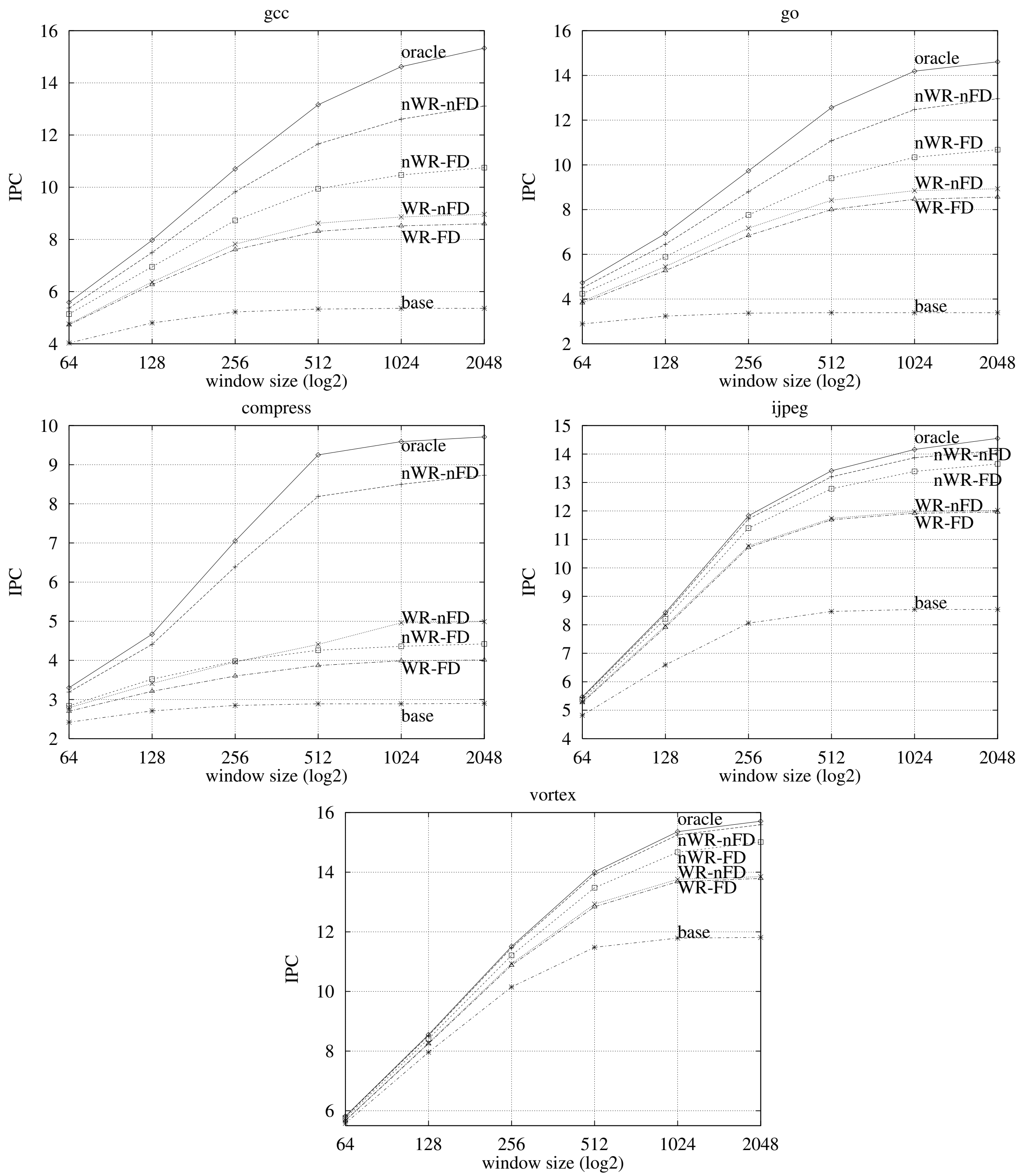


FIGURE 3. Performance of the six control independence models.

With *nWR-FD*, the impact of false data dependences is isolated. For four of the five benchmarks, the performance drop is significant, another 1 to 2 IPC below *nWR-nFD*. *Compress* experiences a much larger drop in performance. False dependences in *compress* limit IPC to under 5 for all window sizes.

With *WR-nFD*, we isolate the effects of wasting resources by executing incorrect control dependent instructions until the branch is resolved. Some resources are still used for the control independent path -- but not until and unless the fetch unit reaches the control independent region. This results in a major drop in performance, bigger than the drop caused by *nWR-FD*. For all benchmarks except *compress*, the effect of wasted time and resources dominates that of false dependences, by about a factor of 2.

With *WR-FD*, we see the combined impact of wasted resources and false dependences caused by incorrect control dependent instructions. Fortunately, the effects are not additive. The *WR* component already dominates, so there is little additional penalty caused by repairing and reissuing false data dependent instructions in the control independent stream (except for *compress*). At this point performance gains are about 100% over the *base* machine.

2.5 Summary and applications of the study

This initial study has established performance bounds for control independence in the context of superscalar processors. The *WR-FD* model reduces the gap between the *oracle* and *base* models by half, and a realistic implementation will fall somewhere between *base* and *WR-FD*.

The other three control independence models also have interesting implications. A major performance limiter is the incorrect control dependent path, primarily because of wasted fetching and window space (*WR-nFD*), but also false data dependences (*nWR-FD*). If these limitations could be mitigated in some way, performance of the *nWR-nFD* model indicates the remaining problem is less significant, i.e. the problem of true data dependences between the deferred, correct control dependent path and control independent instructions.

A possible approach to mitigating the effects of incorrect control dependent instructions is to design instruction windows and fetch units that are less sensitive to wasted resources. The multiscalar architecture is a candidate due to its multiple program counters and “expandable, split-window” [28]. Although strictly speaking our study is only applicable to processors with a single flow of control, we at least get a hint of the control independence potential for *some* multiscalar design points. For example, Vijaykumar’s thesis [29] indicates average task sizes on the order of 15 instructions (comparable to the fetch width of 16 instructions) and effective window sizes of under 200 instructions for integer benchmarks. Given a multiscalar processor with aggressive resolution of inter-task data dependences and selective reissuing capability, the *nWR-FD* model rather than *WR-FD* gives the more appropriate performance bound due to the expandable window.

The large performance drop between *nWR-nFD* and *WR-nFD*, the result of wasted fetch and execution resources, tends to indicate that both hardware and software forms of multi-path execution should be performed carefully. These techniques are applied to both correctly predicted and incorrectly predicted branches. We have shown that wasted resources caused by incorrect predictions alone is a problem; adding some fraction of correct predictions worsens the problem.

3. Implementation Issues

In this section we discuss important implementation issues for exploiting control independence in superscalar processors. This discussion allows us to better understand, qualitatively, where implementation complexities may lie. We do not mean to suggest that the methods we describe are the only ones possible, but we feel the approaches outlined here are adequate for highlighting the major implementation issues that must be considered, and they form a basis for our later performance simulations in Section 4.

3.1 Handling of branch mispredictions

When a branch misprediction is detected in a traditional superscalar processor, the processor performs a series of steps to ensure correct execution. Instructions after the mispredicted branch are squashed and all resources they hold are freed. Typically, freeing resources includes returning physical registers to the freelist and reclaiming entries in the instruction issue buffers, reorder buffer, and load/store queues. In addition, the mapping of physical registers is backed up to the point of the mispredicted branch. The instruction fetch unit is also backed up to the point of the mispredicted branch and the processor begins sequencing on the correct path.

Exploiting control independence requires modifications to the recovery sequence. The overall process is illustrated in Figure 4. Recovery may proceed as follows, although not necessarily in a strict time sequence -- some of these steps can potentially be overlapped.

1. After a branch misprediction is discovered, the first control independent instruction (if it exists) must be found in the instruction window. We call this the **reconvergent point**, because in general control independence exists when control flow diverges and subsequently re-converges.
2. Instructions are selectively squashed, depending on whether they are incorrect control dependent instructions or control independent instructions. Squashed instructions are removed from the window, and any resources they hold are released.
3. Instruction fetching is redirected to the correct control dependent instructions, and these new instructions are inserted into the window which may already hold subsequent control independent instructions. This step combined with steps 1 and 2 above constitute the *restart sequence*.
4. Based on the new, correct control dependent instructions, data dependences must be established with the control independent instructions already in the window. Any modified data dependences cause already-executed control independent instructions to be reissued with new data. This step is called the *redispatch sequence* in Figure 4.

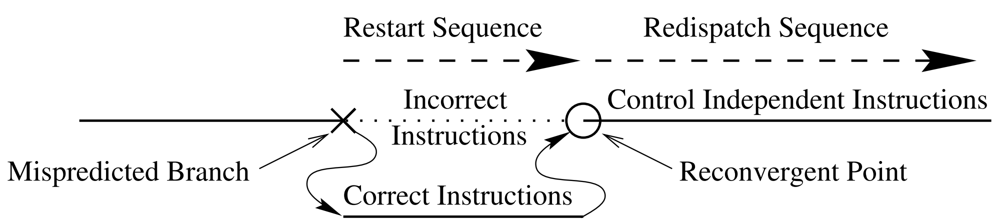


FIGURE 4. Misprediction recovery in a superscalar processor implementing control independence.

3.2 Key microarchitecture mechanisms

To support the above recovery steps, we have identified four underlying microarchitecture mechanisms to be implemented. These are: detecting the reconvergent point, supporting arbitrary insertion and removal of instructions within the window, establishing correct data dependences following a misprediction, and selectively reissuing instructions. In the following subsections we consider implementation alternatives for each of these.

3.2.1 Detecting the reconvergent point

Ideally, one would find reconvergent points by associating with every branch instruction its **immediate post-dominator**: the basic block nearest the branch which lies on every path between the branch and the CFG exit block [30, 31]. In Figure 1, for example, block 4 is the immediate post-dominator of the mispredicted branch. Although the post-dominator does not directly specify the program's control dependences, it is sufficient for identifying all reconvergent points. Finding immediate post-dominators could be very difficult using hardware alone. If binary compatibility does not have to be maintained, software can aid the hardware by encoding this information. For example, the compiler could encode this information by including in each branch instruction an offset to its post-dominator instruction. In most cases this offset is quite small. A second option is to incorporate post-dominator registers into the architecture. Software can load these registers with the addresses of post-dominator instructions for soon-to-be-executed branches and then specify a post-dominator register in each branch instruction.

Hardware-only solutions for detecting reconvergent points probably require heuristics that are less accurate than using complete post-dominator information. One less aggressive hardware alternative is to identify points in a program where multiple paths converge. There are some common constructs in a program that exhibit this behavior, such as targets of subroutine return instructions, or targets of backward branches that form a loop. These points can be determined with hardware tables that monitor the dynamic stream and record program counter values of such reconvergent points. When a branch misprediction is detected, hardware can consult the table for the first such reconvergent point and assume it to be the correct reconvergent point for the mispredicted branch. This approach preserves only a subset of the control independent code after a branch misprediction, but requires less information to be learned by hardware. A more complicated approach could attempt to learn pairs of branches and their corresponding reconvergent points.

3.2.2 Instruction removal/insertion

Following the detection of a reconvergent point, the instruction window must be repaired by selectively removing incorrect control dependent instructions preceding the reconvergent point, and fetching instructions from the correct control dependent path. We refer to this process as the *restart sequence*, shown in Figure 4.

The restart sequence requires selectively removing and inserting instructions while maintaining a correct ordering. The reorder buffer (ROB) of a traditional superscalar processor can be augmented to support this. One option is to have the ROB support arbitrary physical shifting of instructions to collapse and expand the window for restart sequences. This first option causes the physical ROB slots to move, and any instruction tags in the pipelines pointing to them will become out-of-date. This complication can be partially solved by adding a level of indirection.

A second option is to implement the ROB as a linked list. Then, any outstanding instruction tags do not change as the ROB is repaired, but dispatch and retirement will be complicated by multiple linked list operations being done in parallel. The complexity of manipulating the linked list can be reduced by implementing it at a granularity larger than a single instruction. That is, ROB space can be partitioned into multi-instruction blocks. For example, a 256 instruction ROB can be implemented as 16 blocks of 16 instructions each. Then, a block at a time can be inserted or removed from the ROB in a more-or-less conventional way. This reduces complexity but also reduces full utilization of the window as ROB blocks will often not be fully utilized. For example, when the processor needs to insert eight instructions into the middle of the ROB, it will allocate a full block of 16 but use only half the entries.

Load/store buffers with insertion and removal can be implemented in a similar manner as the ROB, but they have the added complication that they may require sequence-sensitive address comparisons to resolve dependences.

Freeing resources for selectively squashed instructions is likely to be less efficient than complete squashing. Reclaiming resources includes returning physical registers to the freelist and freeing load/store buffer entries. Reclaiming resources selectively may require sequencing through the squashed instructions and iteratively reclaiming their resources. However, if selective squashing is done in parallel with fetching new instructions, at least some of the latency may be effectively hidden. In the process, new instructions may acquire the resources being freed by the old instructions.

Finally, another complication occurs if the window fills with new instructions before the reconvergent point is reached. That is, there are more new correct control dependent instructions than there were old incorrect ones. In this case, it is necessary to begin squashing control independent instructions (youngest first), allowing the restart sequence to proceed.

3.2.3 Forming correct data dependences

As pointed out earlier, although instructions may be *control* independent with a preceding block of instructions, they may not be *data* independent. Consequently, correct ordering of data dependences, both through registers and memory, must be recovered when a misprediction occurs. Register dependences may be maintained through the existing physical register mapping mechanisms. To update dependence information, instructions in a control independent region must be redispatched [1]. During redispatch of instructions their register source operands are remapped while their register destination operands maintain their original assignments. If an instruction's register source operand is mapped to a new physical register, the instruction must be reissued.

Memory dependences can be maintained through an augmented memory-ordering buffer. The memory-ordering buffer must detect when a preceding store is removed or inserted by a restart sequence and direct subsequent loads to reissue. This functionality can be added to an address resolution buffer [32] or large load/store queue, the main modifications being that the structures have to support selective insertion and removal similar to the reorder buffer.

3.2.4 Selective reissuing of instructions

If an instruction's register source operand is mapped to a new physical register, the instruction must be reissued. As these instructions are reissued, they will produce new values, and instructions in data dependence chains following these instructions will also need to reissue.

Ultimately, instructions may issue and execute multiple times before they eventually retire. Reissuing, therefore, becomes a common case and the microarchitecture must be modified to reflect this. To reduce the complexity and latency of reissuing instructions, they remain in the instruction issue buffers until they retire [1, 12]. Instruction issue buffers can be built to reissue their instructions autonomously when they observe a new value being produced for a source operand. This functionality can be built into the normal issue logic. Thus, the redispatch logic need only identify instructions directly affected by incorrect data dependences, and the following data dependent chain of instructions will automatically reissue.

4. Performance of control independence in a superscalar processor

The idealized studies of Section 2 provide insight into the factors that govern performance of control independence. Having done so, we now proceed with a more refined analysis, focusing on an implementation of the model *WR-FD*. The analysis is based on a detailed, fully-execution driven simulator, and reflects the performance impact of implementing the basic mechanisms outlined in Section 3.

4.1 Simulator detail

Many of the basic hardware constraints are the same as in Section 2. The machine width is 16 instructions and the underlying pipeline is similar. Instruction fetching remains ideal, but a more realistic data cache is modeled. The data cache is 64KB, 4-way set associative. The cache access latency is two cycles for a hit instead of one, and the miss latency to the perfect L2 data cache is 14 cycles. Also, realistic, but aggressive, address disambiguation is performed. Loads may proceed ahead of unresolved stores, and any memory hazards are detected as store addresses become available [32] -- recovery is via the selective reissuing mechanism. Lastly, the branch predictor, while identical to that in the ideal study, may have lower accuracy due to delayed updates (tables are updated at retirement).

The key mechanisms for supporting control independence, outlined in Section 3, are modeled as follows.

Detecting the reconvergent point is done via software analysis of post-dominator information. Several hardware-only mechanisms are discussed and evaluated in Appendix A.5.

Instruction removal/insertion gives equivalent performance whether the shift register or linked list approaches are used. In the simulator, we implemented a linked list approach that uses single instruction granularity. Larger granularities are evaluated in Appendix A.4.

Forming correct data dependences is delayed some number of cycles after the misprediction is detected, unlike the ideal study, because the redispatch sequence cannot proceed until after the restart sequence completes. Further, redispatch proceeds at the maximum dispatch rate. However, we also modeled single-cycle redispatch of all control independent instructions (after the restart phase completes), in order to study its performance impact.

Selective reissuing is modeled in detail, whereas the ideal study models only the *delay* caused by repaired dependences, i.e. only the final instruction issue. The source of reissuing includes both register rename repairs and loads squashed by stores, followed by a cascade of reissued instructions along the dependence chains.

4.2 Performance results

Figure 5 shows the instructions per cycle (IPC) for three different machines: a superscalar processor that squashes all instructions after branch mispredictions (BASE), a processor with control independence capability (CI), and one with the added capability to instantaneously repair data dependences and redispatch all control independent instructions after the restart sequence completes (CI-I). Measurements are made for three window sizes, 128, 256, and 512 instructions.

For less predictable workloads, control independence offers a significant performance advantage over complete squashing, although less than the ideal study indicated. The relative performance improvement of CI over BASE for each of the window sizes is summarized in Figure 6. *Go*, *compress*, and *jpeg* show improvements on the order of 20% to 30%. While *jpeg* is fairly predictable, it is also rich in parallelism and any misprediction cycles result in a large penalty. *Go* on the other hand is a very control-intensive workload with frequent mispredictions, and it demonstrates the most performance benefit.

Gcc also shows a substantial performance gain, about 10%. Statistics presented in the next section show that approximately 60% of *gcc*'s mispredictions have a corresponding reconvergent point in the window, while for *go*, *jpeg*, and *compress* the same statistic is over 70%. The fact that less control independence is exposed may partially account for the lower performance gain.

From Figure 5 we see that CI-I, as expected, gives better performance than CI. However, the gain is surprisingly small -- between 1% and 4%. This is a positive result because it means the time spent during redispatch sequences has less impact than anticipated. Redispatch ties up the sequencer, preventing it from fetching new instructions into the window, and also delays the repair of some register dependences. As for the latter, statistics in Section 4.3 (Table 2) show that not many instructions need to repair register dependences, and we also suspect that those in need of repair are close to the reconvergent point and thus repair quickly.

Compress actually shows a small drop in performance for the CI processors when the window is increased from 256 to 512 (although performance is still better than BASE). As will be seen in the next section, *compress* exhibits an unusually high number of memory ordering violations. This situation is only worsened with larger window size -- and particularly where control independent instructions are saved -- because more loads have the opportunity to proceed before dependent stores. The drop in performance is due to a 1-cycle penalty for loads squashed by stores. The effect is amplified in *compress* because there are extremely long dependence chains in the benchmark, as can be seen by the large number of reissued instructions presented in the next section.

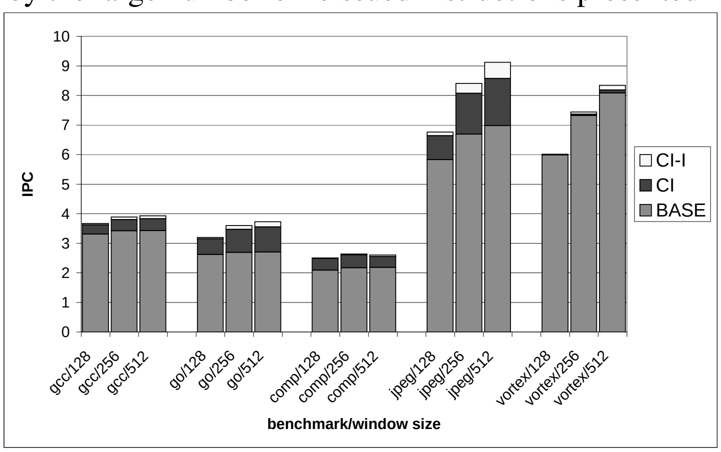


FIGURE 5. Performance with and without control independence, for three window sizes.

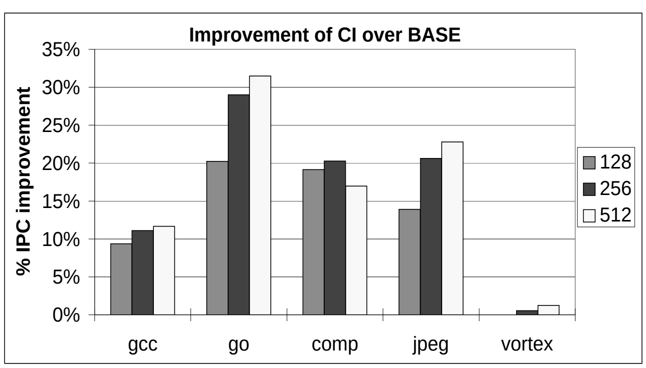


FIGURE 6. Percent improvement in IPC due to control independence.

We would expect with larger window sizes, more control independence is exposed. However, according to Figure 6, only two of the benchmarks show a substantial variation with increasing window size -- *go* and *jpeg* -- and even then most of the variation occurs between 128 and 256. Yet our ideal study shows more variation with window size. In addition to the obvious configuration differences enumerated in Section 4.1, there are a host of subtle issues that contribute to differences between the ideal and implementation studies; some of these issues are treated in Appendix A.

4.3 Other control independence measures

This section explores the behavior of control independence in a superscalar processor to better understand the performance results given in the previous section. The results in this section are for the intermediate window size of 256 instructions.

The first column of Table 2 shows how often a control independent reconvergent point is in the window at the time a control misprediction is detected. In all the benchmarks except *vortex* a reconvergent point is present for over 60% of mispredictions.

The second and third columns of Table 2 show the average number of instructions removed and inserted *for those restart sequences that reconverge in the window*. The average number of instructions removed for a restart, the dynamic distance between the misprediction point and reconvergent point on the incorrect path, is less than 14 for all the benchmarks. The average number of instructions inserted for a restart, the dynamic distance between the misprediction point and reconvergent point on the correct path, is less than 20 for all the benchmarks. For both removal and insertion the distance is 32 or less for over 80% of the restarts (not shown in the table).

The average number of inserted instructions is higher than that of removed instructions because we only consider mispredictions that have a corresponding reconvergent point in the window. Consequently, mispredictions with many incorrect control dependent instructions do not contribute to the average number of removed instructions if the reconvergent point is not reached.

TABLE 2. Statistics for restart/redispatch sequences.

Benchmark	% of mispredictions that reconverge	Avg. # of removed control dep. instr.	Avg. # of inserted control dep. instr.	Avg. # of control indep. instr.	Avg. # of control indep. instr. squashed due to new register name(s)
gcc	61.8	13.2	16.5	51.8	2.75
go	71.2	13.5	18.1	62.4	2.18
compress	90.8	6.8	6.6	122.1	1.74
jpeg	81.6	9.0	10.7	79.8	2.17
vortex	46.8	9.2	12.8	81.5	2.10

The fourth column in Table 2 shows that the average number of control independent instructions after the reconvergent point is greater than 50 for all the benchmarks. Further, the last column in Table 2 shows that on average, only 2 to 3 of the control independent instructions will acquire new physical register names during redispatch, requiring them to reissue. Additional control independent instructions will reissue due to memory dependences or data dependences with other control independent instructions that reissue. Also, some of these control independent instructions may be parts of incorrect control paths and will later be squashed.

Table 3 shows the amount of useful work that can be saved with control independent instructions. In this table we look only at correct instructions that ultimately retire. Ignoring *vortex*, 13% (*jpeg*) to 70% (*compress*) of all retired instructions are fetched before a preceding mispredicted branch is resolved. Without using control independence these instructions would be squashed and fetched again. More importantly, 11% (*jpeg*) to 39% (*compress*) of all retired instructions issue and have their final value before a preceding mispredicted branch is resolved. Without using control independence this work would be lost. Of control independent instructions that do not have their final value at the time the misprediction is resolved, most have issued and are forced to reissue due to data dependences (the column labeled "work discarded").

TABLE 3. Work saved by exploiting control independence, as a fraction of retired instructions.

benchmark	fetch saved	work saved	work discarded	had only fetched
gcc	27%	20%	5%	2%
go	39%	30%	6%	3%
comp	70%	39%	27%	4%
jpeg	13%	11%	2%	0%
vortex	5%	4%	1%	0%

Table 4 shows how often and why instructions reissue. Even without control independence, memory ordering violations due to incorrect disambiguation cause instructions to reissue. Without control independence, instructions issue on average 1.04 (*jpeg*) to 1.24 (*compress*) times. 0.5% to 6% of instructions are loads that reissue due to memory ordering violations, which in turn cause chains of dependent instructions to reissue.

With control independence, the average number of times each instruction issues increases to 1.10 (*jpeg*) to 2.44 (*compress*). Memory ordering violations result from (1) incorrect disambiguation and (2) incorrect memory dependences caused by branch mispredictions. The two components tend to be equal. Other instructions reissue because of incorrect register dependences caused by branch mispredictions. When instructions reissue due to memory or register data dependences, they cause chains of dependent instructions to reissue.

TABLE 4. Instruction issues per retired instruction.

Benchmark	no control independence		control independence		
	total	due to memory violations	total	due to memory violations	due to register violations
gcc	1.07	0.015	1.19	0.027	0.033
go	1.10	0.015	1.32	0.032	0.025
comp	1.24	0.061	2.44	0.063	0.051
jpeg	1.04	0.005	1.10	0.010	0.007
vortex	1.12	0.019	1.14	0.021	0.002

5. Conclusions and Future Work

This research refines our understanding of control independence, perhaps the least understood solution to the conditional branch problem. The study establishes new performance bounds that account for practical implementation constraints and incorporate all data dependences. To gain insight, the study identifies three important factors and isolates their impact on performance: true data dependences between correct control dependent instructions and control independent instructions, false data dependences created by incorrect control dependent instructions, and wasted resources consumed by incorrect control dependent instructions. A conclusion is that both types of data dependences limit the potential of control independence in perhaps unavoidable ways, but the biggest performance limiter is wasted resources consumed by incorrect control dependent instructions. This limitation may be reduced in designs capable of "absorbing" wasted instruction fetch and execution bandwidth.

This paper also discusses important implementation issues and provides some design alternatives. Simplified alternatives are also discussed to address some of the more complex aspects, such as the segmented ROB for arbitrary insertion/removal of instructions, and hardware heuristics for identifying the reconvergent point. Detailed simulations of a superscalar processor implementing the key features show typical performance improvements of 10 to 30 percent over a baseline superscalar processor. The speedup is derived from 20 percent of retired instructions whose computation is saved as a result of control independence.

The purpose of this work is not so much to advocate control independence in superscalar processors as to promote other control independence architectures. This research is a necessary step towards improving control independence in trace processors, whose hierarchical structure provides a simpler implementation in many respects, including arbitrary instruction insertion/removal. Further, the abstract *nWR-FD* machine model suggests combining the expandable window model of multiscalar processors with the aggressive data dependence resolution and recovery model of trace processors.

Appendix

A. Detailed issues in control independent designs

This section describes many of the issues we encountered when trying to understand and exploit control independence. These issues only became apparent during the translation from ideal study to detailed implementation, and they partially explain discrepancies between the idealized experiments and the measurements taken from the detailed execution-driven simulator.

While a few of the problems are unique to control independence processors with a single program counter (e.g. handling multiple concurrent branch mispredictions), several apply to any control independence architecture, including those with multiple flows of control. In particular, the problem of false mispredictions (Section A.2) and the interaction between control independence and global branch history (Section A.3) have more far-reaching implications.

Unless otherwise stated, all results are for a 256 instruction window.

A.1 Handling multiple branch mispredictions

In Section 3, implementation issues were discussed in the context of recovery from a single mispredicted branch. In reality, the recovery process can potentially consume many cycles, and while a recovery is in progress, the processor may determine that a branch logically preceding the current restart sequence has also been mispredicted. This can easily occur when branches are allowed to execute out-of-order. Even if branches are required to execute in-order this can still occur in limited cases -- while fetching instructions for a restart sequence, a newly fetched branch may execute and determine that its prediction was incorrect. Our preliminary performance studies indicated that handling restart sequences serially without preemption can lead to significant performance degradation, because the processor may be delayed from bringing good instructions into the window while it is fetching and/or redispatching instructions from an incorrect path.

We have determined this effect to be quite significant and some form of preemption is necessary. We begin with a simple preemption strategy that results in some performance loss but has minimal impact on the instruction fetch unit. This method was used in the primary performance evaluation of Section 4. To determine the performance degradation of simple preemption, optimal preemption is also presented (the ideal study of Section 2 models optimal preemption).

A.1.1 Simple preemption

Figure 7 shows three possible cases where a branch misprediction logically preceding an active restart/redispatch sequence is detected. The logical sequence of instructions is represented by the solid line going from left to right. The terms "later" and "earlier" refer to the times that mispredictions are detected. So, in the figure the later mispredicted branch in fact appears first in the logical program sequence. The three cases listed below differ in the location of the reconvergent point of the later mispredicted branch.

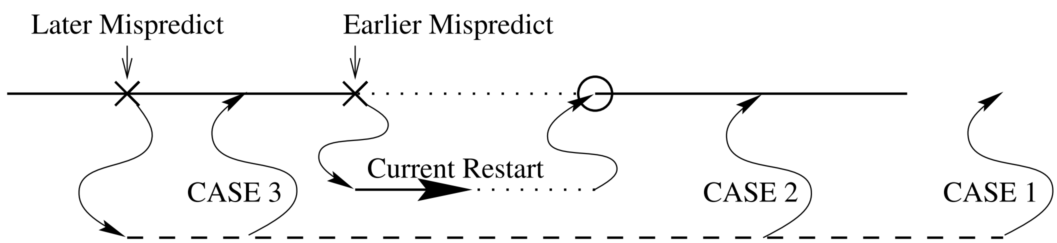


FIGURE 7. Three cases for preemption of a restart/redispatch sequence.

CASE 1: the later mispredicted branch may not have a corresponding reconvergent point in the window. In this case, all the instructions in the window following the later mispredicted branch can be squashed.

CASE 2: the later mispredicted branch has a reconvergent point that occurs after the current reconvergent point (caused by the earlier misprediction). In this case all the instructions from the current restart sequence will be squashed and instructions after the new reconvergent point will have to go through redispatch again. In these first two scenarios, it is reasonable to preempt the active restart/redispatch sequence, i.e. the behavior is identical to recovery from a single misprediction.

CASE 3: the later mispredicted branch has its reconvergent point before the current restart sequence. In this case the instructions in the current restart sequence and those following the current reconvergent point may still be part of the correct path. In order to avoid delays in servicing the new misprediction and to avoid adding extra state to the sequencer, the most straight-forward approach is to preempt the active restart sequence, and squash instructions following the current reconvergent point. The more complex alternative is to have the sequencer remember that there was a restart in progress, and after servicing the new restart sequence, the sequencer must return to the preempted restart to continue filling the gap in the instruction window.

The simple preemption strategy for CASE 3 results in a performance loss (compared to the complex alternative). However, the sequencer does not have to keep track of multiple outstanding restart sequences, only the most recent one.

Note that preempting a re-dispatch sequence is simpler because backing up the sequencer ensures that the instructions will eventually be re-dispatched by the latest recovery process.

A.1.2 Optimal preemption

As described above, optimal preemption requires maintaining state for all outstanding restart sequences. This may not be overly complex: a minimum of sequencer state (PC, where in the window instructions are to be inserted, and information about the reconvergent point) might be pushed onto a hardware stack to preempt a restart sequence, and resuming restart sequences in the proper order is achieved by popping state from the stack. However, preemption state may have to be selectively deleted from the middle of the stack if the corresponding restart sequences themselves belong to a mispredicted path and are squashed.

A.1.3 Preemption results

Figure 8 shows the performance of both simple and optimal preemption models. Simple preemption performs as well as optimal preemption, at least for a 256 instruction window, because restart sequences that reconverge in the window have a duration of only 1 or 2 cycles on average. *Gcc*, *go*, *compress*, and *jpeg* have average durations of 1.6, 1.6, 1.1, and 1.2 cycles respectively. For all of the benchmarks, about 90% of all restarts require 3 or fewer cycles. As a result, preemptions (including case-3 preemptions) are rare.

Preemptions will become more frequent in larger windows, due to more branches and a higher chance for concurrent misprediction detection. A lower fetch bandwidth also increases the frequency of preemptions, because restarts take longer to service.

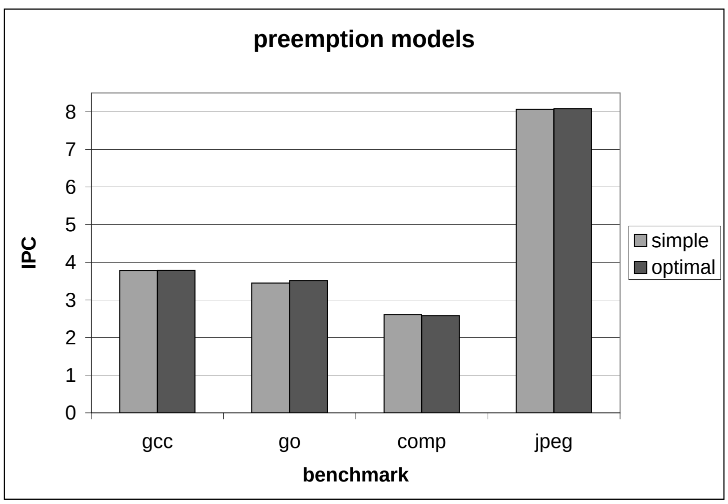


FIGURE 8. Evaluation of simple and optimal preemption for handling multiple branch mispredictions.

In the experiments that follow, optimal preemption is used because other enhancements may be artificially limited by simple preemption. This probably is not the case, but rather than simulate all combinations, we chose the least restrictive preemption model.

A.2 False mispredictions

A *false misprediction* occurs when a branch that is predicted correctly executes with speculative, incorrect operands, and as a result, the branch prediction is assumed to be incorrect. A false misprediction causes what are actually correct instructions to be squashed.

The operands of a branch may be incorrect for various reasons. In a processor with control independence, a mispredicted branch can introduce incorrect data dependences which ultimately affect subsequent control independent branches. Other sources include incorrect values produced by data speculation, e.g. value prediction and memory dependence speculation. In *compress* for example, the high frequency of loads that issue before dependent stores may cause false mispredictions.

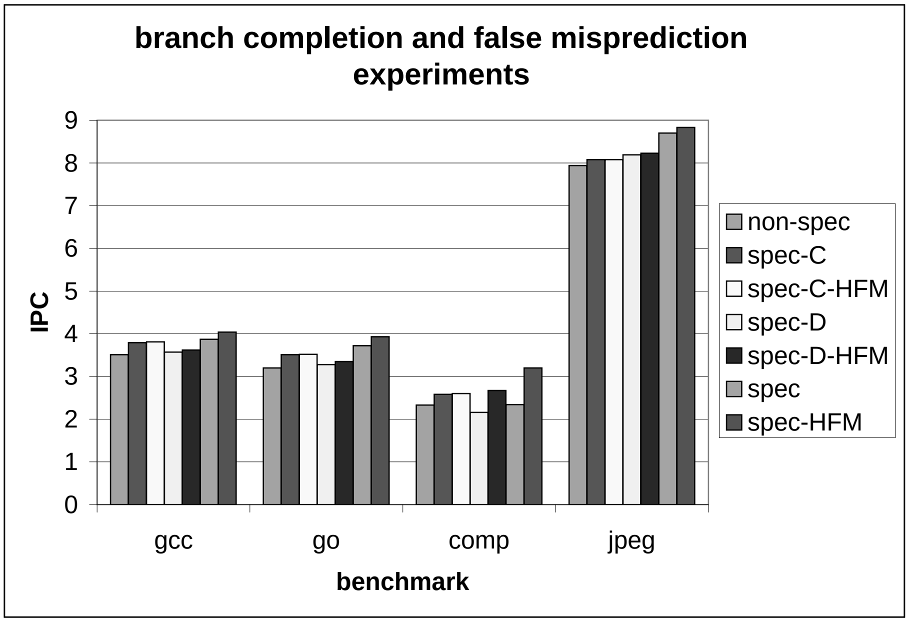
A.2.1 Performance impact of false mispredictions

False misprediction is one source of discrepancy between the idealized models and the detailed execution-driven simulator. The impact of false mispredictions is measured in the execution-driven simulator by using oracle information to detect and prevent false mispredictions from occurring. The following configurations are simulated (all in the context of a processor with control independence mechanisms).

- *non-spec*: Branches are not allowed to complete until their operands are known to be non-speculative. This means (1) branches must execute in-order, so that operands are non-speculative in terms of *control flow*, and (2) all instructions that may affect a branch's operands must themselves be non-speculative before the branch can execute, so that operands are non-speculative in terms of *data flow*. In this branch completion model, there are no false mispredictions.
- *spec-D*: Branches must execute in-order, but branches need not wait for any other instructions to be non-speculative. Hence, *spec-D* refers to the fact that operands may still be speculative due to **d**ata speculation, in our case loads issuing early.
- *spec-D-HFM*: This is the same as *spec-D*, except oracle information is used to detect branches that will cause false mispredictions if allowed to complete. In these cases, branch completion is delayed, thereby preventing false mispredictions: *HFM = hide false mispredictions*.
- *spec-C*: This is the dual of *spec-D*. Branches may complete out-of-order, but other instructions that may affect a branch's operands must be non-speculative before the branch can complete. Hence, *spec-C* refers to the fact that operands may still be speculative due to **c**ontrol speculation.
- *spec-C-HFM*: This is the same as *spec-C*, but false mispredictions are prevented.
- *spec*: Branches may complete whenever operands are available. This means branches complete without regard to speculative operands.
- *spec-HFM*: This is the same as *spec*, but false mispredictions are prevented.

The results of the seven models are shown in Figure 9. The first graph shows IPC for each model, and the second graph shows the percent IPC difference between any two specified models.

Referring to the second graph, it is clear from the first bar (*spec-C* over *non-spec*) that completing branches out-of-order is important, about a 10% impact. This performance improvement comes from detecting true mispredictions quickly, although not as early as possible because branch operands cannot be data-speculative. Further, from the fourth bar (*spec-C-HFM* over *spec-C*) it is clear that this early evaluation does not result in many false mispredictions; preventing false mispredictions in *spec-C* results in less than 1% improvement.



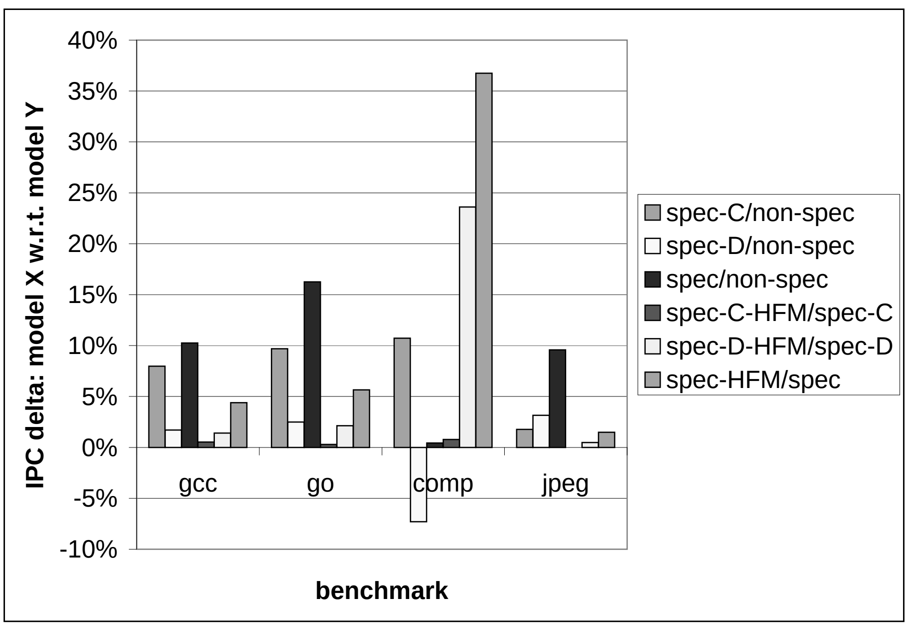


FIGURE 9. Performance impact of branch completion models and false mispredictions.

From the second and third bars (*spec-D* and *spec* over *non-spec*, respectively), we conclude that (1) except for *jpeg*, allowing data-speculative operands (*spec-D*) is less important than completing branches out-of-order (*spec-C*), but (2) allowing data-speculative operands becomes more important when branches are allowed to complete out-of-order (*spec*). That is, the combined effect of *spec-C* and *spec-D* is greater than the sum of the two. The only exception is *compress*, for which allowing data-speculative operands has negative consequences. This is understandable considering the large number of load-store ordering violations in *compress*.

From the fifth bar (*spec-D-HFM* over *spec-D*), it is apparent that allowing data-speculative operands results in more false mispredictions than allowing control-speculative operands. Still, if false mispredictions can be prevented in the *spec-D* model, the result is only about a 3% improvement for three of the benchmarks. *Compress,* as expected, can benefit significantly by eliminating false mispredictions -- a 24% improvement over *spec-D*.

Finally, from the sixth bar (*spec-HFM* over *spec*) we can assess the total impact of false mispredictions when branches are allowed to execute as soon as operands are available. False mispredictions affect performance by 5% for *gcc* and *go*, 2% for *jpeg*, and 37% for *compress*.

From these results, we conclude that with only a small degree of data speculation (i.e. memory dependence speculation, but not value prediction), it is probably best to implement the *spec* model. We have shown that it is more important to resolve true mispredictions as early as possible than try to avoid false mispredictions by being conservative. In the following section, we present intelligent techniques for identifying false mispredictions, so that branches may be selectively identified for early or late completion. These techniques may be used as a hedge against false mispredictions if they are a major problem in other workloads, or other processor configurations (e.g. larger, more speculative windows).

Spec-C is the branch completion model used in our primary results section (Section 4) and unless otherwise stated is used for the remainder of the experiments. *Spec-C* was chosen for its robustness across all of our benchmarks. *Compress*, however, is somewhat of a microbenchmark (as seen in the next section) and its anomalies should not have too much influence in designing control independent processors.

A.2.2 Identifying and preventing false mispredictions

In this section ways of detecting and avoiding false mispredictions are discussed. One obvious solution is to use a branch prediction confidence mechanism [33], which assesses the likelihood that a given branch prediction will turn out to be incorrect. A high-confidence assessment of a branch prediction delays the completion of a branch if its operands are speculative. Delaying a correctly-predicted branch does not degrade performance and may prevent false mispredictions from occurring. On the other hand, delaying a true misprediction from being resolved can seriously degrade performance.

Our early experiments using branch confidence to prevent false mispredictions have not produced good results. All too often more true mispredictions are delayed than false mispredictions prevented.

These early experiments motivate a second technique to identify false mispredictions. Branch prediction confidence is indirect in that the history of correct and incorrect branch predictions is monitored. It may prove more useful to directly monitor the history of true and false mispredictions instead.

We begin by collecting true/false misprediction statistics per static branch, analogous to the static confidence measurements in [33]. For each static branch, we measure the total number of true mispredictions it contributes as well as the total number of false mispredictions it contributes. This data is used to compute the *false misprediction rate* per branch, that is, the ratio of false mispredictions to total mispredictions for a given branch. The branches are then sorted from higher to lower false misprediction rate. Finally, using the sorted list of mispredicted branches, the cumulative fractions of true and false mispredictions are computed. The resulting graph is shown in Figure 10, with cumulative fractions of true and false mispredictions plotted along the x-axis and y-axis respectively.

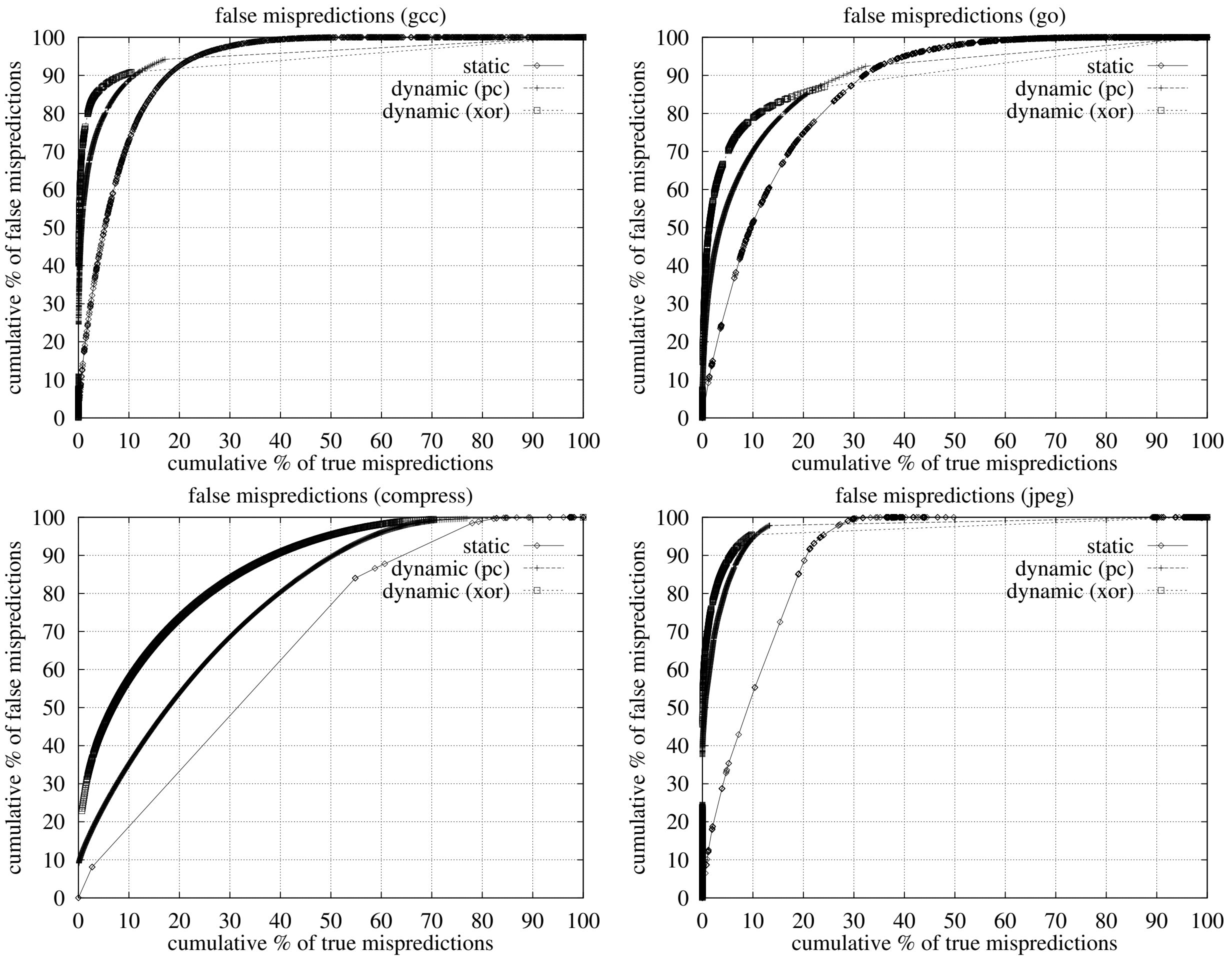


FIGURE 10. Using true/false misprediction history to detect false mispredictions.

From the curve labeled *static*, we can see that 90% of all false mispredictions can be detected and prevented at the expense of delaying only 20% of all true mispredictions, for *gcc* and *jpeg*. For *go*, 75% of false mispredictions can be detected for the same point. In compress, a single branch accounts for over 50% of the true mispredictions and 75% of the false mispredictions -- clearly a static identification scheme is ineffective in such cases.

The *static* implementation implies profiling per-branch false misprediction rates, choosing a threshold rate, and marking branches above the threshold. At run-time, these branches are delayed until their operands are non-speculative.

The *static* scheme does not exploit dynamic behavior in that a branch is either always delayed or never delayed. A dynamic scheme may be more effective in separating true from false mispredictions. A hardware table is used to collect true/false misprediction history. Rather than propose a specific automaton, we begin by maintaining a 16-bit shift register of history, called the TFR ("**T**rue/**F**alse misprediction **R**egister"). This is analogous to the CIR in [33], but the TFR is updated only for mispredicted branches. A '1' is shifted in for a false misprediction and a '0' for a true misprediction. In these experiments a 2^{16}-entry table of TFRs is maintained, indexed either by the PC or the PC XORed with global branch history (like *gshare*).

The same process described above is used to generate curves for the dynamic schemes, but instead of gathering misprediction statistics per static branch, they are gathered per TFR pattern. The TFR patterns are sorted by false misprediction rate and cumulative fractions of true and false mispredictions are plotted.

From Figure 10, it is apparent that dynamic schemes identify more false mispredictions while delaying less true mispredictions. The curve labeled *dynamic(pc)* uses only the PC to index into the TFR table, and the curve labeled *dynamic(xor)* uses a *gshare* index. If only 10% of true mispredictions are to be delayed, 90%, 80%, 60%, and 95% of all false mispredictions can be detected for *gcc*, *go*, *compress*, and *jpeg*, respectively. This is for the *dynamic(xor)* scheme. If we can tolerate delaying 20% of true mispredictions, then 75% of false mispredictions can be detected in *compress*.

The results for the dynamic techniques demonstrate the *potential* for identifying false mispredictions. Developing *reduction functions* [33] that capture the desired TFR patterns is left for future work. It is not clear that resetting counters, which perform well for confidence estimation, are well-suited for identifying false mispredictions.

A.3 Branch prediction issues

For the most part, branch predictors have been designed for processors that sequentially predict and fetch instructions, with the implicit assumption that all instructions following a branch misprediction are squashed and re-predicted with the most up-to-date branch history. This poses problems for any form of out-of-order instruction fetching, e.g. control independence in superscalar processors, or hierarchical sequencing in multiscalar and multithreaded processors. The problem is a branch may have to be predicted based on an incomplete or *incorrect* history of prior branches.

Two-level predictors that use global branch history, such as the *gshare* predictor used in this work, while highly accurate, are potentially problematic in control independence machines. In Figure 11, the two branches b1 and b2 are correlated and b1 is mispredicted. Because of the correlation, the *gshare* predictor is likely to also mispredict b2. In a conventional processor with complete squashing, the second misprediction b2 is irrelevant: the sequencer backs up to b1 and re-predicts branch instructions, this time with the up-to-date history *including b1's correction*. Thus, b2 is likely to be predicted correctly.

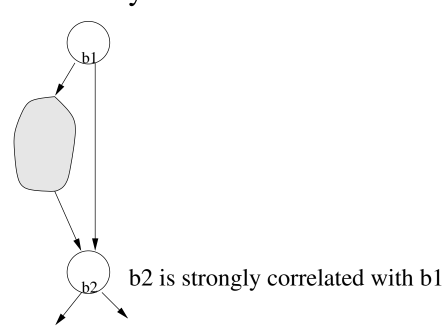


FIGURE 11. Example of using incorrect global branch history to predict branches.

This has two implications.

- Control independence does not obviate the need for re-predicting branches. As with complete squashing, the branch predictor must be backed up to the misprediction, the global history corrected, and instructions re-predicted during the re-dispatch sequence. Thus, re-dispatch sequences are not only needed to repair data dependences, but also to iteratively improve branch predictions within the instruction window as global history is corrected. Without these early corrections, the advantages of correlation are negated and performance may actually worsen with respect to a simpler, local-history branch predictor.

- Simulation models that assume a correct global history for every branch prediction are misleading in the context of control independence. The conventional branch prediction accuracy metric does not hold. For example, the initial prediction for b2 would in fact appear as a misprediction and reduces the apparent benefit of control independence. The idealized study in this paper, Lam and Wilson's limit study, and Uht and Sindagi's limit study are overly optimistic in this respect: the studies assume correct global history for predicting branch b2 the first time, so b2 is predicted correctly, whereas the accurate timing model used in Section 4 of this paper mispredicts b2.

A.3.1 Global branch history

The second bullet above is potentially a source of discrepancy between the idealized study and the detailed timing model. To evaluate the impact of assuming correct global history, we implemented *oracle global history* in the detailed execution-driven simulator: a given branch is predicted using what is ultimately the correct global branch history leading up to that branch.

The graph in Figure 12 shows that the effect is not large, a maximum change in IPC of plus or minus 5% with respect to using timing-accurate, possibly incorrect global history. Strangely, *jpeg* exhibits worse performance with oracle branch history. We do not have a definite reason for why this is the case. *Jpeg* may legitimately perform better with the patterns created by delayed corrections to the global history register.

Or this may be an artifact of the simulation method, which cannot *guarantee* matching a given branch with its correct global branch history. The simulator runs a second, fully-accurate instruction window in parallel with the actual processor window, and maintains a mapping of good instructions in the processor to counterparts in the fully-accurate window; these counterparts provide the oracle branch history. Because loop iterations and function instances may be inserted at any time into the middle of the instruction window, initial mappings may be incorrect due to instance mismatches.

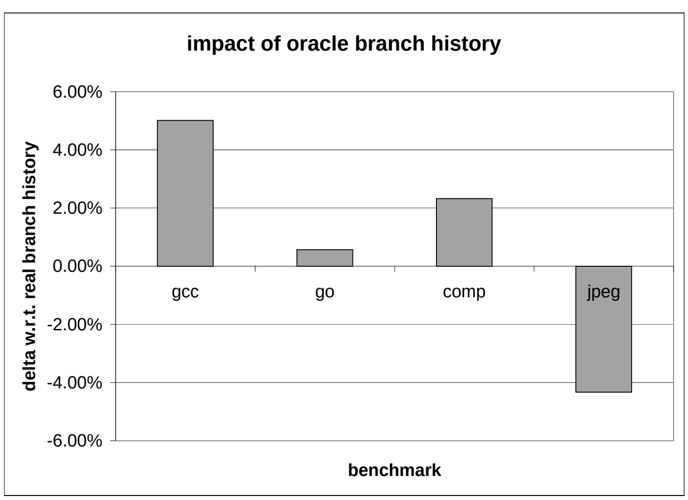


FIGURE 12. Impact of assuming oracle global branch history.

A.3.2 Re-predict sequences

It is quite possible for a re-prediction to overturn a correct prediction, or worse, to overturn a branch that has already executed. We have determined that the latter case is important and can often be avoided. A good heuristic that is implemented in the execution-driven simulator is to *force* the branch predictor if a branch is in the "completed" state. On the other hand, if the branch is not in the "completed" state, the branch predictor dictates the re-prediction.

In Figure 13, we first evaluate the importance of re-predicting branches. The bar labeled *CI-NR* shows the performance of control independence mechanisms with no re-predict sequences. That is, initial predictions are maintained until and unless branches complete and overturn the predictions. Thus, there are no *early corrections* of predictions as global history changes. For reference, the performance of a processor without control independence is also shown, labeled *base*.

Second, to assess the re-prediction heuristics implemented in our design, labeled *CI*, they are compared with *oracle re-predict* sequences, labeled *CI-OR*. The model *CI-OR* is oracle in the sense that correct predictions are never overturned during re-predict sequences. *CI* differs from *CI-OR* in two ways: (1) branches not in the "completed" state cannot force the predictor where the oracle model might and (2) branches in the "completed" state may have an incorrect outcome and wrongly force the predictor.

The important conclusion is that re-predict sequences are necessary. For *gcc* and *compress*, not having re-predict sequences degrades performance to near or below the *base* machine. For *go* and *jpeg*, not having re-predict sequences reduces the benefit of control independence by half: from 30% to 15% for *go*, and 20% to 12% for *jpeg*.

Comparing *CI* to *CI-OR*, we see that our re-prediction mechanism performs within 5% of oracle re-prediction for three of the benchmarks. For *compress*, however, *CI-OR* performs 25% better than *CI*. All too often, either the predictor overturns correct predictions or completed branches incorrectly override the predictor. Because these results are for the *spec-C* completion model, we suspect the branch predictor to be at fault (re-predictions overturning correct predictions).

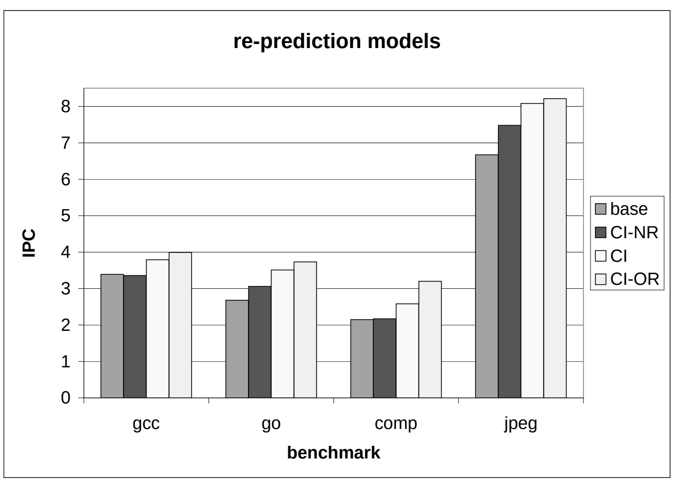


FIGURE 13. Evaluation of re-predictions.

A.4 Segmented reorder buffers

The non-hierarchical, inflexible, contiguous window organization of superscalar processors is a primary source of complexity for implementing control independence. In Section 3.2.2 we proposed implementing the reorder buffer (ROB) as a linked-list to support arbitrary instruction insertion and removal. To reduce the number of concurrent linked-list operations, we proposed a hierarchical organization composed of ROB segments. The logical (program) order of instructions within a segment corresponds directly with their physical order, as in a conventional ROB. However, the logical ordering among segments varies. In this way, the linked-list data structure need only specify the logical order of physical segments. The complex alternative to this hierarchical approach is to maintain an instruction-granularity linked-list.

A.4.1 Segment size

Maintaining the linked-list mapping is less complex for larger segments. For example, if the number of instructions per segment is equal to the dispatch/retire rate, up to 3 linked-list operations need to be performed each cycle: inserting one segment for dispatching new instructions, removing one segment for retiring instructions, and removing one segment for squashing instructions (we envision a processor that concurrently frees resources held by incorrect control dependent instructions and allocates resources for correct control dependent instructions). Halving the segment size doubles the number of concurrent linked-list operations, resulting in a more complex implementation.

On the other hand, larger segments result in internal fragmentation of ROB entries, i.e. poor ROB utilization. This occurs because segments are allocated as a unit. If fewer instructions are inserted in the window than there are instructions in a segment, space in the segment is wasted. Likewise, some fraction of leading or trailing instructions within a segment may be squashed, also leaving the segment underutilized.

In Figure 14 the ROB segment size is varied. In all cases the total ROB size is 256 instructions and the machine width is 16 instructions per cycle. Segments of 1, 4, and 16 instructions are simulated. 1 instruction per segment amounts to exploiting control independence at the granularity of individual instructions; it is clearly the most flexible approach, resulting in optimal ROB utilization and high performance, but may be overly complex. Using larger segments degrades performance in two ways. First, fragmentation due to insertion and removal of instructions from the middle of the ROB results in wasted buffer space that is not reclaimed until retirement or until the entire segment is squashed. Second, segments must be retired as a unit. This delays reclaiming ROB entries until *all* instructions in the segment are ready to retire.

Both IPC and performance improvement over a processor without control independence (*base*) are shown in Figure 14. For *compress* and *jpeg*, 4-instruction segments exploit control independence as well as 1-instruction segments, and 16-instruction segments reduce performance by less than 5%. Likewise, for *go* and *gcc* 4-instruction segments reduce performance by less than 5%. However, 16-instruction segments reduce the performance improvement due to control independence by half in *gcc* and by a third in *go*. These benchmarks exhibit more fragmentation because their control flow is much more irregular than *compress* and *jpeg*.

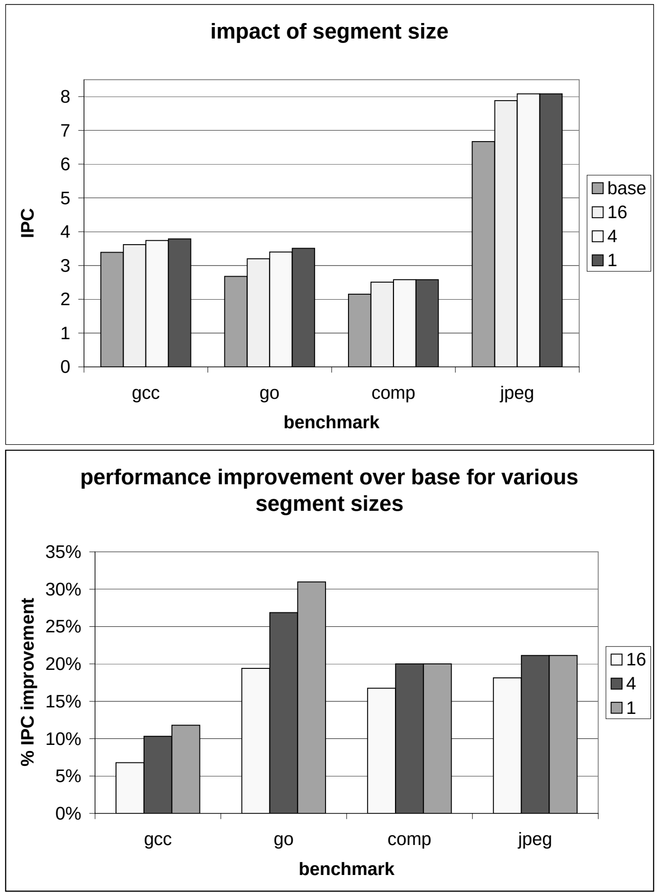


FIGURE 14. Varying ROB segment size.

A.4.2 Control for logically ordering instructions

The processor must maintain the correct program order of instructions for two reasons: in-order retirement and establishing data dependences. Thus far we have only briefly discussed instruction ordering for establishing memory dependences, but it deserves some attention.

A conceptual view of the contents of the linked-list control structure is shown in Figure 15. The structure holds one entry per ROB segment and is indexed by physical segment number. An entry consists of three fields: logical segment number (head segment in the list is logical segment 0), *previous* physical segment number, and *next* physical segment number. Inserting and removing segments (corresponding to allocating and reclaiming segments, respectively) involves updating the *previous* and *next* pointers of logically adjacent segments. Further, inserting or removing a segment requires incrementing or decrementing the logical number of all segments that logically follow the segment.

The first field, called the *physical-to-logical segment translation*, and the *previous-next* pointers are essentially redundant information, since they both represent a linked-list. However, the different representations may simplify different tasks. As will be seen in the next section, the physical-to-logical segment translation may prove useful for resolving memory dependences.

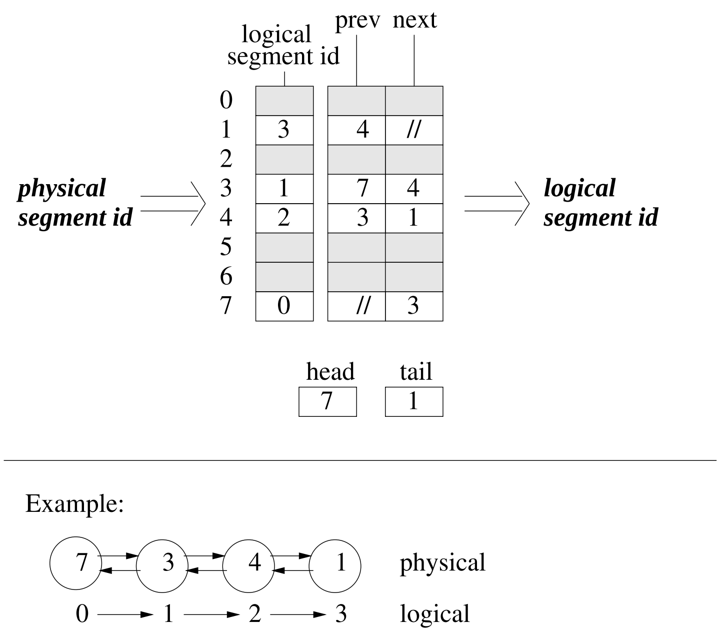


FIGURE 15. Linked-list control structure.

A.4.3 Resolving memory dependences

A scheme for ordering loads and stores based on physical sequence numbers was proposed in the context of trace processors in [1]. Assigning physical sequence numbers based on instruction buffer number to all loads and stores, the mechanism allows for memory operations to be selectively inserted and removed from anywhere within the window, while still maintaining correct load-store ordering. However, the approach relies on a very simple, circular mapping of physical-to-logical sequence number. That is, the processing elements (segments) are organized in a ring.

This requirement is alleviated if a general mechanism is provided to translate physical to logical sequence numbers, like the linked-list control structure in Figure 15. Therefore, we can apply the same memory ordering algorithm used in the trace processor[1], the only changes to the algorithm being a translation step before any sequence number comparison.

A.5 Hardware heuristics for detecting reconvergent points

Thus far we have assumed accurate, per-branch post-dominator information for identifying reconvergent points. In this section we discuss two other general approaches for identifying reconvergence and measure the performance of one of them. Clearly, other heuristics are possible, and hardware identification of reconvergence is a topic for future study.

A.5.1 Associative-search technique

As a restart sequence progresses, one approach is to compare the PCs of the incoming instructions with the PCs of all instructions logically after the mispredicted branch. If the reconvergent point is in the window, in most cases it will be found using this associative-search technique.

1. Because the load-store ordering algorithm is involved, we do not reproduce it here and the reader is referred to [1].

There is one major problem with this approach. Because we do not know before-hand where incorrect control dependent instructions end and control independent instructions begin, dispatching new instructions requires reclaiming instruction buffers from the tail of the reorder buffer, when in fact buffers could be reclaimed from incorrect control dependent instructions first. Thus some control independent instructions are unnecessarily squashed.

A.5.2 Identifying reconvergent points by instruction type

In Section 3.2.1 we proposed examining the dynamic instruction stream for common control flow constructs such as loops and procedures. Both loops and procedures exhibit obvious reconvergence and, as a first approximation, they are identifiable by examining instruction words at decode time.

The following two heuristics identify "global" reconvergent points: these points are not necessarily the precise, i.e. *nearest*, control independent point of any one branch, but they cover regions of branches and their mispredictions.

- procedure return points (*return* heuristic): The decoder identifies all return instructions. The predicted target instruction of a return is remembered as a potential reconvergent point.
- top-of-loop and loop-exit points (*loop* heuristic): The decoder identifies all backward branches by examining branch offsets. The predicted target instruction of a backward branch is remembered as a potential reconvergent point. Depending on the prediction, this may be either the taken or not taken target of the branch, corresponding to the top-of-loop or loop-exit point, respectively.

Whether the *return* and *loop* heuristics are used singly or in combination, the global reconvergent point nearest a mispredicted branch is assumed to be the branch's reconvergent point.

The third heuristic is an example of precisely identifying the reconvergent point of a class of branches.

- mispredicted loop-terminating branches (*ltb* heuristic): If a backward branch is mispredicted, the not taken target of the branch is found in the window and assumed to be the reconvergent point of the branch.

If the *ltb* heuristic is used in conjunction with the *return* and/or *loop* heuristics, the *ltb* heuristic takes priority if the mispredicted branch is a backward branch.

The two global heuristics are shown in Figure 16(a) and the *ltb* heuristic in Figure 16(b). Candidate reconvergent points are marked with a black dot and mispredictions with an X. The *return* heuristic covers all mispredictions within a function, and even some mispredictions before the call if the call is among the control independent instructions. Likewise, the *loop* heuristic covers all mispredictions within a loop and possibly some before the loop. Finally, the *ltb* heuristic specifically and precisely covers the mispredicted backward branch of a loop.

In general, heuristics will not perform as well as complete post-dominator information for the following reasons.

1. Choosing the nearest global reconvergent point from among many in the window will yield no benefit if the chosen point is in the incorrect control dependent path of the mispredicted branch.
2. Even if the chosen global reconvergent point is among the control independent instructions, it may be too distant from the mispredicted branch's immediate post-dominator to yield benefit.

3. There is a case where the *ltb* heuristic fails. If the loop is exited via some other branch, then the not taken target of the mispredicted backward branch is possibly among the incorrect control dependent instructions.

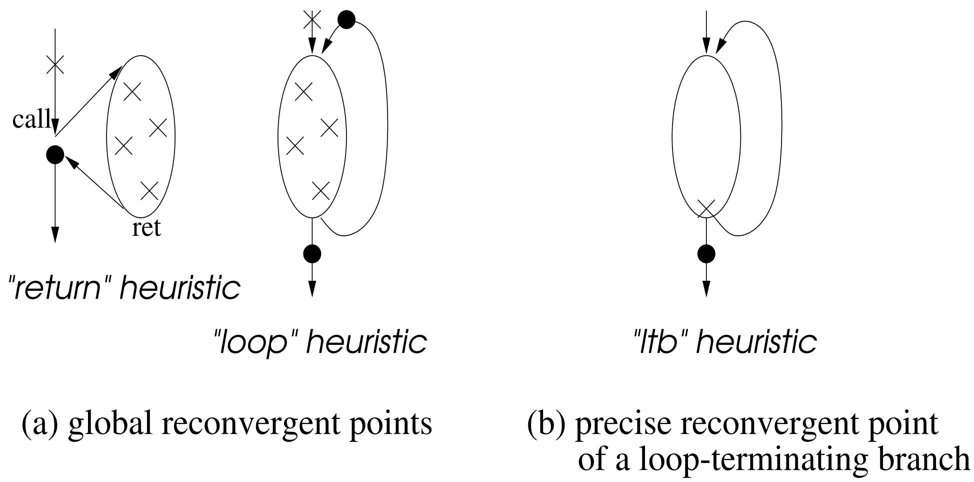


FIGURE 16. Instruction-type heuristics for identifying reconvergent points.

Performance of all combinations of the three heuristics is shown in Figure 17. Performance improvement is measured with respect to a machine with no control independence. For reference, a processor using full post-dominator information is shown as well, labeled *CI*.

When the three heuristics are applied individually (first three bars in Figure 17), the *return* heuristic is generally the best performer. The only exception is *jpeg*, for which the *loop* heuristic performs best. *Jpeg* has one loop in particular that has many internal mispredictions, and control independence is easily exploited across loop iterations.

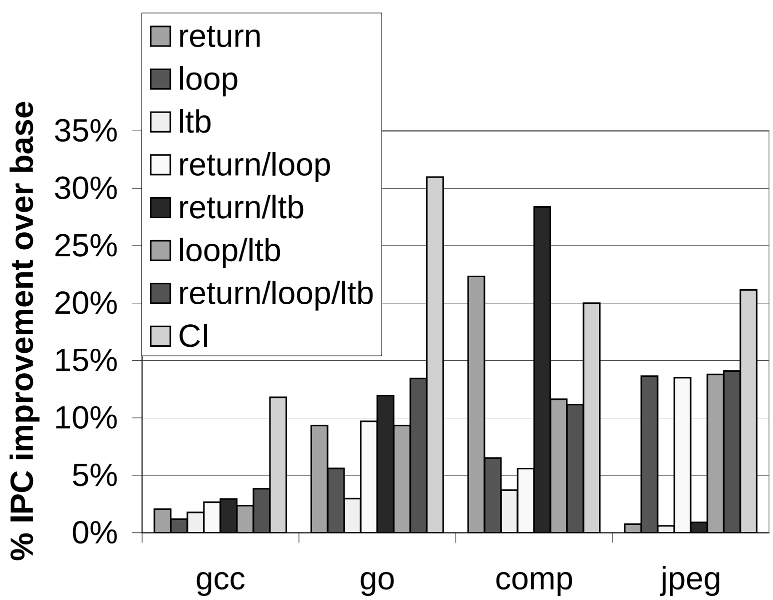


FIGURE 17. Performance of simple instruction-type heuristics for identifying reconvergent points.

Except for *compress*, using all heuristics together (*return/loop/ltb*) yields the best performance. For *gcc*, heuristics achieve only a third of *CI*'s performance potential; for *go*, nearly half of the potential is achieved; and for *jpeg*, nearly three quarters of the potential is achieved.

Interestingly, for *compress,* the *return* heuristic and combined *return/ltb* heuristic perform better than *CI*. Conceivably, heuristics can identify better reconvergent points than a compiler can, as shown in Figure 18. The branch in basic block A is mispredicted in the direction of block B (dashed edge). According to the compiler, block D is the reconvergent point because it is the immediate post-dominator of block A. But if the left edge of block C is taken, then block B is the closest reconvergent point -- *dynamically* the control independent instructions begin with block B. In fact, if the left edge of block C is taken very often (e.g. 99% as shown), then the compiler would be wiser to indicate block B is the immediate post-dominator. In this example, the *return* heuristic by chance selects a reconvergent point that is closer to block A, saving potentially many useful instructions in the region of E.

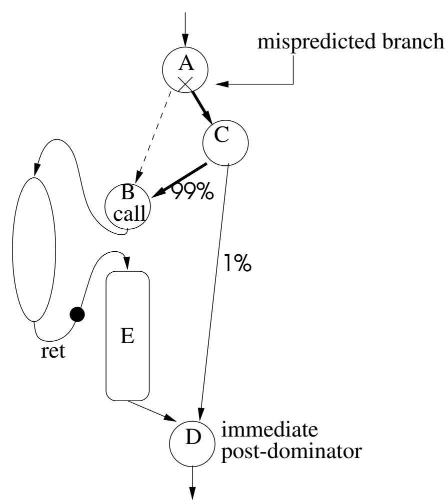


FIGURE 18. An example where the heuristic-based reconvergent point is closer than the compiler-based reconvergent point.

B. A philosophy of control independence

In the introduction to this paper, exploiting control independence is described as "selectively squashing instructions after a branch misprediction to reduce the penalty", primarily because this description is simple. However, there are more fundamental formulations of the problem that, while academic and perhaps not so useful to a designer, I feel provide better motivation for researching control independence. The formulation presented in Section B.1 is based on the view that there are analogs between control dependences and data dependences, and that *conceptually* the same techniques should be applied to both.

In Section B.2, a range of control independence solutions is discussed, focusing on the merits of using multiple flows of control or a single flow of control. To complete the discussion, control independence is contrasted with other branch-misprediction tolerant architectures in Section B.3.

B.1 Control independence is evolutionary

Control and true data dependences in a program impose a partial ordering among instructions to be executed. This ordering can be satisfied trivially by executing instructions in strict program order. However, modern high performance processors use several techniques to more closely approach the partial ordering constraints, and they often go even further by using prediction and speculation to reduce the performance effects of the true dependences. Thus, the techniques applied to control and data dependences can be classified into two categories.

1. *Non-speculative techniques to achieve the partial ordering of true dependences.* This class of techniques has been applied primarily to data dependences. First, to eliminate all but true dependences, *renaming* of register and memory storage is used. Second, to achieve the partial ordering implied by true data dependences, *out-of-order issue* is used.

2. *Speculative techniques to eliminate ordering altogether.* This technique has been applied primarily to control dependences. Predicting branches allows the processor to continue fetching and executing instructions despite unresolved branches. As long as the predictions are correct, all ordering constraints due to control are essentially eliminated.

It is interesting that the dominant processing paradigm (superscalar) has evolved such that the non-speculative techniques are reserved for data dependences and the speculative techniques are reserved for control dependences. There are at least two explanations for this evolution. First, this arrangement may be sufficient. For example, branch prediction techniques are perhaps sufficient to keep processors busy with instructions for the windows being designed today. But clearly, this will not always be the case. Second, this arrangement happens to be the "path of least resistance" for achieving the current level of performance. It is easier to speculate control dependences than data dependences because there are fewer of them, and because they are quite predictable. And as demonstrated in this paper, applying non-speculative out-of-order concepts to control dependences is not particularly intuitive.

Nevertheless, data prediction and speculation techniques are now beginning to appear in the literature [12,34,35], and we argue that non-speculative techniques normally reserved for data dependences should also be considered for control dependences. There are subtle analogies between data and control dependences that suggest conceptually similar solutions.

B.1.1 True dependences

An instruction stalls when its data operands are unavailable. In an in-order machine, all subsequent instructions, whether data dependent or independent of the stalled instruction, must also stall. Instructions are totally ordered at run-time despite the partial ordering implied by data dependences. Similarly, if all instructions after a branch misprediction are squashed and re-fetched, an ordering between these instructions and the mispredicted branch is created despite the partial ordering implied by control dependences.

But neither data stalls nor control mispredictions should force a total ordering. Just as out-of-order issue mechanisms allow *data independent* instructions to proceed despite prior stalled instructions, control independence mechanisms allow *control independent* instructions to proceed despite prior branch mispredictions. The microarchitecture should resolve mispredictions much the same way stalls are resolved. Viewed in this way, control independence is an evolutionary extension of out-of-order instruction issue, generalizing independence and carrying it to its logical conclusion.

B.1.2 Artificial dependences

Anti-dependences, output dependences, and structural hazards are artificial dependences that can be alleviated by renaming registers and memory locations (in the case of anti- and output dependences) and providing more resources in general (structural hazards).

In terms of control flow, the single program counter introduces an artificial dependence, because instructions are fetched sequentially and not necessarily in the order in which they are needed. For example, there may be several independent instructions that are ready to issue but are too far into the instruction stream to be reached by the PC. The PC must first sequence through less urgent instructions to get to the ready instructions. The single PC is a resource limitation that can artificially delay the critical path through the program, just as a lack of registers or functional units artificially delays execution. To alleviate this, the single PC can be "renamed" into multiple PCs just as a single architected register can be renamed into multiple physical registers.

The following architectures implement multiple program counters either directly or implicitly.

- VLIW: Hardware maintains a single PC, but the compiler prepares instructions such that the order in which they are fetched is identical to the order in which they issue.
- Wide superscalar: A single PC may not be so much of a bottleneck if it is a "wide PC", that is, if many instructions can be brought in at once. Much of the effect of multiple control flows may be realizable, but the solution is somewhat brute-force. On the other hand, it is robust in that it does not rely on the compiler or hardware doing a good job of *placing* multiple program counters across the dynamic instruction stream.
- Multiscalar and multithreading: Architecturally, there is only a single logical PC. But the hardware maintains multiple physical program counters, and the placement of the program counters across the dynamic instruction stream is guided by the compiler (although a fully-dynamic scheme is possible).
- Dataflow: There is essentially an unlimited number of control flows, dictated by the data flow graph of the program.

B.2 Control independence architectures

Control independence is a property of a dynamically executed program. Ways of exploiting control independence can vary with the hardware and software techniques being used. We identify two general classes of implementations (although hybrids are possible).

- *Multiple flows of control with a noncontiguous instruction window.* This class of machines has multiple instruction fetch units and can simultaneously fetch from disjoint points in the dynamic instruction stream. The instruction window, i.e. the set of instructions simultaneously being considered for issue and execution, does not have to be a contiguous block from the dynamic instruction stream. Clearly, control independent code regions are good candidates for parallel fetching, though this is not a requirement. Multiscalar processors and parallel multiprocessors fall into this class.
- *Single flow of control with a contiguous instruction window.* This class of machines has a single program counter and can fetch along a single flow of control at any given time. The instruction window is a contiguous set of dynamic instructions. Control independence is implemented by allowing the program counter to skip back and forth in the dynamic instruction stream. (This paper focuses on this class of machines.)

Each class of machines has advantages. With implementations having multiple flows of control, there is a natural hierarchical structure: each flow of control fetches and operates on its own "task" or thread. Control decisions are separated into inter-task and intra-task levels. Intra-task mispredictions can be isolated to the task containing the misprediction, and later control independent tasks can proceed in a fairly straightforward manner. This hierarchical task-based structure leads to what is effectively a non-contiguous instruction window where instructions can be fairly easily inserted and removed as control mispredictions occur. Further, the hierarchy allows for multiple branch mispredictions to be serviced simultaneously if they are in different tasks.

An advantage of a single control flow implementation is that the single fetch unit can scan all the instructions as it builds the single instruction window and, therefore, has more complete knowledge of potential dependences. This leads to more robust and less conservative data dependence resolution and recovery mechanisms (discussed below). In addition, these methods may be able to take advantage of finer grain control independence, at the level of individual basic blocks, for example.

The aggressive data dependence resolution and recovery mechanisms presented in this paper are important distinctions with other control independence architectures. Specifically, *some* design points of the multiscalar and multithreading approaches resolve inter-thread data dependences conservatively [29]. That is, even though control flow within a thread does not directly affect other threads, values dependent on the control flow are not forwarded to other threads until the control flow is resolved. If speculative data forwarding is performed, entire threads are squashed when incorrect values are referenced, losing some or all of the benefits of control independence. This is only true for designs without selective reissuing capability, e.g. large threads may preclude being selective. In a sense, this approach to control independence more closely resembles guarding [36,37,8,9], which shifts the problem of control flow to data flow. But clearly these are not fundamental restrictions [38]; conservatism reflects a simpler and perhaps more practical design.

B.3 Other misprediction-tolerant solutions

B.3.1 Instruction reuse

Instruction reuse [18] is a mechanism that exploits control independence. Rather than explicitly preserving instructions within the *instruction window*, input and output values of completed instructions are buffered in a *cache-like structure*. When a misprediction is detected, the instruction window is not preserved, but the control and data independent state of the window is in some sense restored from the reuse buffer. Control independent instructions that were written into the reuse buffer before the misprediction is detected, and whose inputs do not change due to the misprediction, bypass re-execution.

The reuse buffer greatly simplifies preserving the instruction window. In addition to its simplicity, there are at least two performance advantages of instruction reuse with respect to explicit control independence. First, if the incorrect control dependent path is shorter than the correct control dependent path, more control independent instructions can be executed and preserved in the reuse buffer than can be preserved in the instruction window (the additional control independent instructions are "pushed out" of the window by the longer, correct control dependent path). Second, instruction reuse is a unified approach for exploiting both control independence (squash reuse) and *general reuse*.

Reuse has potential disadvantages, however, when compared with explicitly preserving instructions in the window. First, with explicit control independence, control independent instructions that have not issued, executed, or broadcast their results by the time the misprediction is detected may continue processing in spite of the misprediction. Instruction reuse may not capture these instructions. With very large instruction windows, explicitly preserving instructions in the window and allowing work to proceed in parallel with servicing mispredictions may account for much of the benefit of control independence; this is an area that deserves further study. Second, because instructions are stored in the reuse buffer based on PC, the number of dynamic instances of an instruction that may be recovered is constrained by the associativity of the reuse buffer. This may be a problem for instructions in loops. Clearly, other reuse buffer organizations may overcome this limitation.

Instruction reuse requires re-fetching instructions. On the other hand, conceivably there are explicit control independence implementations that do not require re-fetching and re-dispatching instructions. More advanced register repair models than those proposed in this report are possible. However, re-fetching may be necessary for maintaining high prediction accuracy -- this was discussed in Appendix A.3.2 in terms of the need for re-predict sequences.

B.3.2 Predication and selective multi-path execution

Predication [36,37,8,9] and selective multi-path execution [2,3,4,5,6,7] attempt to identify hard-to-predict branches, either through profiling or branch confidence estimators (respectively), and fetch both paths of these branches. In the case of multi-path execution, both paths are fully renamed and executed as separate threads. When the branch is resolved, one of the threads is squashed and the other becomes the primary thread of execution.

Predication is in some sense the software equivalent of multi-path execution applied to forward-branching regions of the CFG. In one form of predication, the control dependent instructions do not execute until their predicates are computed, i.e. multiple paths are fetched but only the correct path is executed. Alternatively, with *predicate promotion* [39] or *predicated state buffering* [9], instructions from multiple paths may execute concurrently, and only the results from the correct path are committed.

Predication and multi-path execution waste resources by fetching and possibly executing both the correct and incorrect control dependent paths of branches. This results in a performance gain over conventional speculation if the branches are mispredicted. Unfortunately, multi-path execution is applied to some fraction of correctly predicted branches, and alternatively, some fraction of incorrectly predicted branches are not covered by multi-path execution. In our experience with static and dynamic confidence estimation [33], it is not often the case that specific branches are always predicted correctly or incorrectly. Rather, most branches -- or patterns in the case of dynamic schemes -- identified as "unpredictable" are actually in a gray area, with prediction accuracies of 80% or more. *To cover a significant fraction of mispredictions, an even larger number of correct predictions must also be covered.*[1]

A problem specific to predication is the aggravation of data dependences. The purpose of branch prediction is two-fold: (1) quickly determine which instructions to fetch next and (2)

1. For example, a dynamic confidence mechanism can concentrate 90% of all mispredictions within 20% of all dynamic predictions for the IBS benchmarks [33]. Assuming a 90% branch prediction accuracy, this means 9% of predictions are correctly identified for multi-path execution, 11% of predictions are incorrectly identified for multi-path execution, and 1% of predictions are not identified for multi-path execution when they should be. For a static profiling scheme, which predication may rely on, the same numbers are 6%, 14%, and 4% respectively, to concentrate 60% of all mispredictions within 20% of all dynamic predictions.

quickly establish and resolve data dependences among instructions. Predication only addresses the first aspect. It “removes” branches, so the instructions to be fetched are known in advance (all instructions in the predicated region are fetched). It does not, however, address the second aspect. Without predicated state buffering, all predicated instructions must wait for their controlling predicate to be resolved. Branch prediction eliminates this control dependence if the prediction is correct, and it is correct more often than incorrect. With predicated state buffering, instructions within a region need not wait for predicates, but their computed results are not forwarded *outside* the region until predicate conditions are resolved.

Predication and multi-path execution can potentially reduce the branch misprediction penalty more than control independence, because only part (or none) of the path after the branch is recovered in the case of control independence. On the other hand, because only a single path is followed, control independence may still capture more control independent instructions within the window than predication or multi-path execution.

The idea behind control independence is to always trust branch prediction and speculation, and take measures only when a misprediction occurs, thereby avoiding the above difficulties. After all, branch prediction performs well most of the time, so it makes sense to exploit its potential fully and employ other optimizations when it does not perform.

References

[1] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

[2] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. *28th Intl. Symp. on Microarchitecture*, Dec 1995.

[3] T. Heil and J. Smith. Selective dual path execution. Technical report, University of Wisconsin, ECE Department, Nov 1996.

[4] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR-346-97, University of Michigan, EECS Department, 1997.

[5] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. *25th Intl. Symp. on Computer Architecture*, June 1998.

[6] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. *25th Intl. Symp. on Computer Architecture*, June 1998.

[7] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath execution: Opportunities and limits. *Intl. Conf. on Supercomputing*, July 1998.

[8] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu. A comparison of full and partial predicated execution support for ilp processors. *22nd Intl. Symp. on Computer Architecture*, June 1995.

[9] H. Ando, C. Nakanishi, T. Hara, and M. Nakaya. Unconstrained speculative execution with predicated state buffering. *22nd Intl. Symp. on Computer Architecture*, June 1995.

[10] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *19th Intl. Symp. on Computer Architecture*, pages 46–57, May 1992.

[11] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Nov 1993.

[12] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[13] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. *22nd Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[14] P. Dubey, K. O’Brien, K. M. O’Brien, and C. Barton. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. *Intl. Conf. on Parallel Architecture and Compilation Techniques*, 1995.

[15] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. *Intl. Conf. on Parallel Architecture and Compilation Techniques*, 1996.

[16] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism in multiprocessors. Technical Report CSL-TR-97-715, Stanford University, Computer Systems Laboratory, Feb 1997.

[17] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallel-

ization. *4th Intl. Symp. on High Performance Computer Architecture*, Feb 1998.
[18] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *24th Intl. Symp. on Computer Architecture*, June 1997.
[19] K. Sundararaman and M. Franklin. Multiscalar execution along a single flow of control. *ICPP'97*, Aug 1997.
[20] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *24th Intl. Symp. on Computer Architecture*, pages 1–12, June 1997.
[21] M. Lipasti and J. Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer, Billion-Transistor Architectures*, Sep 1997.
[22] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer, Billion-Transistor Architectures*, Sep 1997.
[23] J. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer, Billion-Transistor Architectures*, Sep 1997.
[24] S. McFarling. Combining branch predictors. Technical Report TN-36, WRL, June 1993.
[25] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. *24th Intl. Symp. on Computer Architecture*, June 1997.
[26] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th Intl. Symp. on Computer Architecture*, pages 34–42, May 1991.
[27] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin, CS Department, July 1996.
[28] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. *19th Intl. Symp. on Computer Architecture*, May 1992.
[29] T. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin, Jan 1998.
[30] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *ACM Conf. on Programming Language Design and Implementation*, June 1991.
[31] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. *ACM Symp. on Principles of Programming Languages*, Jan 1989.
[32] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.
[33] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. *29th Intl. Symp. on Microarchitecture*, pages 142–152, Dec 1996.
[34] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. *29th Intl. Symp. on Microarchitecture*, pages 238–247, Dec 1996.
[35] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report 1080, Technion - Israel Institute of Technology, EE Dept., Nov 1996.
[36] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. *10th Symp. on Principles of Programming Languages*, Jan 1983.
[37] D. Pnevmatikatos and G. Sohi. Guarded execution and branch prediction in dynamic ilp processors. *21st Intl. Symp. on Computer Architecture*, April 1994.
[38] T. N. Vijaykumar, S. E. Breach, and G. S. Sohi. Register communication strategies for the multiscalar architecture. Technical Report 1333, CS Dept., Univ. of Wisc. - Madison, Feb 1997.
[39] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipelined architectures. *Supercomputing '90*, Nov 1990.