A Study of Control Independence in Superscalar Processors

Eric Rotenberg*, Quinn Jacobson, Jim Smith
Computer Sciences Dept.* and Dept. of Electrical and Computer Engineering
University of Wisconsin - Madison

Abstract

Control independence has been put forward as a significant new source of instruction-level parallelism for 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.

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 10-30%.

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 many years. 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 [1-6]. Predicated execution is a software method for achieving a similar effect [7, 8]. 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 [9, 10]. This can be illustrated with a simple example.

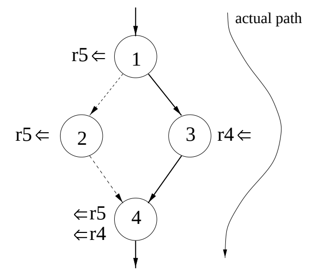


FIGURE 1. An example of control independence.

Figure 1 shows a control flow graph (CFG) containing four basic blocks. Basic blocks are used for simplicity and may be substituted with arbitrary control flow. The 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.

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 mispredicted branch, 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 [11, 12] in some form is necessary.

Lam and Wilson's limit study on control independence [9] 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 [10,12-19]. 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 artificial design limitations.

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.

1.1 Prior work

Lam and Wilson's limit study [9], and a similar study by Uht and Sindagi [1], demonstrates that control independence exposes a large amount of instruction-level parallelism, on the order of 10 to 100. 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 [20], 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 more accurate dynamic branch predictors. More importantly, because the simulation is fully trace-driven, it does not account for false data dependences created on mispredicted paths, 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 fundamental processor features. There is no concept of a limited instruction window or instruction fetch bandwidth, whether considering a single or multiple flows of control. The entire dynamic instruction stream is scheduled at once; exposing the observed parallelism may require buffering speculative state for thousands of instructions and using an impractical number of parallel fetch units.

Multiscalar processors [10,13] and other speculatively multithreaded architectures [14-17,19] 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, which may contain arbitrary control flow. 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 [12,21] are 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 are the same as those already residing in the processing elements [12]. Only modest improvements are reported because no optimization in trace selection or processor assignment was done to expose 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. 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 fundamental 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).

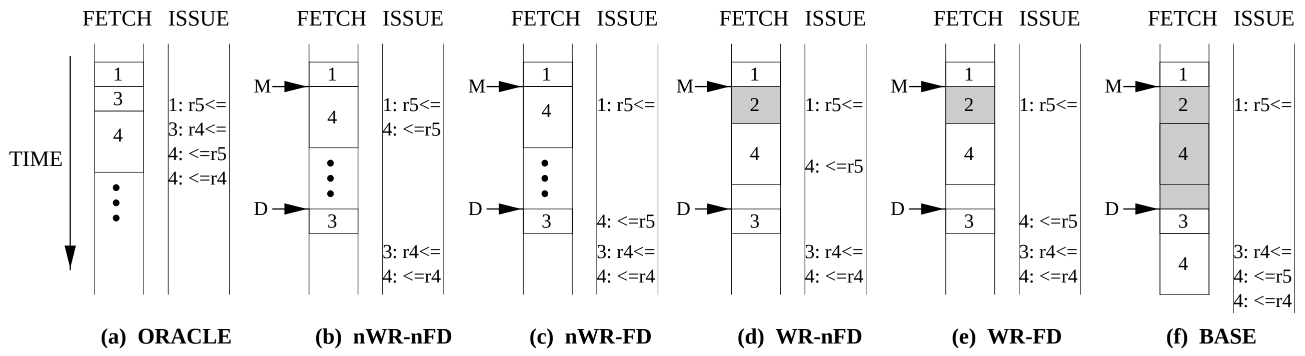


FIGURE 2. Fetch and issue timing for the six models, corresponding to the example CFG in Figure 1.

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*.

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. We implement the following additional hardware constraints and assumptions:

- Instruction fetch is ideal: up to 16 instructions, including any number of branches, can be fetched every cycle.
- Instruction fetch, dispatch, issue, execute, and retire stages are modeled. Fetch and dispatch take 1 cycle each. Issue takes at least 1 cycle, possibly more if the

instruction must stall for operands. Execution takes a fixed latency based on instruction type, plus any time spent waiting for a result bus. Address generation takes 1 cycle, and all data cache accesses are 1 cycle (i.e. perfect data cache). Instructions retire in order.

- Any 16 ready instructions may issue in a cycle.
- Output and anti-dependences for both registers and memory are eliminated (i.e. perfect renaming).
- Oracle memory disambiguation is used. (However, stores fetched down the wrong control path may still interfere with subsequent, control independent loads.)
- A 2^{16}-entry *gshare* predictor [22] is implemented for predicting the direction of conditional branches. All direct target addresses are assumed to be predicted correctly. For indirect calls and jumps, a 2^{16}-entry correlated target buffer [23] is used. Returns are predicted using a perfect return address stack [24].

2.3 Benchmarks

Dynamic instruction traces, including both correctly speculated and misspeculated instructions, are generated by the Simplescalar simulator [25]. Five integer SPEC95 benchmarks -- chosen to reflect a variety of prediction accuracies (Table 1) -- were simulated to completion.

TABLE 1. Benchmark information.

benchmark	input dataset	dyn. instr. count	misp. rate
gcc	-O3 genrecog.i	117 M	8.3%
go	9 9	133 M	16.7%
compress	400000 e 2231	104 M	9.1%
ijpeg	vigo.ppm	166 M	6.8%
vortex	modified train input	101 M	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 prediction accuracy is quite high. Performance for the *base* model typically saturates at a window size of 128 or 256. 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. 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 [9], because misspeculated instructions are ignored.

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*.

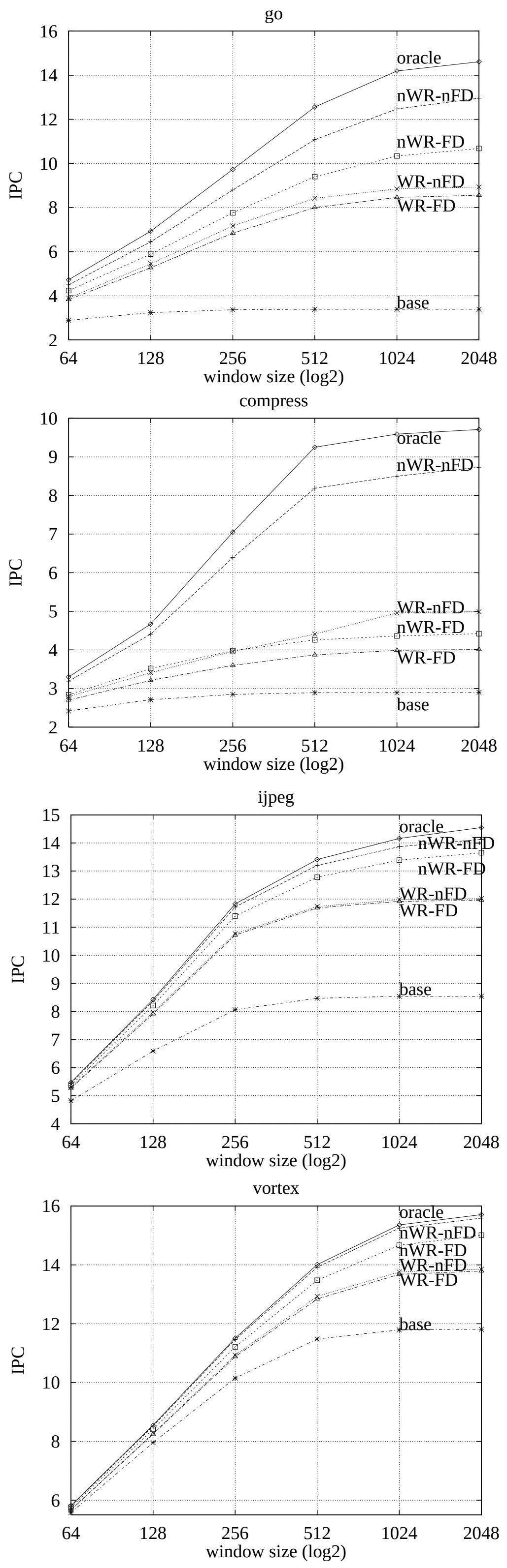
The other three control independence models also have interesting implications. A major performance limiter is the incorrect control dependent path, primarily because of wasted fetching and window space (*WR-nFD*), but also false data dependences (*nWR-FD*). If these limitations could be mitigated in 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" [10]. 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 [26] 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.



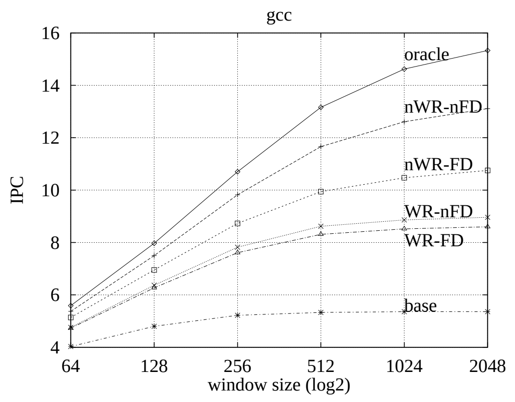


FIGURE 3. Performance of the six models.

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, as illustrated in Figure 4 and described below. Steps 1-3 below constitute the *restart sequence*, and step 4 the *redispatch sequence*.

1. After detecting a branch misprediction, the first control independent instruction (if it exists) must be found in the 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.
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.

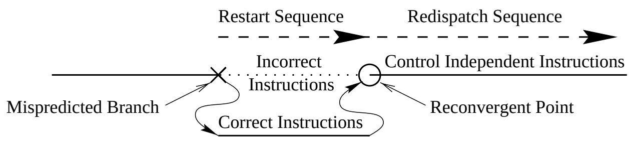


FIGURE 4. Misprediction recovery sequence.

3.2 Key microarchitecture mechanisms

To support the above recovery steps, we have identified four underlying microarchitecture mechanisms. 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.

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 [27]. 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 difficult using hardware alone. Software can aid the hardware by encoding this information. For example, the compiler could encode this information by including in each branch instruction a small offset to its post-dominator instruction. 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 imprecise heuristics. One alternative is to exploit easily-identified control flow constructs such as loops and functions. The targets of subroutine return instructions and backward branches are detectable by hardware, and they may serve as "global" reconvergent points. While these points are not the precise, i.e. nearest, reconvergent point of any particular branch, they often identify a subset of control independent instructions common to many branches in a region. Hardware can easily detect and record the location of such points in the window, and when a misprediction is detected, the nearest such point is assumed to be the correct reconvergent point.

3.2.2 Instruction removal/insertion

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.

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.

During the restart sequence, resources (physical registers and load/store buffers) of squashed control dependent instructions are iteratively reclaimed. In parallel, as the correct control dependent path is fetched, new instructions may acquire the resources freed by the old instructions. If there are more correct control dependent instructions than incorrect ones, the resources of control independent instructions, youngest first, are reclaimed to make room.

3.2.3 Forming correct data dependences

Although instructions may be *control* independent with a preceding block of instructions, they may not be *data* independent. Consequently, both register and memory dependences of control independent instructions must be repaired after a misprediction.

When the restart sequence completes, the register rename maps reflect state up to the re-convergent point. Control independent instructions are redispatched [12] using the up-to-date register maps. During redispatch, source operands are remapped while destination operands maintain their original assignments. If an instruction’s source operand is mapped to a new physical register, the instruction reissues with new data.

To repair memory dependences, the memory-ordering mechanism detects when a preceding store is removed or inserted by a restart sequence and directs affected loads to reissue. An implementation can be found in [12].

3.2.4 Selective reissuing of instructions

If a control independent instruction reissues due to incorrect register/memory dependences, then subsequent data dependent 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 [11,12]. 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. 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 [12] -- 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 and temporarily incorrect global history.

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.

Instruction removal/insertion is implemented via the linked list approach, using single-instruction granularity.

Forming correct data dependences is delayed a variable number of cycles after the misprediction is detected, unlike the ideal study, because (1) the redispatch sequence cannot proceed until after the restart sequence completes and (2) redispatch proceeds at the maximum dispatch rate.

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 in *gcc* 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 small -- between 1% and 4% -- meaning the time spent during redispatch sequences has less impact than anticipated.

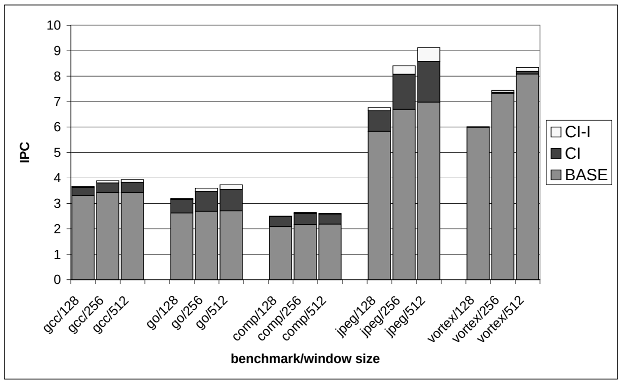


FIGURE 5. Performance of the three models.

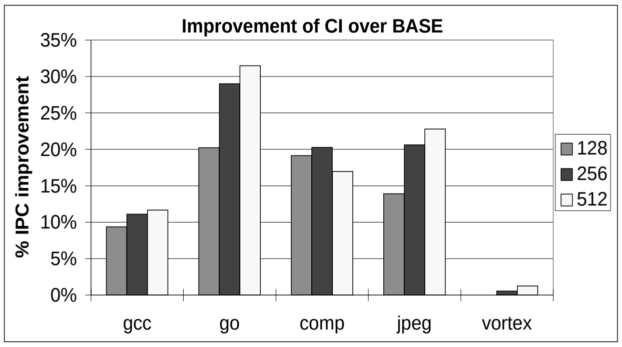


FIGURE 6. Percent improvement in IPC.

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 a 256-instruction window.

The first row of Table 2 shows how often a control independent reconvergent point is in the window at the time a misprediction is detected. Except for *vortex*, a reconvergent point is present for over 60% of mispredictions.

The second and third rows of Table 2 show the average number of instructions removed and inserted *for those restart sequences that reconverge in the window.* On average, fewer than 14 incorrect control dependent instructions are removed, and fewer than 20 correct control dependent instructions are inserted. For over 80% of the restarts that reconverge in the window, both the number of instructions inserted and removed is fewer than 32 (not shown in table).

The fourth row in Table 2 shows that the average number of control independent instructions after the reconvergent point is greater than 50 for all the benchmarks. The fifth row 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 (as well as subsequent data dependent instructions).

The last row in Table 2 shows the amount of useful work that can be saved with control independent instructions. Ignoring *vortex*, 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.

TABLE 2. Control independence measures.

statistic	gcc	go	comp	jpeg	vortex
% of misp. that reconverge	62%	71%	91%	82%	47%
# removed ctl. dep. instr.	13.2	13.5	6.8	9.0	9.2
# inserted ctl. dep. instr.	16.5	18.1	6.6	10.7	12.8
# control indep. instr.	51.8	62.4	122	79.8	81.5
# instr. w/ new reg. names	2.8	2.2	1.7	2.2	2.1
work saved	20%	30%	39%	11%	4%

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 proposed to address some of the more complex aspects, such as the segmented ROB for arbitrary insertion/removal of instructions, and hardware heuristics for identifying reconvergent points. Detailed simulations of a superscalar processor implementing the key features show typical performance improvements of 10-30%, derived from the 20% 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 conventional 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* model suggests combining the expandable window model of multiscalar processors with the aggressive data dependence resolution and recovery model of trace processors.

A much more comprehensive treatment of control independence can be found in [28], an extension of this paper.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. Eric Rotenberg is supported by an IBM Fellowship, and Quinn Jacobson by an Intel Fellowship.

References

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

[2] T. Heil and J. Smith. Selective dual path execution. Technical report, Univ. of Wisc., ECE Dept., Nov 1996.

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

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

[5] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. *25th Intl. Symp. on Comp. Arch.*, June 1998.

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

[7] 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 Comp. Arch.*, June 1995.

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

[9] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *19th Intl. Symp. on Comp. Arch.*, May 1992.

[10] M. Franklin. *The Multiscalar Architecture*. PhD thesis, Univ. of Wisc., Nov 1993.

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

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

[13] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. *22nd Intl. Symp. on Comp. Arch.*, 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. *PACT*, 1995.

[15] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. *PACT*, 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, CSL, Feb 1997.

[17] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *4th Intl. Symp. on High Perf. Comp. Arch.*, Feb 1998.

[18] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *24th Intl. Symp. on Comp. Arch.*, June 1997.

[19] H. Akkary and M. Driscoll. A dynamic multithreading processor. *31st Intl. Symp. on Microarchitecture*, Dec 1998.

[20] K. Sundararaman and M. Franklin. Multiscalar execution along a single flow of control. *ICPP’97*, Aug 1997.

[21] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *24th Intl. Symp. on Comp. Arch.*, June 1997.

[22] S. McFarling. Combining branch predictors. Technical Report TN-36, WRL, June 1993.

[23] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. *24th Intl. Symp. on Comp. Arch.*, June 1997.

[24] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th Intl. Symp. on Comp. Arch.*, May 1991.

[25] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, Univ. of Wisc., CS Dept., July 1996.

[26] T. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, Univ. of Wisc., Jan 1998.

[27] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. *Symp. on Principles of Prog. Languages*, Jan 1989.

[28] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. Technical Report 1389, Univ. of Wisc., CS Dept., Nov 1998.