Control Independence in Trace Processors

Eric Rotenberg
Dept. of Electrical and Computer Engr.
North Carolina State University

Jim Smith
Dept. of Electrical and Computer Engr.
University of Wisconsin - Madison

Abstract

Branch mispredictions are a major obstacle to exploiting instruction-level parallelism, at least in part because all instructions after a mispredicted branch are squashed. However, instructions that are control independent *of the branch must be fetched regardless of the branch outcome, and do not necessarily have to be squashed and re-executed. Control independence exists when the two paths following a branch re-converge.*

A trace processor microarchitecture is developed to exploit control independence and thereby reduce branch misprediction penalties. There are three major contributions. 1) *Trace-level re-convergence is not guaranteed despite re-convergence at the instruction-level. Novel trace selection techniques are developed to expose control independence at the trace-level.* 2) *Control independence's potential complexity stems from insertion and removal of instructions from the middle of the instruction window. Trace processors manage control flow hierarchically (traces are the fundamental unit of control flow) and this results in an efficient implementation.* 3) *Control independent instructions must be inspected for incorrect data dependences caused by mispredicted control flow. Existing data speculation support is easily leveraged to selectively re-execute incorrect-data dependent, control independent instructions.*

For five of the SPEC95 integer benchmarks, control independence improves trace processor performance from 5% to 25%, and 17% on average.

1. Introduction

Dynamically scheduled superscalar processors achieve high performance by extracting instruction-level parallelism (ILP) from ordinary, sequential programs. Because there are data dependences among instructions, finding a sufficient number of independent instructions to execute in parallel requires examining and scheduling a large group of instructions, called the *instruction window*. The larger this window, the farther a processor may "look ahead" into the program — and the greater the chance of finding independent instructions.

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. High performance processors deal with control dependences by using branch prediction. Predicting branch outcomes allows instruction fetching and speculative execution to proceed despite unresolved branches in the window. Unfortunately, branch mispredictions still occur, and current 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 execution proceeds at full efficiency.

Often only a subset of dynamic instructions immediately following a branch truly depend on the branch outcome, however. 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 [1,2].

Control independence typically occurs when the two paths following a branch re-converge before the control independent instruction, as depicted in Figure 1. Upon detecting the misprediction, the incorrect control dependent instructions are squashed and replaced with the correct control dependent ones, but processing of control independent instructions can proceed relatively unaffected. Exploiting control independence requires three basic mechanisms outlined below.

1. The re-convergent point must be identified in order to distinguish and preserve the control independent instructions in the window.
2. The processor must support insertion and removal of control dependent instructions from the *middle* of the window.
3. The mispredicted control flow may cause some incorrect data dependences to be formed between control independent instructions and instructions before the re-convergent point. These incorrect data dependences must be repaired and the incorrect-data dependent, control independent instructions selectively re-executed.

Control independence is an effective technique for mitigating the effects of branch mispredictions. A recent study shows potential performance improvements of 30% in wide-issue superscalar processors [3]. However, practical mechanisms for the three outlined requirements need to be explored. In [3], control dependence information is ideally conveyed from the compiler to hardware for identifying re-convergent points, yet simple hardware-only detection of re-convergent points is desirable. And the reorder buffer of superscalar processors is managed as a fifo with insertion and removal of instructions performed only at the head and tail of the fifo, not in the middle. Arbitrary expansion and contraction, beginning and ending at *any point* in the window, may be complex. Finally, conventional register and memory dependence mechanisms are inadequate for control independence. A new overall data flow management strategy is needed — one that supports data speculation in general.

In this paper, the *trace processor* microarchitecture [4,5] is explored as a practical and effective platform for control independence.

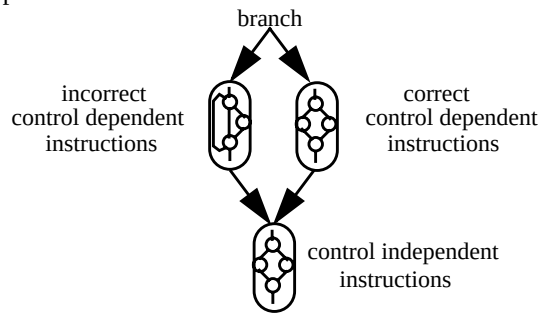


FIGURE 1. Control independence example.

1.1. Trace processor control independence

A trace processor (Figure 2) is organized entirely around *traces*, long dynamic instruction sequences spanning multiple basic blocks and constrained primarily by a hardware-determined maximum length. Traces introduce a larger granularity of work and therefore hierarchy.

Rather than predicting, fetching, and dispatching/ renaming instructions as individual units, the frontend of the trace processor works more efficiently at the higher level of traces. Traces are predicted using a next-trace predictor [6] which implicitly predicts multiple branches each cycle with only a single trace prediction. Traces themselves are stored in a trace cache [7,8,9,10] for low-latency, high-bandwidth instruction fetching. Traces are also efficiently dispatched and renamed as a unit [4]: *intra-trace values* are pre-renamed in the trace cache, so only *inter-trace values* (live-in and live-out registers) need to be dynamically renamed.

The instruction window and issue mechanisms are distributed among multiple processing elements (PEs) [4]. Each PE is allocated a single trace. Fast instruction issue and bypassing of intra-trace values are possible due to a small (trace-sized) window and modest, dedicated issue bandwidth within each PE.

Trace processor control independence mechanisms are described in the following three subsections. We begin with the hierarchical instruction window and how it is inherently suited to flexible window management. Then, the interesting problem of ensuring and identifying *trace-level re-convergence* is described. Finally, we describe how the selective misspeculation recovery model of trace processors [5] supports the data flow management requirements of control independence.

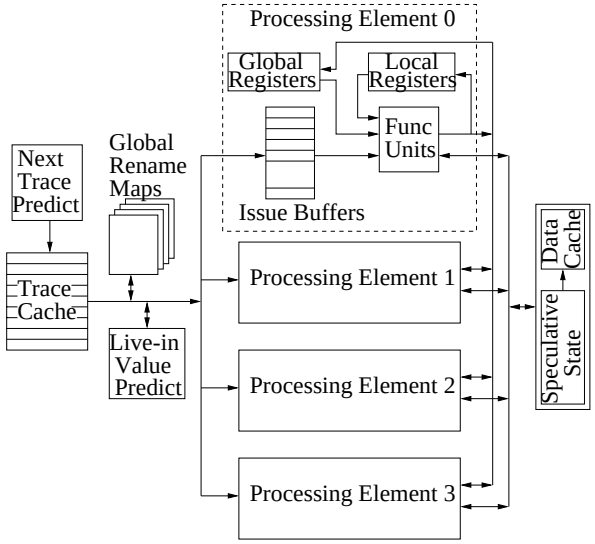


FIGURE 2. Trace processor.

1.1.1. Exploiting hierarchy: flexible window management. The hierarchical instruction window enables flexible window management in two ways.

1. *Hierarchical management of control flow.* In some cases it is possible to isolate the effects of *intra-trace control flow* from *inter-trace control flow.* This is true if the longest control dependent path of a branch fits entirely within a trace. If such a branch is mispredicted, instructions following the branch but in the same trace are squashed, while subsequent traces are not squashed. Within a PE, a simple (non-selective) squash model is preserved, yet at a higher level it appears as if instructions are inserted/removed from the *middle* of the instruction window.

   This is called *fine-grain control independence (FGCI)* because the branch and its re-convergent point are close together. Figure 3(a) shows an example in which a misprediction in *PE1* affects only control flow within the PE, and inter-trace control flow (links between PEs) is unaffected.

2. *Hierarchical management of resources*. If one or more control dependent paths of a branch are longer than a trace, then recovering from a misprediction involves squashing and inserting an arbitrary number of traces in the middle of the window. To do so, the PEs are managed as a linked-list instead of a fifo. This is no more complex than fifo management, however, because the unit of insertion/removal (a trace) is large and therefore efficient to manage.

   This is called *coarse-grain control independence (CGCI)* because the branch and its re-convergent point are in different traces. Figure 3(b) shows an example in which two traces *t2* and *t3* must be removed from the middle of the instruction window and trace *t6* inserted in their place.

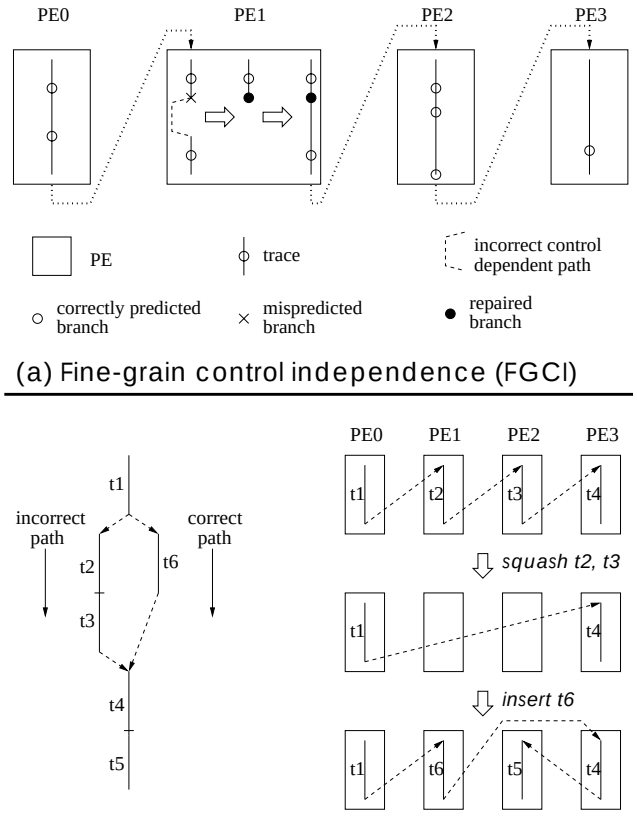


FIGURE 3. Flexible window management.

1.1.2. Trace selection: ensuring and identifying trace-level re-convergence. Trace-based window management simplifies arbitrary instruction insertion and removal but introduces a new problem. Although control flow eventually re-converges after a branch, *trace-level re-convergence* is not guaranteed. Consider Figure 4(a), in which the two control dependent paths of the branch are of different lengths. In Figure 4(b), a different set of traces is selected depending on the direction of the branch, even traces after the re-convergent point. I.e. re-convergence is not manifested at the trace-level.

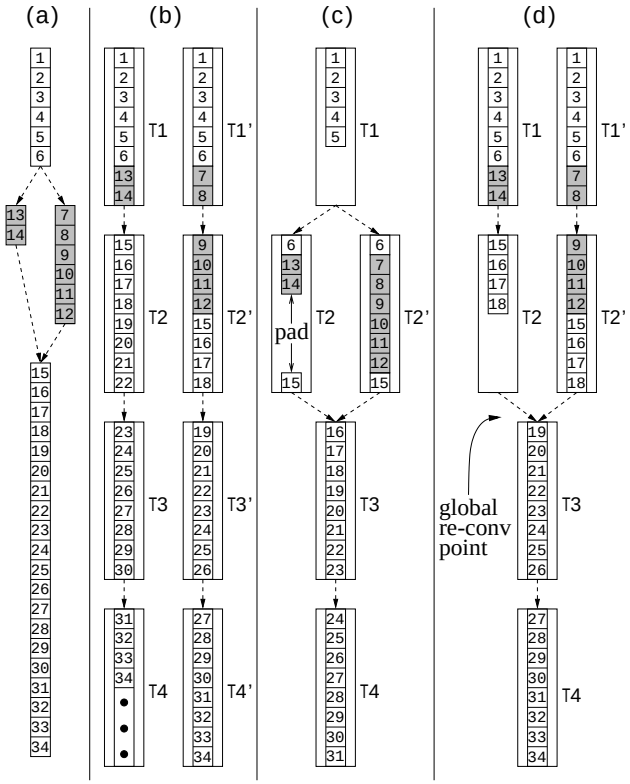


FIGURE 4. Trace-level re-convergence problem.

Trace-level re-convergence must be ensured in order to exploit both FGCI and CGCI. This rests with *trace selection*, the algorithm for dividing the dynamic instruction stream into traces. Essentially, trace selection must synchronize the control dependent paths of a branch so that regardless of which path is taken, the same sequence of control independent traces are selected. Traces may be synchronized at the re-convergent point itself, but more generally at any control independent point after the re-convergent point. We develop two different trace selection techniques to address FGCI and CGCI separately.

Small `if-then`, `if-then-else`, and nested `if-then-else` constructs that do not contain loops or function calls are ideally suited to FGCI mechanisms. First, they have fixed-length and relatively short control dependent paths, most of which fit within a trace. Table 5 in Section 6.2 shows that in the worst case, less than 10% of these constructs have a control dependent path longer than 32 instructions. Secondly, they account for a large enough fraction of mispredictions (20% - 60%) to be specially targeted for control independence. Lastly, these regions can

be precisely and efficiently detected by hardware because of their directed, acyclic control flow.

We propose a hardware algorithm that detects these forward-branching regions, locates the re-convergent point that closes the region, and computes the length of the longest control dependent path through the region. Trace selection then uses this information to conceptually "pad" any selected path until its length matches the longest path. By equalizing path lengths, trace selection synchronizes control dependent paths at the re-convergent point — this is shown in Figure 4(c). This padding technique enables traces to expand or contract to recover from mispredictions, without affecting the boundaries of subsequent traces, similar to *tasks* in multiscalar processors [2]. Section 3 describes how forward-branching regions are detected and analyzed, and how trace selection uses the resultant information to expose FGCI.

All other branches are covered by CGCI. In the extreme case, trace selection could search for the precise, i.e. nearest, control independent points for all branches, and then use these points to delineate traces. However, experience with this approach has yielded negative results. There are so many re-convergent points that synchronizing at every one of them creates a large number of small traces, worsening PE utilization, trace cache performance, and trace predictor performance.

Instead, trace selection can exploit a limited number of easily identified, "global" control independent points in the dynamic instruction stream. Loop back-edges, loop exits, and subroutine return points are all examples of global re-convergent points [5,11,3]. To ensure trace-level re-convergence for CGCI branches, traces are delineated at chosen global re-convergent points — as shown in Figure 4(d). Then, if a branch is mispredicted, an exposed global re-convergent point nearest the branch is found in the window and assumed to be the first control independent trace. Section 4 describes CGCI trace selection.

1.1.3. Managing data dependences. A mispredicted branch instruction causes not only incorrect control flow, but potentially incorrect data flow as well. After repairing the control flow, control independent instructions must be inspected for incorrect data dependences both through registers and memory, and any incorrect-data dependent instructions selectively reissued.

A primary feature of trace processors is the pervasive use of data speculation — value prediction [e.g., 12], load/store address prediction [e.g., 13], and speculative memory disambiguation [e.g., 14]. Selective misspeculation recovery, therefore, was anticipated as an important problem and played a central role in the trace processor microarchitecture [5]. The selective recovery model as it applies to control independence is reviewed in Section 2.2.

1.2. Related work

Lam and Wilson's limit study [1] demonstrates that control independence exposes a large amount of instruction-level parallelism, on the order of 10 to 100, for control-intensive integer benchmarks. This work was followed up by Uht and Sindagi [15] in their study of "minimal control dependences" and showed similar results.

A recent study [3] examines the potential of control independence in the context of wide-issue superscalar processors. An aggressive implementation achieves improvements on the order of 30%. The proposed mechanisms are complex due to the non-hierarchical superscalar organization, and there is a reliance on the compiler to provide complete control dependence information. Nonetheless, the study is useful for understanding control independence.

Multiscalar processors [16,2], Dynamic Multithreading [11], and other multithreaded architectures [17, 18, 19, 20] exploit control independence by pursuing multiple flows of control. Either the compiler or hardware partitions the program into tasks/threads, or subgraphs of the CFG, which may contain arbitrary control flow. Branch mispredictions within a task/thread may not cause subsequent tasks to squash if they are control independent of the branch.

The instruction reuse buffer [21] 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 misprediction. However, control independent instructions after the squash can be quickly re-evaluated via the reuse buffer.

Another approach using a dual reorder buffer design is presented in [22]. A misprediction squashes one of the reorder buffers, but control and data independent instructions from the second reorder buffer are preserved.

Dynamic Hammock Predication (DP) [23] is loosely related to the FGCI concepts developed in this paper, only in that both techniques exploit `if-then-else` constructs. There are clear distinctions: 1) FGCI places full trust in branch prediction and reduces penalties when mispredictions do occur, whereas DP eagerly executes multiple control dependent paths in anticipation of mispredictions. 2) FGCI implements *dynamic detection* of control flow constructs, whereas DP still requires the compiler to identify and mark `if-then-else` regions for predication (FGCI could make DP fully dynamic). 3) Our FGCI algorithm dynamically analyzes arbitrarily complex, nested forward-branching code, whereas DP is restricted to regions containing only a single conditional branch.

The SIMP architecture [33] fetches multiple control dependent paths after a branch. Special hardware 1) detects re-convergence and 2) establishes all *potential* data dependences between control independent instructions and

instructions before the re-convergent point. The result is an interesting predication/control independence hybrid: instructions are selectively discarded but not inserted (*similar to predication approaches*); data dependent/control independent instructions may re-issue several times due to having multiple, alternative data dependences (*similar to control independence approaches*).

The advantages of trace processors in terms of complexity and performance are discussed in [4,5]. Control independence in trace processors is briefly introduced in [5] but, because it is not the focus of that paper, the major issues are not formalized, conveyed, nor fully understood. The problem is not formalized in terms of FGCI and CGCI control flow management, and trace-level reconvergence and its implications to trace selection are not discussed. Performance gains are observed in only two of the benchmarks and these gains are due to manually-inserted, FGCI-like trace selection hints conveyed in the benchmark binaries; PEs are managed in a fifo queue so CGCI is not explicitly exploited.

Basic trace selection studies for trace caches and trace processors can be found in [8,9,10,5,24], and for compilers in [25,26]. Task selection studies can be found in [27,28].

1.3. Paper organization

Section 2 describes the trace processor's novel window management, i.e. support for instruction insertion/removal from the middle of the window (both control flow and data flow aspects). This is followed by trace selection for ensuring trace-level re-convergence, for both FGCI (Section 3) and CGCI (Section 4). Our experimental setup is described in Section 5, and Section 6 provides performance analysis.

2. Trace processor window management

In Section 1.1.1, we highlighted the two ways in which trace processors flexibly insert and remove instructions from the middle of the instruction window: FGCI and CGCI. The following two sections provide details regarding control flow and data flow management, respectively.

2.1. Managing control flow

Sophisticated control flow management is performed by the trace processor frontend, shown in Figure 5, since it controls PE allocation (trace dispatch) and deallocation (trace squash).

The trace predictor and trace cache together provide trace-level sequencing. Ideally, during trace-level sequencing the next trace is predicted correctly, and it hits in the trace cache. The trace is passed to the dispatch stage where live-in and live-out registers are renamed, establishing the register dependences with previous traces in the processor. The renamed trace is allocated to a PE via the dispatch bus.

Trace-level sequencing does not always provide the required traces to the PEs. Instruction-level sequencing is required to construct non-existent traces or repair trace mispredictions. The trace construction hardware consists of multiple *outstanding trace buffers*, one per PE. On a trace cache miss, the respective trace buffer is notified to construct a new trace. It uses the trace prediction (starting PC and branch outcomes) to fetch instructions from the instruction cache. Meanwhile, the trace dispatch pipe is stalled — no other traces may pass through renaming because of the missing trace. However, the fetch stage is free to continue predicting traces, and these traces are placed in their outstanding trace buffers despite not reaching the dispatch stage. When the missing trace has been constructed and pre-renamed, the dispatch pipe is restarted and supplied with traces from the buffers in sequential order. The non-blocking fetch pipe allows multiple trace cache misses to be serviced in parallel, restricted only by the number of datapaths to/from the instruction cache and branch predictor.

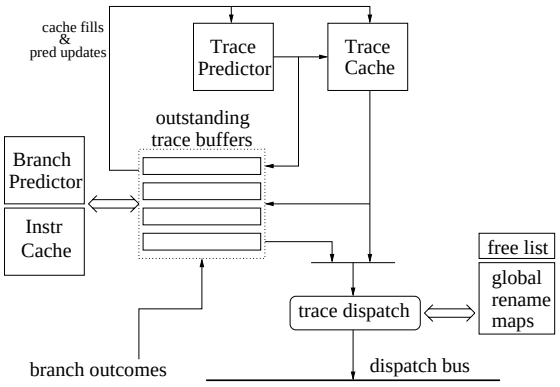


FIGURE 5. Frontend of the trace processor.

On a trace cache hit, the trace is not only dispatched to the PE but also placed in the corresponding trace buffer. The trace buffer monitors branch outcomes as they become available from the PE. If a misprediction is detected, the trace predictor is backed up to that trace, as are the global register rename maps. Also, the trace buffer begins repairing the trace from the point of the branch misprediction, using the simple branch predictor to fetch instructions. However, subsequent PEs and their traces are not affected at this point, i.e. they continue processing instructions. When the mispredicted trace has been repaired, the trace-level sequencer is restarted in one of two ways, depending on whether the branch is covered by fine- or coarse-grain control independence.

1. Fine-grain: Control flow recovery for FGCI is very simple because the PE arrangement is unaffected. The frontend merely dispatches the repaired trace from its trace buffer to the affected PE. Within the affected PE, only instructions after the mispredicted branch are squashed.

At this point, the register rename map reflects correct register dependences up to and including the repaired trace; a *trace re-dispatch sequence*, described in the next section, makes a pass through the control independent traces to update their register dependences.

2. Coarse-grain: First, the sequencing hardware locates an exposed global re-convergent point in the window — the one *after* and generally *nearest* the mispredicted trace. CGCI trace selection (Section 4) detects and chooses certain global re-convergent points at which to terminate traces; these points are always "exposed" as trace boundaries and, therefore, visible to the sequencing hardware. Note that an exposed global re-convergent point may not exist in the window, in which case CGCI is not exploited. Even if one is located, it may or may not actually be control independent with respect to the mispredicted branch.

   Next, the traces between the mispredicted branch and the first (assumed) control independent trace are squashed and their PEs deallocated. The trace predictor fetches the correct control dependent traces and they are allocated to newly freed PEs. Squashing and allocating PEs proceed in parallel, just as dispatch and retirement proceed in parallel. If there are more correct control dependent traces than incorrect ones, then PEs must be reclaimed from the tail (i.e. the most speculative PE).

   Finally, the control flow is successfully repaired when re-convergence is detected, i.e. *when the next trace prediction matches the first control independent trace*. Although re-convergence is not guaranteed, if and when it occurs, a trace re-dispatch sequence is performed on the control independent traces to update their register dependences, as described in the next section.

With CGCI, the logical or program order of PEs can no longer be inferred from just the head/tail pointers and the *physical* order of PEs. Logically inserting and removing PEs between two arbitrary PEs, i.e. inserting and removing control dependent traces, requires managing the PEs as a linked-list. The linked-list control structure is simply a small table indexed by physical PE number, with each entry containing three fields: logical PE number (order in the list) and pointers to the previous and next PEs. Also, head PE and tail PE pointers are needed as before. The control structure (table plus head/tail pointers) is consulted and possibly updated by the trace-level sequencer when dispatching, retiring, squashing, and re-dispatching traces.

2.2. Managing data flow

After the sequencing hardware repairs control flow, incorrect-data dependent, control independent instructions must be identified and selectively reissued. The first step is to detect the "source" of a data dependence violation and reissue that instruction. There are two possible sources: 1) a stale physical register name representing an incorrect register dependence, or 2) a load instruction that loaded an incorrect version of a memory location. The second step is to selectively reissue all subsequent data dependent instructions.

2.2.1. Stale physical register names. The frontend initiates a *trace re-dispatch sequence* after control flow is repaired. Control independent traces are re-dispatched in sequential (program) order from the trace buffers to respective PEs. Live-in registers are renamed using the updated maps, and live-out registers do not change their mappings. The source register names of each instruction in the PE-resident trace are checked against those in the re-dispatched trace. Only those instructions with updated register names are reissued.

2.2.2. Incorrect loads. For memory dependences, we leverage the existing mechanism for detecting incorrectly disambiguated loads [5]. Speculative memory disambiguation is performed in that 1) loads issue as soon as their addresses are available, irrespective of prior stores in the window, and 2) load and store addresses may be predicted or based on speculative values.

All loads and stores are assigned sequence numbers that indicate their program order within the window. Sequence numbers are derived from the PE number plus location in the PE's trace. A variant of the *address resolution buffer* (ARB) [14] resides before the data cache to maintain a list of speculative versions per address location; i.e. speculative store data is buffered and arranged in program order via sequence numbers. When a store is performed, it sends its address, data, and sequence number on one of the cache ports. When a load is performed, it queries the ARB with its address and sequence number. The ARB returns the correct version of data for that load *and* the sequence number of the store that produced the data. Thus, loads maintain two sequence numbers: its own and that of the load data.

Detection of memory dependence violations is based on loads *snooping* store addresses and sequence numbers on the cache ports. A load must reissue if 1) the store address matches the load address, 2) the store sequence number is logically less than that of the load (i.e. the store is before the load in program order), and 3) the store sequence number is logically greater than that of the load data (i.e. the load has an incorrect, older version of the data).

If a store has issued to the ARB using an incorrect address, it must issue again to the correct address. In the same transaction, it also sends the incorrect address and an indication to perform a "store undo" operation for that address. Loads snoop the store undo, and reissue if the

sequence number of the store undo matches that of their data.

This same mechanism works for control independent loads that are incorrectly disambiguated due to mispredicted branches. When a store is removed from the window, i.e. if it is among the incorrect control dependent instructions, it schedules a store undo operation (only if it has performed already). A store on the correct control dependent path is brought into the window late, and may appear as a normal disambiguation violation when control independent loads observe the late-performed store.

Sequence number comparisons first requires translating physical to logical sequence numbers. In [5], because the PEs are organized in a physical ring, the mapping is fairly direct. Now, due to the arbitrary arrangement of PEs, translation requires consulting the linked-list control structure (each PE maintains a copy). Recall that the linked-list table maintains physical to logical PE translations: this field exists solely for disambiguation support.

2.2.3. Selectively reissuing dependence chains. After detecting the "source" of a data dependence violation (stale register name or incorrect load), the violating instruction is reissued. The second step is to selectively reissue all subsequent data dependent instructions. This happens transparently in the trace processor because instructions remain in the PEs until they retire. Therefore, if an instruction has already issued and it receives one or more additional values, it simply reissues as many times as is necessary.

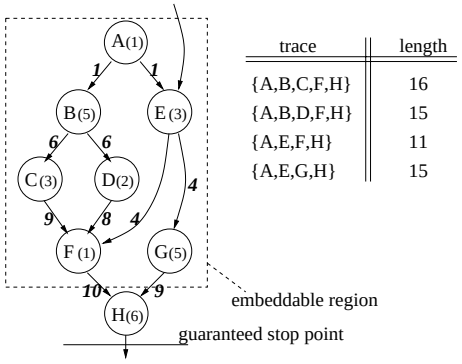
3. Trace selection for FGCI

An example of FGCI trace selection is shown in Figure 6. Basic blocks are labeled with a letter *A* through *H*, and block sizes are shown in parentheses. Control flow edges are labeled with the longest path length leading to that edge. The maximum trace length is 16 in the example.

The branch in block A is a candidate for FGCI because the maximum length of any of its control dependent paths is 10 instructions, well within the maximum trace length. The region enclosed in the dashed box (the branch in block A and its control dependent instructions) is called the *embeddable region*, and the *dynamic region size* of this region is its maximum path length, 10.

During trace construction, if a branch with an embeddable region is encountered, the accrued trace length is incremented by the branch's dynamic region size, *irrespective of which control dependent path is actually selected*. The result will be one of four traces as shown in the table of Figure 6. First, not all traces are 16 instructions long, only the trace which actually embeds the longest control dependent path. Second, all traces end at the same instruction, namely the last instruction in basic block H. This of course achieves the desired effect: if the trace predictor predicts one trace and it turns out to be incorrect, it can be replaced with one of the alternate traces without changing the sequence of subsequent control independent traces. Third, any of three branch mispredictions is covered by this region — the branches in basic blocks *A*, *B*, and *E*.

Exposing FGCI first requires finding branches with embeddable regions (Section 3.1). A *FGCI-algorithm* is applied to each newly-encountered branch to check if it has an embeddable region. If it does, the goal of the algorithm is to determine 1) the re-convergent PC that closes the region and 2) the dynamic region size. The latter amounts to computing the longest path through a topologically sorted DAG [29] in hardware. This gathered information is cached so that it does not have to be re-computed each time a branch is encountered. Then, trace selection can use the cached information to conceptually pad traces (Section 3.2).



trace	length
{A,B,C,F,H}	16
{A,B,D,F,H}	15
{A,E,F,H}	11
{A,E,G,H}	15

FIGURE 6. Example of an embeddable region.

3.1. FGCI-algorithm

For brevity, a detailed description of the FGCI-algorithm is deferred to the corresponding thesis [32].

The underlying idea is to serially scan a static block of instructions following a forward conditional branch. Only a single pass is required. Conceptually, each instruction is modeled as a node having one or more incoming control flow edges, each edge having a value equal to the maximum path length leading to the edge. The algorithm assigns a value to the node equal to the maximum value of incoming edges plus one for the current instruction. In this way, the longest control dependent path lengths are propagated from incoming edges to outgoing edges.

The key to the algorithm, therefore, is determining all incoming edges to an instruction. The implicit edge between two sequential instructions is readily apparent. The edges between branches and their targets require explicit storage. When a branch instruction is scanned, its taken target PC is recorded along with the path length up to the branch. Each scanned instruction checks its PC against the list of accumulated branch targets: a hit indicates an incoming edge and its path length is available.

During scanning, the most distant taken target among forward branches is maintained. The *re-convergent point* is detected when scanning reaches this target. The *dynamic region size* is equal to the maximum path length propagated to the re-convergent instruction. The branch forming the region is not a candidate for FGCI, however, if any computed path length exceeds the maximum trace length before re-convergence, or if a backward branch, function call, or indirect branch is encountered before re-convergence.

The FGCI-algorithm has several characteristics amenable to hardware implementation [32]. First, it performs a single pass which yields a simple controller and simple state maintenance (3 words of state plus a 4- to 8- entry associative array for edges). Second, limiting the scan rate to 1 instruction/cycle helps manage complexity as well as reduce bandwidth to the instruction cache. The design is relatively non-intrusive to the cache port — for the CFG in Figure 6, 2 or 3 cache line fetches are initiated over 21 cycles, for a line size of 16 words.

The information computed by the FGCI-algorithm is written into a cache called the *branch information table* (BIT). All forward conditional branches allocate entries in the BIT, whether they have an embeddable region or not, because trace selection needs to know this determination. For a 16K-entry BIT and a trace length of 32 instructions, a BIT entry is 4 bytes long: a tag (16 bits), a flag indicating embeddable or not (1 bit), the dynamic region size (5 bits), and the re-convergent point in the form of an offset from the start of the region (10 bits is reasonable).

3.2. FGCI trace selection

When a forward conditional branch is encountered during trace selection, the branch PC is used to access FGCI information from the BIT. If the information does not exist, a BIT miss handler initiates the FGCI-algorithm and trace construction stalls until the handler completes.

If the BIT indicates the branch is a candidate for FGCI, and the current trace length plus the dynamic region size of the branch does not exceed the maximum trace length constraint, then the following actions are performed.

1. The cumulative trace length is incremented by the dynamic region size.
2. Incrementing of the cumulative trace length is halted while a given path through the embeddable region (as dictated by branch prediction) is added to the trace.
3. Incrementing the cumulative trace length resumes when the re-convergent point (from the BIT) is reached.

Managing the cumulative trace length in this way guarantees paths shorter than the longest path through the embeddable region are effectively “padded” to the longest path length.

If the current trace length plus the dynamic region size of the branch exceeds the maximum trace length constraint, then the current trace is terminated before the branch. Deferring the branch to the next trace ensures all potential FGCI is exposed.

4. Trace selection and heuristics for CGCI

Trace selection and the trace processor frontend coordinate to exploit CGCI. Trace selection delineates traces at key global re-convergent points. When a misprediction is detected, the frontend chooses one of the points (based on heuristics), if there are any in the window, to serve as the trace-level re-convergent point for recovery.

4.1. CGCI trace selection

In this paper, we consider only two types of global re-convergent points. These are the targets of return instructions and the not-taken targets of backward branches, shown with black dots in Figure 7(a) and Figure 7(b), respectively.

The default trace selection algorithm terminates traces at the maximum trace length or at any indirect branch instruction; indirect branches include jump indirect, call indirect, and return instructions. Therefore, default trace selection already ensures trace-level re-convergence at the exits of functions, i.e. return targets.

An additional CGCI trace selection constraint, called *ntb*, terminates traces at predicted not-taken backward branches. This ensures trace-level re-convergence at the exits of loops.

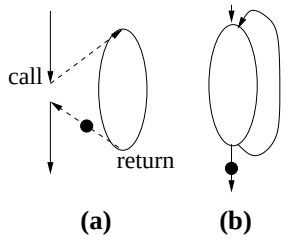


FIGURE 7. Global re-convergent points.

4.2. CGCI heuristics

Trace selection only ensures trace-level re-convergence at easily identified, global re-convergent points. When a branch misprediction is detected, one from possibly many such points in the window must be chosen as the trace-level re-convergent point used for recovery actions. Only two CGCI heuristics are considered.

- *RET*: The frontend locates the nearest trace that ends in a return instruction. The immediately subsequent trace is assumed to be the first control independent trace.

- *MLB-RET*: If the mispredicted branch is a backward branch, we assume it is a loop branch as depicted in Figure 7(b). Based on this assumption, the frontend locates the nearest trace whose starting PC matches the not-taken target of the branch; it is likely the correct re-convergent point. This heuristic, Mispredicted Loop Branch (*MLB*), is always considered first. However, if the mispredicted branch is not a backward branch, then the *RET* heuristic is applied.

The *RET* heuristic is designed to cover arbitrary mispredictions within a function since control flow re-converges at the function exit. However, due to nested functions, there may be any number of other return instructions before the intended function exit. Choosing the nearest return instruction, therefore, is only a guess. Often, the result is better than intended — when the chosen return is closer than the function exit, but still control independent with respect to the misprediction. Other times, however, the chosen return is on the incorrect control dependent path.

The *MLB* heuristic (of *MLB-RET*) is designed to specifically and accurately cover mispredicted loop branches, a substantial source of branch mispredictions (refer to Table 5). Loops with small bodies and a small but unpredictable number of iterations fall in this category, and there are often many control independent traces after the mispredicted loop branch.

The *RET* heuristic requires only default trace selection, whereas *MLB-RET* requires, in addition, the *ntb* selection constraint to expose loop exits.

5. Experimental setup

A detailed, fully-execution driven trace processor simulator is used to evaluate control independence. The simulator was developed using the *simplescalar* toolset [30].

Table 1 summarizes the major parameters in the trace processor configuration; a comprehensive description can be found in [32]. Several of the parameters are worth discussing. First, a very large trace predictor is used to achieve high overall branch prediction accuracies (which are reported in the next section). Also, the frontend and execution pipelines are not heavily-pipelined. Together the accurate trace predictor and small branch misprediction penalty potentially skew results in the conservative direction (i.e. less benefit from control independence). However, also note that 16 PEs are simulated in anticipation of future large instruction windows, and for which control independence techniques are likely to be more relevant. The maximum trace length (and hence PE window size) is 32 instructions.

Five of the SPEC95 integer benchmarks, shown in Table 2, were simulated to completion.

TABLE 1. Trace processor configuration.

frontend latency	2 cycles (fetch + dispatch)
trace predictor (hybrid)	2^{16}-entry path-based pred.: 8 traces hist.
	2^{16}-entry simple pred.: 1 trace hist.
trace cache	size/assoc/repl = 128kB/4-way/LRU
	trace line size = 32 instructions
instruction cache	2-way interleaved, 1 basic block/cycle
	size/assoc/repl = 64kB/4-way/LRU
	line size = 16 instructions
	miss penalty = 12 cycles
branch predictor	16K-entry tagless BTB, 2-bit counters
BIT	8K-entry, 4-way assoc.
trace construction b/w	1 port to instr. cache, branch pred., BIT
processing elements	16 PEs, 4-way issue per PE
global result buses	8 buses, up to 4 can be used by 1 PE
	extra 1-cycle result bypass latency
cache buses	8 buses, up to 4 can be used by 1 PE
data cache	size/assoc/repl = 64kB/4-way/LRU
	line size = 64 bytes
	miss penalty = 14 cycles
execution latencies	address generation = 1 cycle
	memory access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies
	load re-issue penalty = 1 cycle (snoop lat.)

TABLE 2. Benchmarks.

benchmark	input dataset	# dynamic instr.
compress	400000 e 2231	104 million
gcc	-O3 genrecog.i	117 million
go	9 9	133 million
ijpeg	vigo.ppm	166 million
xlisp	queens 7	202 million

6. Results

Two sets of experiments are presented. The first set focuses on the impact of FGCI and CGCI *trace selection* in a trace processor without control independence mechanisms. This is required to isolate the effects of trace selection on trace cache performance, trace predictor performance, and PE utilization. The second set evaluates the performance of *control independence*.

6.1. Performance impact of trace selection

Default trace selection terminates traces at a maximum length of 32 instructions or at any jump indirect, call indirect, or return instruction. The *ntb* trace selection terminates traces at predicted not-taken backward branches, and *fg* denotes FGCI trace selection. The selection-only experiments are prefixed with *base* to indicate no control independence, and are followed by the trace selection

algorithm(s) used. Default trace selection is always in effect and, therefore, is not explicitly specified. The four experiments are: *base*, *base(ntb)*, *base(fg)*, and *base(fg,ntb)*.

Performance results in instructions per cycle (IPC) are tabulated in Table 3. Also, the performance improvement with respect to *base* is graphed in Figure 8. Additional selection constraints (*fg*, *ntb*) tend to affect basic performance adversely. To help understand why, Table 4 shows the impact of selection on trace length, trace mispredictions, and trace cache misses (the latter two are given as misses per 1000 instructions and as a rate).

TABLE 3. IPC without control independence.

	base	base(ntb)	base(fg)	base(fg,ntb)
gcc	4.44	4.51	4.34	4.36
go	3.17	3.20	3.07	3.10
comp	2.02	1.92	1.96	1.92
ijpeg	7.12	7.24	6.96	6.96
xlisp	4.72	4.31	4.72	4.34

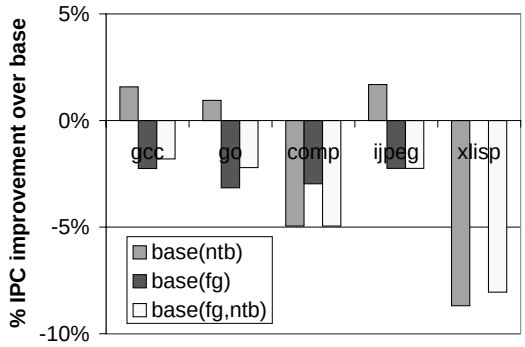


FIGURE 8. Performance impact of trace selection.

Additional selection constraints always decreases average trace length, and from Table 4, this almost always increases trace mispredictions per 1000 instructions. The trace predictor uses a path history of traces, and reducing the lengths of traces effectively reduces the amount of implicit branch history. Also, *synchronizing* trace selection among many disjoint paths — in nested hammocks (*fg*) or after exiting a loop (*ntb*) — reduces the number of unique traces significantly. Yet it is this uniqueness that provides a very distinct context for making predictions [31].

Reducing the average trace length also results in a waste of issue buffers in the PEs, effectively making the instruction window smaller. The only positive effect is on trace cache performance, but the benefit is generally overshadowed by costlier trace mispredictions.

The *base(ntb)* model improves performance slightly for three of the five benchmarks, but for *compress* and *xlisp*, performance degrades by 5% and 10%, respectively. The effect is pronounced in *xlisp* because trace length drops by 25%, double what other benchmarks experience. The *base(fg)* model degrades performance between 2% and 3% for four of five benchmarks.

6.2. Performance of control independence

In this section, we evaluate the performance of four control independence models:

- *RET*: coarse-grain only, using the *RET* heuristic.
- *MLB-RET*: coarse-grain only, using the *MLB-RET* heuristic.
- *FG*: fine-grain only.
- *FG + MLB-RET*: fine-grain and coarse-grain using the *MLB-RET* heuristic.

Figure 9 (performance improvement over *base*) shows that control independence improves performance substantially. Coarse-grain control independence performs uniformly well across the benchmarks, with the exception of *ijpeg*. The *RET* model improves performance by about 5% for *gcc*, nearly 10% for *xlisp*, and about 20% for *compress* and *go*. Going from *RET* to *MLB-RET* improves performance moderately for *gcc* and *go*, due to greater misprediction coverage and establishing more precise control independent points for backward branches. The improvement is perhaps moderate due to overlapping coverage between the two kinds of global re-convergent points. For *xlisp*, *MLB-RET* drops performance with respect to *RET*, an artifact of *ntb* trace selection.

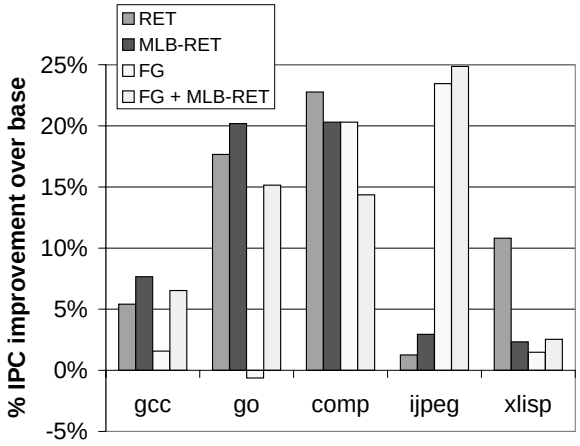


FIGURE 9. Performance of control independence.

To give insight into the performance of FGCI, conditional branch statistics are shown in Table 5. Branches are classified into those that can be captured by FGCI, all other forward branches, and backward branches. FGCI branches are further divided into those whose regions fit (≤ 32) or do not fit (>32) in a trace; a trace length of 32 can capture almost all FGCI-type branches. The fraction of dynamic branches and mispredictions are given for each class.

TABLE 4. Impact of trace selection on trace length, mispredictions, and misses.

		gcc	go	compress	ijpeg	xlisp
base	avg. trace length	24.0	27.2	24.9	31.1	19.7
	trace misp. rate	4.2 (10.1%)	7.3 (19.9%)	10.6 (26.3%)	3.1 (9.5%)	4.8 (9.4%)
	trace $ miss rate	4.7 (11.2%)	10.2 (27.7%)	0.0 (0.0%)	0.3 (1.1%)	0.0 (0.0%)
base(ntb)	avg. trace length	21.6	24.4	21.6	30.1	14.7
	trace misp. rate	4.3 (9.3%)	7.4 (18.1%)	11.2 (24.2%)	3.0 (9.0%)	6.0 (8.8%)
	trace $ miss rate	4.1 (8.8%)	9.7 (23.7%)	0.0 (0.0%)	0.3 (0.9%)	0.0 (0.0%)
base(fg)	avg. trace length	21.8	23.9	24.6	28.9	18.9
	trace misp. rate	4.4 (9.7%)	8.1 (19.2%)	10.8 (26.5%)	3.8 (11.0%)	4.9 (9.2%)
	trace $ miss rate	4.0 (8.8%)	9.4 (22.4%)	0.0 (0.0%)	0.2 (0.7%)	0.0 (0.0%)
base(fg,ntb)	avg. trace length	19.7	21.6	21.2	28.1	14.2
	trace misp. rate	4.7 (9.2%)	8.3 (17.9%)	10.9 (23.2%)	3.9 (10.8%)	6.0 (8.6%)
	trace $ miss rate	3.6 (7.2%)	9.0 (19.4%)	0.0 (0.0%)	0.2 (0.7%)	0.0 (0.0%)

FGCI branches account for 23% to 41% of all branches, and over 60% of all mispredictions, in *compress* and *ijpeg*. This explains why the model *FG* performs very well on these benchmarks, namely a 20% to 25% performance improvement.

FGCI branches account for 24% of all mispredictions in *go*, yet *FG* actually degrades performance by less than 1%. Looking further into the misprediction behavior of *go*, we have noticed that FGCI has large potential in some frequently executed code (e.g. the "addlist" function), but that neighboring mispredictions not covered by FGCI nullify this potential; combined with the minor adverse effects of *fg* trace selection, the result is no gain. In contrast, the *MLB-RET* model performs well by capturing *clusters* of mispredictions in these same code regions.

Returning to Table 5, *gcc*, *go*, *ijpeg*, and *xlisp* have large FGCI regions (13 to 40 instructions) with many conditional branches enclosed (3 to 4). The average dynamic region size of FGCI branches is from 1 to 8 instructions smaller than the corresponding static code region.

TABLE 5. Conditional branch statistics.

			gcc	go	comp	ijpeg	xlisp
FGCI branches	≤ 32	frac. br.	21.4%	24.5%	40.8%	22.5%	10.0%
		frac. misp.	20.3%	24.4%	63.1%	60.6%	3.0%
	> 32	frac. br.	1.9%	2.6%	0.1%	2.0%	0.0%
		frac. misp.	1.3%	2.7%	0.0%	1.9%	0.0%
	misp. rate		2.8%	8.7%	14.6%	14.8%	1.0%
	dyn. region size		11.3	13.8	4.3	31.9	13.2
	stat. region size		12.9	16.4	5.7	40.2	16.3
	# cond. br. in reg.		3.2	2.6	1.6	4.3	3.8
other forward branches		frac. br.	58.3%	52.8%	23.6%	24.8%	63.2%
		frac. misp.	55.8%	51.8%	17.8%	15.8%	36.1%
		misp. rate	2.9%	8.5%	7.1%	3.7%	1.9%
backward branches		frac. br.	18.4%	20.1%	35.5%	50.7%	26.7%
		frac. misp.	22.6%	21.1%	19.1%	21.7%	60.9%
		misp. rate	3.8%	9.1%	5.1%	2.5%	7.4%
overall branch misp. rate			3.1%	8.7%	9.4%	5.8%	3.3%
branch misp./1000 instr.			4.7	10.4	13.5	3.8	5.1

Backward branches account for a large fraction of mispredictions, 20% for four of the benchmarks and 60% for *xlisp*. Unfortunately for *xlisp*, applying *ntb* trace selection — so the *MLB-RET* heuristic can cover these mispredictions — also worsens the prediction accuracy of the backward branches. While *MLB-RET* performs only slightly better than *base*, it improves performance by 10% over *base(ntb)* — i.e. CGCI is being exploited, if only to break even with *base*.

In summary, using FGCI and CGCI techniques together achieves the best performance improvement on average: 13% (*FG + MLB-RET*). Clearly, some techniques work better than others depending on the benchmark, perhaps suggesting the need for adaptive trace selection. Using the best-performing technique for each benchmark, control independence achieves an average improvement of 17%.

7. Summary

Control independence is a promising technique for overcoming the branch misprediction bottleneck. Trace processors exploit hierarchy to manage the complexity of implementing control independence, while maintaining the performance advantages of a contiguous instruction window and a relatively accurate single flow-of-control.

The ideas presented in this paper can be summarized as follows.

- A primary source of control independence complexity is the insertion and removal of instructions at arbitrary points in the window. Fortunately, the hierarchical instruction window of trace processors accommodates flexible control flow management. In the case of fine-grain control independence (FGCI), control flow recovery is localized within a single PE. In the case of coarse-grain control independence (CGCI), control flow recovery involves multiple PEs, but treating traces as the fundamental unit of control flow results in efficient recovery actions.

- Traces facilitate flexible control flow management but introduce a new problem: trace-level re-convergence is not guaranteed despite re-convergence at the instruction-level. Novel FGCI and CGCI trace selection techniques were developed for ensuring trace-level re-convergence.
- Trace processors exploit a variety of data speculation techniques and, therefore, already incorporate high-performance, selective data recovery mechanisms. These mechanisms are easily leveraged to selectively re-execute incorrect-data dependent, control independent instructions.

Control independence improves trace processor performance from 5% to 25%, and 17% on average. Exploration of other, more sophisticated CGCI heuristics holds the potential for even larger performance gains.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853, the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346, an IBM Partnership Award, and SUN Microsystems.

References

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

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

[3] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. *5th Intl. Symp. on High Perf. Comp. Arch.*, Jan 1999.

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

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

[6] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. *30th Intl. Symp. on Microarch.*, Dec 1997.

[7] J. Johnson. Expansion caches for superscalar processors. Tech. Rep. CSL-TR-94-630, Stanford Univ., June 1994.

[8] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent 5,381,533, Jan 1995.

[9] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *29th Intl. Symp. on Microarch.*, Dec 1996.

[10] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Tech. Rep. CSE-TR-335-97, Univ. of Michigan, 1997.

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

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

[13] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. *29th Intl. Symp. on Microarch.*, Dec 1996.

[14] 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.

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

[16] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin, Nov 1993.

[17] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism in multiprocessors. Tech. Rep. CSL-TR-97-715, Stanford Univ., Feb 1997.

[18] 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.

[19] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. *PACT-95*, 1995.

[20] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. *PACT-96*, 1996.

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

[22] Y. Chou, J. Fung, and J. Shen. Reducing branch misprediction penalties via dynamic control independence detection. *Intl. Conf. on Supercomputing*, June 1999.

[23] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. *PACT-98*, Oct 1998.

[24] S. Patel, M. Evers, and Y. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *25th Intl. Symp. on Comp. Arch.*, June 1998.

[25] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[26] W. Hwu and P. Chang. Trace selection for compiling large c application programs to microcode. *21st Intl. Symp. on Microarch.*, Dec 1988.

[27] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin, 1998.

[28] T. N. Vijaykumar and G. Sohi. Task selection for a multiscalar processor. *31st Intl. Symp. on Microarch.*, Dec 1998.

[29] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, pages 536–538.

[30] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Tech. Rep. CS-TR-96-1308, Univ. of Wisconsin, July 1996.

[31] E. Rotenberg, S. Bennett, and J. Smith. A trace cache microarchitecture and evaluation. *IEEE Trans. on Computers*, 48(2):111–120, Feb 1999.

[32] E. Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. Ph.D. Thesis, University of Wisconsin, 1999.

[33] K. Murakami, N. Irie, M. Kuga, and S. Tomita. SIMP: A Novel High-Speed Single-Processor Architecture. *16th Intl. Symp. on Comp. Arch.*, May 1989.