Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences

Sriram Vajapeyam

Supercomputer Education and Research Centre
and
Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, INDIA 560012
sriram@csa.iisc.ernet.in

Tulika Mitra[1]

Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, INDIA 560012
tul@csa.iisc.ernet.in

Abstract

Superscalar processors currently have the potential to fetch multiple basic blocks per cycle by employing one of several recently proposed instruction fetch mechanisms. However, this increased fetch bandwidth cannot be exploited unless pipeline stages further downstream correspondingly improve. In particular, register renaming a large number of instructions per cycle is difficult. A large instruction window, needed to receive multiple basic blocks per cycle, will slow down dependence resolution and instruction issue. This paper addresses these and related issues by proposing (i) partitioning of the instruction window into multiple blocks, each holding a dynamic code sequence; (ii) logical partitioning of the register file into a global file and several local files, the latter holding registers local to a dynamic code sequence; (iii) the dynamic recording and reuse of register renaming information for registers local to a dynamic code sequence. Performance studies show these mechanisms improve performance over traditional superscalar processors by factors ranging from 1.5 to a little over 3 for the SPEC Integer programs. Next, it is observed that several of the loops in the benchmarks display vector-like behavior during execution, even if the static loop bodies are likely complex for compile-time vectorization. A dynamic loop vectorization mechanism that builds on top of the above mechanisms is briefly outlined. The mechanism vectorizes up to 60% of the dynamic instructions for some programs, albeit the average number of iterations per loop is quite small.

1. INTRODUCTION

A significant amount of recent research in superscalar processors has been directed at increasing the effective instruction fetch bandwidth[Rot96a, Con95a, Yeh93a, Hao96a]. Fetching a larger number of instructions per clock enables the processor to step through the program quicker and thus find more independent instructions for issue. However, increased fetch bandwidth can lead to the later pipeline stages of the processor

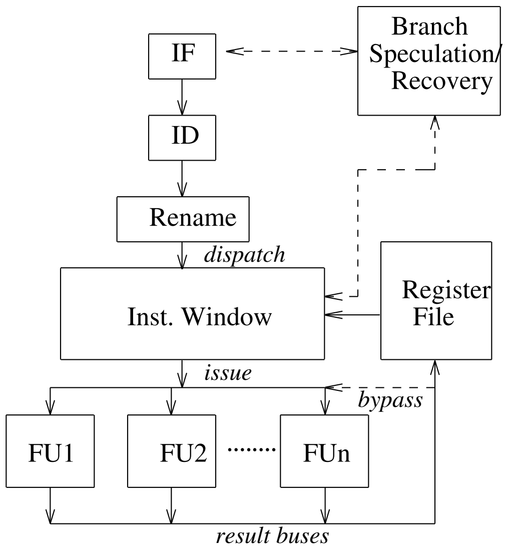


Figure 1. A Typical Superscalar Processor Pipeline

becoming bottlenecks. Typical pipeline stages in an aggressive superscalar processor capable of out-of-order instruction issue are depicted in Figure 1. After instructions are fetched and decoded, source and destination registers are renamed before instructions are dispatched to the instruction window, where they wait until their source operands are ready. Increasing the fetch bandwidth could result in the following bottlenecks in the subsequent pipeline stages:

(1) The register renaming bandwidth has to grow proportionally with the fetch bandwidth. The register renaming process has a sequential component[Wei95a] involving determining data dependencies amongst the instructions that are simultaneously being renamed. Even when partially parallel methods are used for dependence checking [Pal96a], the rename stage latency is limited by the number of read ports needed for the register map table,

PREPRINT: This paper is to appear in the *ACM SIGARCH 24th Annual International Symposium on Computer Architecture*, Denver, June 1997.

[1] Tulika Mitra's current address: Graduate Student, Dept. of Computer Science, SUNY at Stony Brook, NY 11794-4400. email : mitra@cs.sunysb.edu.

Benchmark	Fetch Bandwidth (Instructions/Cycle)							
	8		16		32		64	
	Win.Size (Avg.)	ILP	Win.Size (Avg.)	ILP	Win.Size (Avg.)	ILP	Win.Size (Avg.)	ILP
Bubsort	18	8.0	149	16.0	424	31.8	989	63.2
Eqntott (short input)	28	8.0	1095	15.6	5714	27.8	6496	51.2

Table 1. Impact of Fetch Bandwidth on Instruction Window Size and Performance. The study assumes 100% branch prediction accuracy, unlimited number of functional units, and unlimited maximum window size. The results are for short inputs, and indicate that large windows are needed even with relatively limited fetch bandwidths.

which is proportional to the number of operands being simultaneously renamed. These issues make renaming more than a few instructions per clock difficult.

(2) The instruction window could grow very large if the sustained instruction issue rate does not keep up with the instruction fetch/dispatch rate. Preliminary studies show (Table 1) that the window size grows large with fetch bandwidth even as performance increases, despite assumptions of unlimited functional units and unlimited instruction issue bandwidth. (Several previous studies[Aus92a, Lam92a, Wal91a] have directly or indirectly reported the performance improvements shown by large window sizes when *unlimited* fetch bandwidth is assumed.) This scenario is possible when sequential and highly parallel code segments are interspersed in the program trace. A larger window can increase the time needed to (i) feed a result/result tag to all waiting instructions in the window, and (ii) select and issue n instructions in a cycle from amongst all the ready instructions in the window.

(3) The physical register space needed to support a large instruction window could slow down register access time to more than one clock cycle. Each instruction in the instruction window needs a unique destination physical register, as physical registers are single-assignment registers. Thus a 256-entry instruction window is likely to use a 256-entry register file.

(4) A large instruction window needs a large number of functional units to exploit the exposed parallelism. The complexity of full bypassing of results grows quadratically with functional units since each functional unit output has to be provided data paths to the inputs of all other functional units.

A recent study [Pal96a] of the logic complexity of various superscalar pipeline stages identifies the instruction window logic for large windows as potentially the key limiting factor for processor clock speed. The next slowest stage is the register renaming stage when renaming multiple instructions per cycle. The register file access time for large files is also shown to be a problem. Full bypassing within a clock cycle is shown to be impossible for large windows.

We propose a set of mechanisms to address the above problems in superscalar processors. First, we partition the instruction window into blocks on instruction-fetch boundaries, allowing instruction selection logic and result bypassing to be simplified. Next, the register file is partitioned into a global register file and local register files, one per block, that hold registers that are strictly local to the instructions in the block. Third, we propose the recording of register renaming information in the trace cache[Rot96a], allowing rename requirements to be considerably reduced on a trace cache hit and thus increasing the effective rename bandwidth.

We next observe that several loops in ordinary programs exhibit vector-like behavior at runtime despite possibly having complex control paths in the static loop body. Complex static loop characteristics will preclude compile-time vectorization. We describe a *dynamic vectorization* mechanism that detects vectorizable behavior and alleviates the problems of large instruction windows by executing such loops in vector mode.

Performance studies show that the increased effective dispatch bandwidth and improved issue, bypass, and register access operations of the first set of proposed mechanisms provide significant performance improvements (ranging from 1.5 to 3 times) over a traditional superscalar processor model. Further, the proposed dynamic vectorization mechanism can capture as much as 60% of the dynamic instructions for some of the programs.

We describe the instruction window organization and related mechanisms in section 2. A performance study of the proposed mechanisms is presented in section 3. The dynamic vectorization scheme is described in section 4. We summarize and draw conclusions in section 5.

2. TRACE WINDOW

A large traditional instruction window can slow down the processor clock cycle, as discussed in the introduction. We propose the organization of the instruction window into *trace lines*, where each trace line is the logical equivalent of a line in the recently proposed trace cache [Rot96a]. Figure 2 depicts such an instruction window. A trace line consists of a small number of dynamically successive basic blocks in the program's execution, and is uniquely defined [Rot96a] by the address of the first instruction and outcomes for each of the branches in the dynamic sequence. Trace lookup in the trace cache consists of obtaining a match on both the address and the prediction bits. Multiple trace lines can originate at the same instruction address and can be present simultaneously in the trace cache. The trace window essentially partitions a traditional instruction window into blocks that can hold trace lines. A block is filled when a

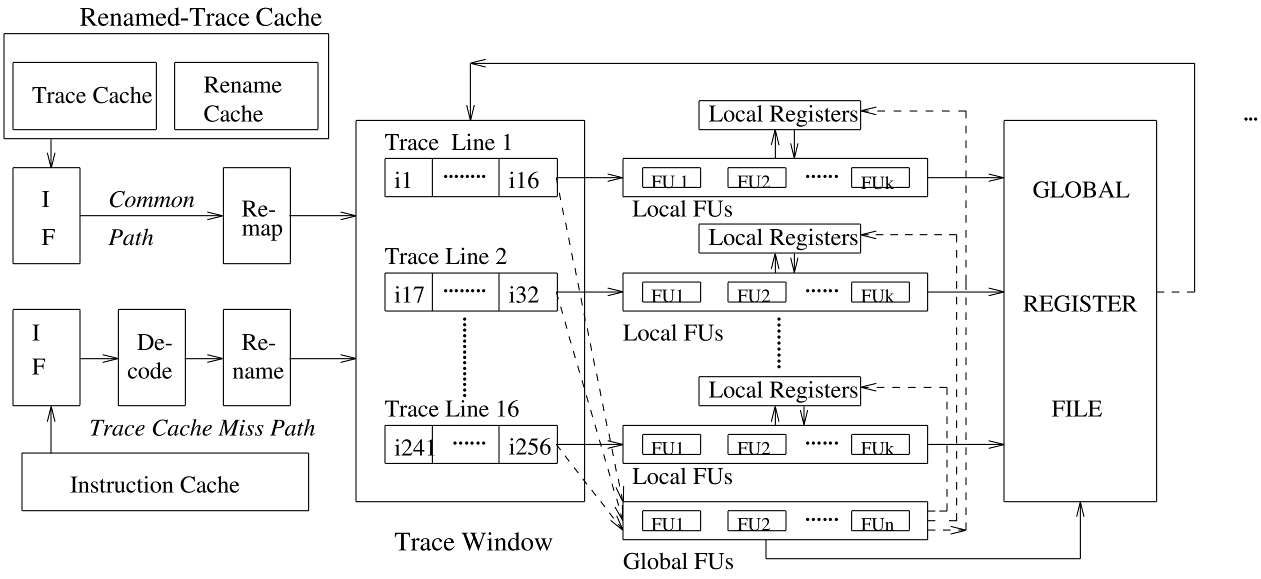


Figure 2. Instruction Window Organized into Trace Lines

trace line is dispatched to it, and freed when all instructions in the trace line have issued.

We describe further the organization of the trace window into trace lines in subsection 2.1. A mechanism for partitioning the large physical register file into a global file and several local files is described in subsection 2.2. Recording and reuse of register renaming is described in subsection 2.3. Subsection 2.4 describes a simple adaptation of the 2-level adaptive block-predictor[Fra95a] (proposed for use with restricted static program subgraphs having multiple targets) for use in conjunction with dynamic traces. We describe related work on instruction window organizations in subsection 2.5.

2.1. TRACE LINES

Each trace line fetched from the trace cache is decoded, renamed, and dispatched to a single trace line in the instruction window (termed trace window, Figure 2). Traditional window operations such as result forwarding and instruction selection are speeded up by this organization. The example configuration in Figure 2 shows each trace line with its own copy of each integer functional unit type, and costlier floating-point units shared across trace lines. Result bypassing is restricted to within the functional unit group associated with the instruction's trace line, and thus to dependent instructions from the same trace line. Previous work [Fra92a] shows that most instruction destinations are consumed within the next 32 dynamic instructions after being produced. This indicates that a trace line size of 32 instructions with restricted bypassing can capture a large fraction of the benefits of full bypassing without incurring the clock cycle penalty[Pal96a] of the latter.

Instructions that depend only on other instructions within the same trace line can have their dependencies quickly resolved due to the presence of local bypassing. Such intra-trace line dependence resolution can proceed in parallel and independently across the trace lines; it is also fast due to the small size of each trace line (i.e. a small subwindow). Dependencies across trace lines will be resolved relatively slowly, and will likely involve centralized logic.

Instruction issue bandwidth is restricted to a small number of instructions per cycle per trace line (e.g. 2 instructions/cycle per trace). This reduces the number of datapaths needed for simultaneously transmitting multiple instructions (i.e., opcodes and operands) to the functional units. Large peak issue rates can slow down window operations indirectly by increasing the window silicon area. The average ILP within a basic block is known to be around 2, and the ILP is not much higher across a few dynamically successive basic blocks. A trace line size of 16 instructions will typically hold about 3 dynamically successive basic blocks.

Further, it might be sufficient to issue instructions in-order from a trace line, assuming good compile-time instruction scheduling within basic blocks. This simplifies instruction selection logic by reducing the number of instructions that need to be examined each cycle to just the two at the head of the trace line. (The trace line then has to be treated as queue, with instructions at the head departing after issue and further instructions moving to the head.)

The trace window thus enables decentralisation of instruction issue operations (identifying ready instructions and arbitrating among them for issue). This avoids the bottleneck centralized issue logic would become [Pal96a] for large windows.

2.2. REGISTER FILE DECOMPOSITION

Each instruction in the instruction window, excepting branch and store instructions, needs a destination register, thus increasing the physical register size for large windows. In conjunction with the likelihood that bypassing has to be limited, register file access time can become a bottleneck. A solution to this problem lies in exploiting the fact that not all physical registers are live beyond the trace line that produce them. Table 2 shows the average number of live-on-exit registers and locally

Benchmark	Trace-Line Size (Instructions)									
	8		16		24		32		64	
	LoEx	Local	LoEx	Local	LoEx	Local	LoEx	Local	LoEx	Local
cc1	3.52	1.09	5.56	3.66	6.95	6.87	8.08	10.35	11.07	25.80
comp	3.55	1.06	5.66	3.56	7.05	6.79	7.97	10.47	10.09	26.79
eqntott	3.87	1.26	4.73	5.52	5.03	10.34	5.28	15.22	6.15	34.86
esp	4.22	1.79	5.77	6.24	6.64	11.37	7.24	16.78	8.01	40.04
xlisp	3.46	0.87	5.38	3.29	6.66	6.34	7.60	9.74	9.59	25.08

Table 2. The average number of Live-On-Exit and Local Destination Registers per trace line, for different trace line sizes. About half the registers are strictly local for the smaller trace lines (16 and 24 insts). The proportion of local registers increases for larger trace lines. The IBM RS6000's 1-bit condition-code registers are excluded in the above measurements; they have negligible impact[Mit97a]. Floating-point registers are rarely used by the benchmarks.

instruction i2 renamed.

```
i1: R1 <- R2 + R3
i2: R4 <- R4 * R1
i3: R1 <- R1 + R5
i4: Branch TARGET
```

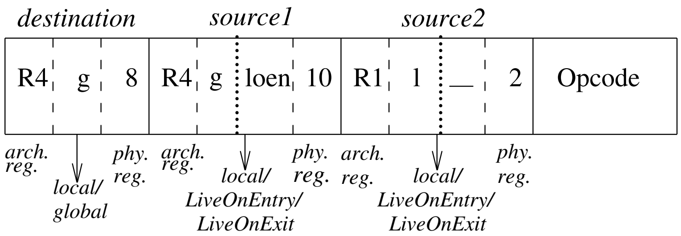


Figure 3. Renaming of Registers into Local and Global Physical Register Files. Register R4 is both live-on-entry to and live-on-exit from this block, and instruction i2 is appropriately renamed. Instruction i1's destination is strictly local to this code block.

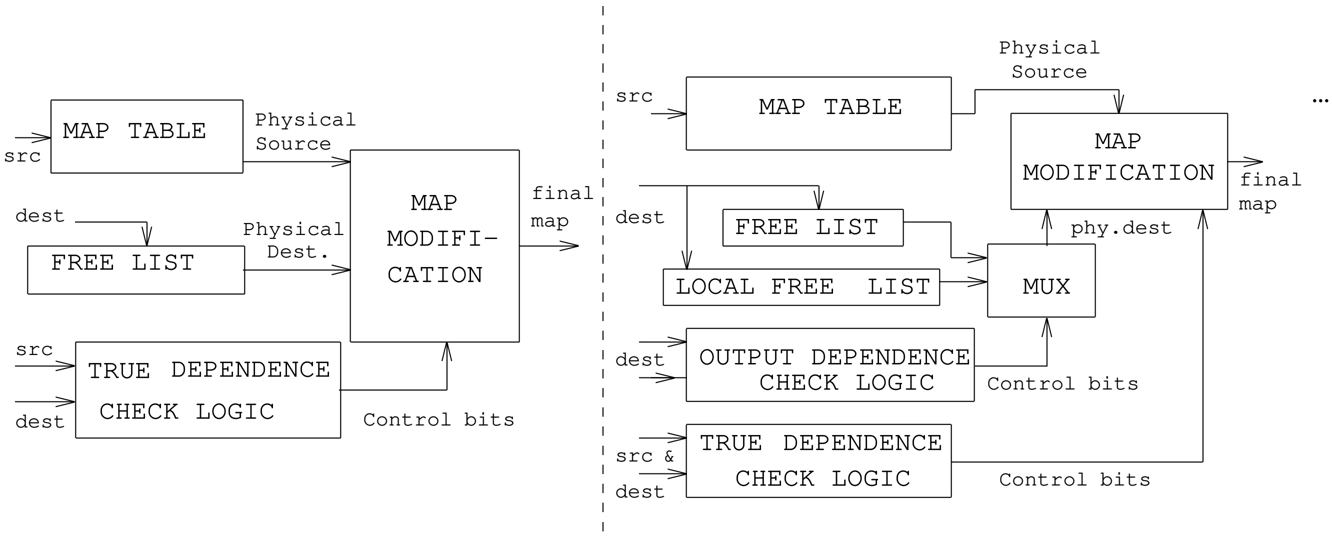
consumed registers for the SpecInt92 benchmarks for different choices of trace line size. Typically, around half the registers written by a small trace line (say of size 16 or 24 instructions) are not live beyond the trace line. The proportion of such local registers increases for larger trace lines.

Locally live destination registers can be renamed to a physical register file local to the trace line; live-on-exit destination registers are renamed to a global register file. A single bit tagged to each destination operand identifier in the renamed trace line indicates whether the specified physical register is from the local or the global register file. Source operands can be classified as live-on-entry (produced by another trace line), local (produced by the local trace line and available in the local register file), or live-on-exit (produced locally, but live-on-exit and hence available in the global register file). Live-on-entry source registers have to use old mappings of the global register file; live-on-exit source registers use new global register file mappings produced after renaming the destinations of their trace line. Two bits are tagged to each source operand to allow this classification. Figure 3 illustrates such renaming using a 4-instruction trace line. The local register file size needed is typically[2] no more than the number of instructions in the trace line.

[2] Some instructions have multiple destinations — for example, the LoadUpdate instruction in the IBM RS6000.

The local file is flushed when the corresponding trace line is removed from the window. The global register file size needed is proportional to the total number of live-on-exit registers in the entire window.

Traditional register renaming involves looking up a map table to pick up current mappings for architectural registers and, in parallel, true-dependence checking, followed by a modification of the mappings for source registers that were produced by earlier instructions in the current rename group (Figure 4a). To identify live-on-exit registers, output-dependence checking is done in parallel with the true-dependence checking (Figure 4b). In parallel, a unique local physical register is also tentatively picked up for each destination. (A simple scheme uses the instruction position in the trace as the destination register identifier for the instruction.) If the check logic indicates that an instruction's output is overwritten by a later instruction in the current rename group, the destination's local mapping is forwarded to the subsequent map modification stage instead of the mapping picked up from the global free list (the latter is then returned to the free list). This involves just an additional 2-to-1 mulitplexor latency per destination register, and is unlikely to affect the rename stage latency since the dependence check logic is faster than the map table lookup[Pal96a]. The tagging of register identifiers with the local/global bit and the live-on-entry/live-on-exit bit is achieved trivially by appropriately tagging the inputs to the 2-to-1 multiplexors.



(a) Traditional. (b) Trace Line.

Figure 4. Register Renaming.

Assuming register rename bandwidth is 4 instructions per cycle, a 16-instruction trace line would ordinarily be renamed and dispatched as four 4-instruction chunks, one chunk per cycle. However, in order to rename registers into local and global physical files, all the chunks are buffered at the rename stage's output until the last chunk is renamed, and the entire trace line is dispatched at the end of 4 clocks. This is necessary because a register that is live on exit from an earlier chunk of the trace line might be overwritten by a later chunk and thus not be live on exit from the trace line. Identification of the trace line's live-on-exit registers is pipelined, just as the renaming. All previously renamed chunks are output-dependence checked against the chunk currently being renamed, in parallel with intra-chunk dependence checking. A match results in the mapping for the previous chunk being changed in the map modification stage. Checking for inter-chunk dependences thus increases the dependence checking hardware in the rename stage but not the latency. When all four chunks are renamed, the buffer holding the trace line is flushed to the instruction window.

Register renaming of a trace line restricts dispatch bandwidth, and identification of live-on-exit registers makes instruction dispatch bursty. The next section proposes a mechanism for alleviating these problems considerably when trace lines are reused (i.e., when a control path is revisited).

2.3. RENAMED-TRACE CACHE

Register renaming bandwidth is unlikely to keep up with fetch bandwidth for superscalar processors dispatching more than a few instructions per cycle. We avoid full renaming on trace reuse by capturing in the trace cache renamed trace lines, obtained from the output of the register rename stage, rather than raw instructions. Figure 5 shows a typical entry in the trace cache. Additional bits are used with each trace line to record the live-on-entry and live-on-exit registers for the line. Each line also holds the instructions in decoded form, thus allowing the instruction decode stage to be skipped on a renamed-trace cache hit. Only the live-on-entry and live-on-exit registers for a reused line need to be remapped, allowing the line to go through a lower-bandwidth renaming stage in a single cycle. Each identifier in the "live-on-entry" tag of the trace line looks up the map table to pick up the current mapping. Each identifier in the "live-on-exit" tag of the trace line picks up a new physical register from the free list. Subsequently, operand specifiers that have the global bit set copy the new mappings in parallel (Figure 6), except for live-on-entry source operands which use old mappings. Operands marked as "local" use the previously recorded mappings. We observe that dependence checking logic is eliminated. A trace line appears to the rename stage as a single CISC instruction with multiple sources and multiple destinations, and only a mapping from architectural to physical registers is required.

The rename stage latency is determined by the number of operands looking up the map table and possibly by the number of operands querying the free list. The copying of the mappings into the instructions is similar to the map modification stage (Figure 4) of traditional renaming. Table 3 indicates that typically less than 6 map table lookups are involved per trace line of 16 instructions; table 2 indicates about 6 free list queries correspondingly. This demand is less than that for renaming 4

Benchmark	Trace-Line Size (Instructions)				
	8	16	24	32	64
cc1	3.79	5.19	5.99	6.55	7.83
comp	4.95	6.80	7.90	8.62	10.21
eqntott	3.43	4.10	4.38	4.65	5.54
esp	4.14	5.68	6.90	7.72	8.39
xlisp	3.65	4.81	5.47	5.89	6.86

Table 3. The average number of unique Live-On-Entry source registers for different trace line sizes.

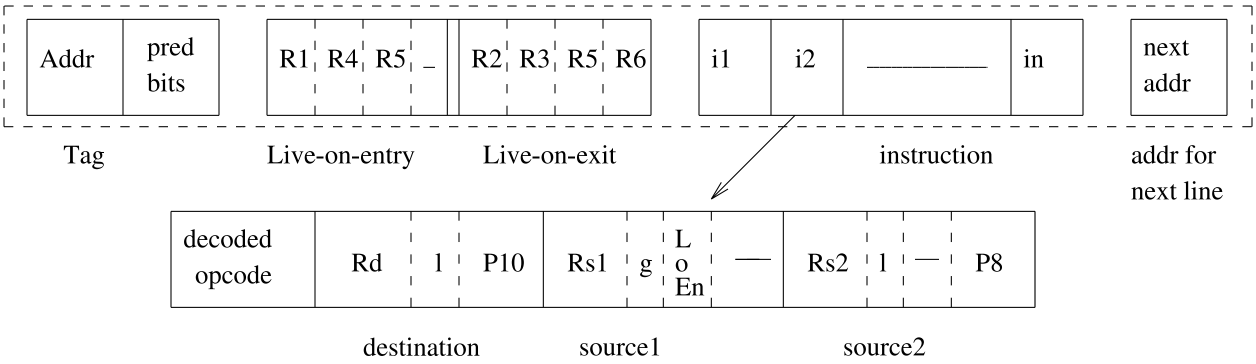


Figure 5. A block in the Renamed-Trace Cache. Live-on-entry and live-on-exit registers have to be renamed before use.

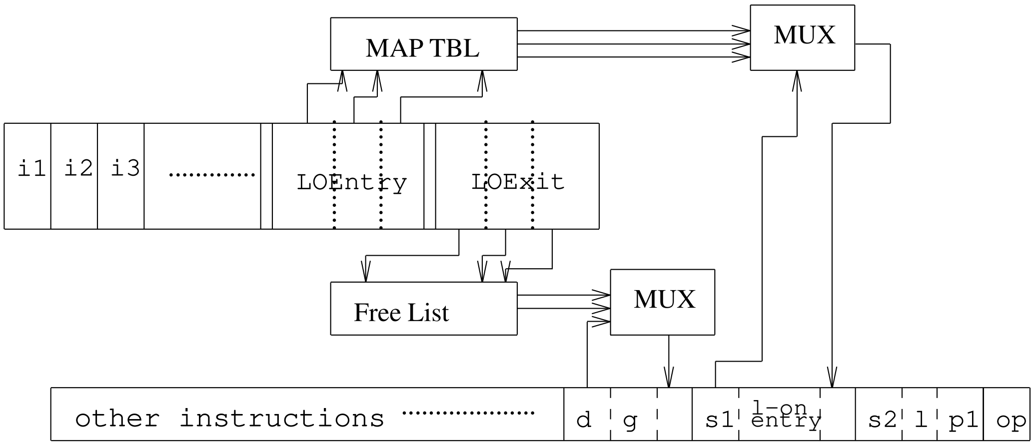


Figure 6. Single-Cycle Renaming of a Trace Line from the Renamed-Trace Cache. Source S1 and the destination for the depicted instruction pick up new mappings, but source S2 is local and uses the recorded mapping P1.

instructions simultaneously and thus can be accomplished in one cycle.

2.4. BRANCH PREDICTION BANDWIDTH

Several extensions to two-level adaptive branch predictors have been proposed to allow the prediction of more than one branch per cycle, an essential requisite for high instruction-fetch bandwidth. We adapt the 2-level adaptive block predictor [Fra95a] for the proposed architecture to obtain multiple predictions quickly. A trace line fetched from the trace cache provides the address of the next trace line in dynamic program order. The global history table corresponding to the next address indexes into a pattern history table, which provides the prediction bits to be tagged on to the next address prior to looking up the trace cache in the next cycle. Each entry in the pattern history table is a simple fixed-size bit pattern holding prediction bits, plus a mask indicating the number of valid prediction bits. The trace cache is indexed using the trace line address tagged with the prediction bits (masked to the correct size). Since the trace cache provides the next address for each trace line, the pattern history table can be enhanced to provide an alternate address usable when a prediction bit returned by the table resolves as incorrect. This can help speedup recovery from mispredictions.

2.5. PREVIOUS WORK

Decoupled access-execute architectures and their variants (for example, [Smi84a, IBM90a]) partition the instruction window into an integer queue, a floating-point queue, and in some cases a branch queue. This reduces result forwarding across the three queues as well as speeds up instruction selection since each queue has an independent set of functional units. However, some data forwarding and dependence checking is often still needed between the queues, complicating and slowing down the issue logic[Wei95a]. The register file is naturally partitioned into an integer file and a floating-point file, thus speeding up register access potentially by a factor of two. Tomasulo's algorithm [Tom67a] physically partitions the instruction window, by associating reservation stations with functional units. This speeds up instruction selection since only the reservation stations associated with a functional unit have to be checked for ready instructions. However, results still have to be forwarded to the entire instruction window.

Dependence-based architectures[Pal96a], being proposed simultaneously and independently, come closest to the trace window approach. Here the large instruction window is partitioned into multiple queues, each queue holding a chain of dependent instructions rather than a trace line. Instruction dispatch logic steers each instruction to the window partition holding its dependence-ancestors. The key differences of the dependence-based architecture are (i) the register file is not separated into local and global files, and (ii) register renaming information is not reused, thus having the same instruction dispatch rate as traditional superscalar processors. The Multiscalar architecture[Fra92b, Fra93a] takes a different approach to window partitioning and achieving large windows. Here code segments identified at compile-time are executed in parallel on multiple PEs organized as a circular chain. The program's instruction window consists of the combined instruction windows of all the PEs. Finding appropriate static program chunks, and load balancing across the PEs are some of the issues that arise in this approach.

Instruction caches that hold decoded instructions have been proposed in a variety of contexts (for example, [Dit82a, Hwu87a, Mel88a, Pat85a, Pat85b]). Static, compile-time partial renaming of individual basic blocks has been proposed in [Spr94a]. The scheme proposed in this paper records and reuses dynamic renaming done across a set of basic blocks. Recording and reuse of dynamic renaming has been proposed in the context of renaming the micro-operations of a CISC instruction[Smo95a]. The register file is partitioned into one for use across the CISC instructions and one for use by micro-operations (from multiple CISC instructions), in the context of speeding up CISC instruction decoding. The renaming mechanism requires a fairly large (for example, 512 registers) micro-operations register file. Recording of dynamic instruction issue order to achieve VLIW-like issue has been proposed in [Fra94a].

3. PERFORMANCE OF TRACE WINDOW

We evaluate the performance of the proposed trace window via trace-driven simulations with appropriate timing assumptions for the various instruction window operations. We first describe the experimental setup and then present results.

3.1. EXPERIMENTAL SETUP

Benchmarks

We use the Spec92 Integer benchmarks to capture the behavior of programs with relatively unpredictable control flow. We simulate the first 100 million instructions from each trace, except for compress which executed only 73 million instructions. (The benchmark sc is not simulated due to a problem with generating traces on our platform). The benchmarks were compiled for an IBM RS/6000 using the IBM XLC compiler, version 3.1, running under the AIX operating system version 4.1. The standard optimization flags distributed in the SPEC suite makefiles (-O2) were used. We note that not using any optimization flags would have biased the study in favor of the trace window. Lack of appropriate optimization settings considerably restricts register allocation, producing code that frequently stores and reloads variables that could have continued to reside in registers. This would have resulted in a spuriously higher proportion of local registers.

Machine Models

We simulate the performance of a representative Trace Window model (TW) and compare it primarily with that of (i) a superscalar model (SUPER) and (ii) a trace window model with ideal instruction fetch (IF). We further study the effect of the renamed-trace cache by simulating a trace window model without the renamed-trace cache (TW-NoRTC). We also consider an impossibly good superscalar model, 8SUPER. To better understand the potential for performance benefits, we consider a TW model with oracle branch prediction (TW-Oracle) and a TW model with a more restrictive trace cache (TW-LockupCache). We first mention the common machine parameters used across the models, and then describe the individual models.

All models use a trace cache of 256 lines and a perfect instruction cache. To remove the effects of limited resources on instruction window size and thus highlight the impact of pipeline latencies, we assume an unlimited number of functional units and memory ports. A 64KB 4-way set associative data cache is simulated (by interfacing Dinero [Hil84a] to our simulator) for all benchmarks except Espresso which used a 16KB cache. (The working set size of Espresso is known [Bur96a] to be significantly smaller). The functional unit and cache latencies assumed are shown in Table 4. A 2-level adaptive GAg(18)[Yeh93b] branch predictor with adequate prediction bandwidth is simulated rather than the block-level predictor based scheme described in section 2.4. The block-level predictor is known to be as accurate[Fra95a] as the GAg predictor. We assume perfect memory disambiguation in all models.

The TW model assumes 16 trace lines of length 16 instructions each, resulting in an instruction window of 256 instructions. No limit is imposed on the number of branches in a trace line. A renamed-trace cache is assumed; no limit is imposed on the number of live-on-entry/live-on-exit registers that can be remembered in a renamed trace line[3] (Figure 5). For renamed-trace cache misses, instructions are fetched from the instruction cache, renamed, and dispatched at the rate of 4 per

Function Unit	Latency (cycles)	Function Unit	Latency (cycles)
Load (Cache hit)	2	Load (Cache miss)	12
Conditional Branch	1	Uncnd. Branch.	1
Logic	1	FP Add	3
Integer Add	1	FP Mult	4
Integer Mult	4	FP Div	8
Integer Div	8		

Table 4. Functional Unit and Data Cache Latencies. The floating point functional units are rarely exercised by the benchmarks.

[3] This can be implemented by using a 1-bit mask per architectural register in the trace cache for live-on-entry information and another 1-bit mask for live-on-exit information. If the alternate implementation of explicitly storing a limited number of register identifiers is used, trace line building can be terminated when a trace line's live-on-entry/live-on-exit registers exceed the limit. The former implementation is preferable.

cycle into the corresponding trace line in the window. For renamed-trace cache hits trace lines can be dispatched to the window at the rate of one per cycle. The renamed-trace cache is implemented as an unlimited lockup-free cache, as described shortly. An instruction can start execution no earlier than the clock cycle subsequent to its entry into the window. A maximum of 2 instructions can be issued per cycle (out of order) from each trace line, resulting in a total peak issue bandwidth of 32 instructions. (The issue bandwidth is kept higher than the fetch bandwidth to allow for bursty parallelism.) Results can be bypassed between instructions from the same trace line. Instructions that write global registers spend an additional clock in the Write Back (WB) stage before dependent instructions from a different trace line can be issued. This implies bypassing from the write ports to the read ports of the global register file. We assume sufficient result buses and write ports. Upon resolution of a mispredicted branch, instruction fetch from the correct path can be initiated in the subsequent cycle.

Unlimited lockup-free operation during trace misses is supported by separating the renamed-trace cache into a trace cache storing raw trace lines (plus rename information associated with individual instructions) and a smaller, higher-associativity rename cache holding critical rename information, specifically the identities of the live-on-entry and live-on-exit registers of the trace line as well as the next address field (Figure 2). The lines in the rename cache have to always be a superset of the raw trace cache. A trace line miss in the raw trace cache miss could still be a hit in the rename cache, allowing register renaming for the missing trace line to be done even before it is fetched from the instruction cache. This allows subsequent renamed-trace cache lookups while the previous miss is being satisfied over 4 clock cycles (i.e. lockup-free operation). Assuming each renamed-trace line holds 5-bit identities of 6 live-on-entry and 6 live-on-exit registers (Tables 2 and 3) and a 12-bit next-address offset, each rename cache line is 9 bytes long compared to the corresponding trace cache line size of at least 112 bytes (4 bytes of raw instruction plus 3 bytes of local rename information associated with the instruction, for 16 instructions in the trace line). Thus the rename cache can be much smaller and of much higher associativity to provide better hit rates. We model a sufficiently large fully-associative rename cache that has only compulsory misses. The fraction of unique trace lines referenced by a program is usually less than 0.1%[Mit97a], thus keeping the required rename cache size small.

The IF model assumes 100% hit rate in the trace cache and 100% branch prediction accuracy to study the performance of the TW model under ideal instruction fetch conditions. This represents an upper bound on the performance of TW.

The TW-NoRTC model assumes that the trace cache does not record renaming and decoding information. Thus all instructions have to be renamed before dispatch, reducing the effective dispatch bandwidth to 4 instructions per cycle (the rename bandwidth). This also increases the branch misprediction penalty since in the TW model the correct path might be present in the renamed-trace cache and thus dispatched at the rate of 16 instructions per cycle. TW-NoRTC uses the same assumptions as the TW model otherwise, thus serving to isolate the benefits of the renamed-trace cache.

The SUPER model simulates a traditional superscalar processor. The improvement shown by the TW model over SUPER represents the performance benefits of the proposed architecture over traditional approaches. The cost of traditionally organizing a large window is captured by assuming that no bypassing is possible, thus incurring a 1-clock penalty of the write-back stage for dependencies. An instruction dependent on a single-cycle operation can thus start execution only 1 clock after the latter starts execution and not on the immediately subsequent cycle. We note that this assumption is optimistic, as result forwarding to a large instruction window is unlikely to be possible in a single-cycle write-back stage. We assume that any 32 instructions can be issued from the window each cycle, as opposed to the restrictions in the trace window models. In addition, we simulate an impossibly good superscalar model, 8SUPER, that has twice the register renaming bandwidth of SUPER and TW-NoRTC but is similar to SUPER otherwise.

We consider two additional machine models to better understand the performance of TW. The TW-Oracle model assumes perfect branch prediction. This model represents the improvements possible to TW by better branch handling; the difference between IF and TW-Oracle lies in the trace cache. The TW-LockupCache model differs from the TW model in that the renamed-trace cache does not support lockup-free operations. A trace-cache miss in the TW-LockupCache model results in a 5-cycle dispatch penalty (refer Figure 2).

3.2. RESULTS

Table 5 shows the instruction level parallelism sustained in the different primary machine models. The first observation is that significant speedups are provided by the TW model over the TW_NoRTC model and the SUPER superscalar model. (All speedups are with respect to the TW-NoRTC model rather than the SUPER model since the former is a more realistic machine model.) This indicates that the additional dispatch bandwidth made possible by the renamed-trace cache increases the effective instruction window available for finding independent instructions. While the speedup is significant for all benchmarks, speedups for eqntott and compress are relatively limited due to the dominating effects of branch mispredictions and cache misses. Mispredictions reduce the dispatch bandwidth by wasting cycles on the wrong control path. Data cache misses increase the lifetime of dependent instructions in the instruction window, thus reducing the effective window size. Table 6 shows that compress has the worst prediction accuracy, trace cache miss rate, and data cache miss rate of the five benchmarks, while eqntott has a significantly poorer data cache miss rate than the other three benchmarks. Further, the average lifetime of a mispredicted branch is the highest for these two benchmarks.

The significant gap between the IF model and the TW model shows the scope for improving performance through a better trace cache and through reducing the branch misprediction penalty (given the fairly high prediction accuracies). Several possible avenues for increasing the trace cache hit rate have been outlined by the trace cache proposers[Rot96a]. Compress accesses relatively few unique dynamic trace lines. We suspect pathological conflicts in the trace cache for those trace lines; this might be alleviated for example by using some of the prediction bits of the address to index the trace cache.

The branch misprediction penalty includes the delay in branch scheduling after dispatch, and the refill time from the correct path after branch resolution. The former is dependent on early scheduling of instructions that compute the condition register tested by the branch. Table 6 shows that the average lifetime (after dispatch) of a mispredicted branch is quite high for many of the bennchmarks. The hardware scheduling

Machine	IF (Peak ILP 16)		TW (Peak ILP 16)		TW-NoRTC (Peak ILP 4)		SUPER (Peak ILP 4)		8SUPER (Peak ILP 8)	
	ILP	Speedup	ILP	Speedup	ILP	Speedup	ILP	Speedup	ILP	
cc1	10.80	(5.40)	4.40	(2.20)	2.00	(1.0)	2.20	(1.10)	3.01	(1.51)
compress	7.93	(4.99)	2.25	(1.42)	1.59	(1.0)	1.78	(1.12)	2.17	(1.37)
eqntott	11.31	(5.36)	3.65	(1.73)	2.11	(1.0)	2.39	(1.13)	3.35	(1.59)
espresso	14.96	(6.31)	7.42	(3.13)	2.37	(1.0)	2.53	(1.07)	3.66	(1.54)
xlisp	11.91	(5.13)	6.17	(2.66)	2.32	(1.0)	2.44	(1.05)	3.54	(1.53)

Table 5. Performance of the various Machine Models. The sustained instructions/cycle and the speedup over a Trace Window model with no Renamed-Trace Cache and limited result bypassing (TW-NoRTC) are shown. The SUPER model has less restricted instruction issue than the TW-NORTC model, and the 8SUPER model has twice the rename bandwidth of SUPER and TW-NORTC. IF models perfect instruction fetch. The significant performance benefits of the TW model over TW-NoRTC and SUPER show the effectiveness of the proposed mechanisms.

Bench-mark	Branch Mispred. Rate (%)	Trace-Cache Miss Rate (%)	Data-Cache Miss Rate (%)	Avg. Mispred. Lifetime (Cycles)
cc1	6.00	55.06	0.51	7.9
compress	9.39	56.0	12.00	19.6
eqntott	3.53	21.87	4.24	14.2
espresso	2.43	46.87	0.51	6.8
xlisp	3.18	32.31	0.01	12.1

Table 6. Factors contributing to the lowering of effective instruction fetch bandwidth. Trace-cache miss rates model interference from fetching the wrong control path (all branches along the wrong path are modeled as predicted not taken). The average lifetime of a mispredicted branch indicates the large cost (in fetch/dispatch cycles) of the relatively few mispredictions.

considered in this paper does not give higher scheduling priority to branch deciding instructions.

The SUPER model performs 7% to 13% better than the TW-NoRTC model despite lack of bypassing. This can be attributed to the ability of SUPER to select any 32 instructions from the window for issue in a clock cycle. Experiments show that the additional penalty incurred due to lack of bypassing does not significantly affect SUPER's performance[Mit97a]. The impossibly good superscalar model 8SUPER performs almost as well as the TW model for compress and eqntott (which have orthogonal factors limiting performance), but performs well below TW for the other three programs.

Table 7 further investigates the performance of the TW model. The TW-Oracle model performs considerably better than the TW model due to perfect branch prediction; in fact it attains a very large fraction of the benefits of perfect instruction fetch (the IF model). This suggests that the relatively poor miss rates of the trace cache are overshadowed by the much longer branch misprediction lifetimes of the benchmarks. This is particularly so since the TW model assumes a perfect lockup-free trace cache and thus reduces the penalty on a trace cache miss. The results for the TW-LockupCache bear this out; the penalty for a trace cache miss is 5 clock cycles in this model. The performance drop due to the lack of the lockup-free feature supported by a lockup-free cache is as dramatic as the performance improvement of TW-Oracle due to removal of long-lifetime branch mispredictions. In fact, we had incorporated the lockup-free cache in our first simulations of the TW model, prior to simulating the TW-LockupCache model, and were surprised to discover its beneficial role. Experiments show that the lockup-free feature provides most of the benefits of a perfect (100% hitrate) trace cache for a machine with realistic branch prediction (i.e. for the TW model)[Mit97a]. Both these results point to the enormous cost of each stall cycle in a highly parallel processor organization.

Overall, the results demonstrate the benefits of the improved dispatch bandwidth provided by the proposed trace window schemes. This increase in dispatch bandwidth increases the costs associated with other sources of pipeline bubbles, such as branch mispredictions and cache misses, pointing to the importance of reducing those costs.

4. DYNAMIC VECTORIZATION

Vector instructions [Rus78a] save instruction fetch bandwidth and reduce demands on instruction window size by fetching a loop body just once and issuing multiple iterations using fairly simple issue logic. Vectorization also allows subsequent sequential code to be fetched and executed in parallel with the loop's execution. Vector instructions provide several other performance improvements which we do not elaborate. In this section, we briefly outline a hardware mechanism for vectorizing dynamic loops that fit into a single trace line of the trace window. A detailed description is not presented in this paper due to space constraints. The fraction of dynamic instructions vectorized by the mechanism is subsequently presented. A more detailed performance evaluation of dynamic vectorization is in progress.

The vectorization mechanism relies on status bits and tags associated with each trace line in the trace window. The default settings of the status bits and tags correspond to normal operation of the trace window, without the vectorization capability.

Machine	TW-Oracle (Peak ILP 16)		TW (Peak ILP 16)		TW-LockupCache (Peak ILP 16)	
	ILP	Speedup	ILP	Speedup	ILP	Speedup
cc1	9.74	(2.21)	4.40	(1.00)	2.58	(0.59)
compress	7.52	(3.34)	2.25	(1.00)	1.98	(0.88)
eqntott	10.44	(2.86)	3.65	(1.00)	2.88	(0.79)
espresso	13.97	(1.88)	7.42	(1.00)	3.72	(0.50)
xlisp	11.12	(1.80)	6.17	(1.00)	3.50	(0.57)

Table 7. Performance of the two variants of the TW machine model. The TW-Oracle model assumes perfect branch prediction. The TW-LockupCache model assumes that the renamed-trace cache does not support lockup-free operation, thus increasing the effective penalty of trace misses.

Benchmark	Fraction of Insts. Vectorized	Avg. Loop Size (Dyn. Insts.)	Loop Limit		
			(Avg.)	(Median)	(Max)
cc1	18.15%	15.12	9.75	4	1031
compress	33.38%	32.95	4.46	3	4311
eqntott	61.73%	7.11	12.38	10	2131
espresso	61.30%	14.29	6.69	3	1075
xlisp	6.20%	8.32	2.81	2	999

Table 8. Dynamic Vectorization. The fraction of dynamic instructions vectorized by the proposed hardware mechanism.

4.1. TRACE LINE VECTORIZATION

Vectorization depends on the detection of a trace line that ends in a (predicted) taken backward branch to (the beginning of) itself. To facilitate this, the occurrence of a (predicted) taken backward branch terminates the current trace line, the target of the backward branch thus starting a new trace line. Each trace line in the trace window is tagged with its address and the address of its immediate predecessor in the trace dispatch order. During trace fetch, the trace window is looked up in parallel with the trace cache to determine whether the current fetch address corresponds to the start of a dynamic loop. A hit occurs if a trace line's address *and* its predecessor tag match the current fetch address. The match of the predecessor tag is necessary to ensure correct renamed-register semantics, not discussed here due to space constraints. On a hit, a copy of the corresponding trace line is sent to the register rename stage, instead of the copy from the trace cache.

A trace line received from the trace window is renamed and dispatched to the trace window with its VECTOR status bit set, indicating to the issue logic that multiple iterations of each instruction in the trace line have to be issued. Thus the entire loop has now been dispatched in a single cycle after loop detection. On subsequent clocks, instruction fetch proceeds down the fall-through path[4] of the loop-terminating backward branch. A DISPATCH COUNTER, initialized to a maximum value, is associated with a vectorized trace line to indicate the number of loop iterations. An ISSUE COUNTER associated with each instruction indicates the number of iterations issued for that instruction. A branch instance that is resolved as mispredicted determines loop termination for the trace line. The DISPATCH COUNTER for the trace is then set to the ISSUE COUNTER of the mispredicted branch. Already issued instances beyond the loop limit are squashed; instructions and results are tagged with their ISSUE COUNTER until committed to machine state to facilitate such squashing.

We observe that different instances of a vectorized instruction refer to the same physical register identifiers. Correctness is maintained by dynamically associating queues with each operand. A queue map table, queried for each instance of a vectorized instruction, maps operands to a hardware pool of queues. When a vectorized instruction finishes execution, the corresponding queue is freed, and its tail entry is copied to the global register file.

Architectural registers that are live-on-entry and those that are modified by the vectorized trace are handled specially to insure correctness. The live-on-exit physical registers are marked busy on vector trace dispatch, and are freed only when the corresponding vector instruction's last iteration completes. Registers that are both live-on-entry and live-on-exit from a vectorized trace indicate recurrences and are appropriately handled.

Once a trace line is vectorized, most of the associated instruction issue and queue operations involved are similar to those in vector machines[Rus78a]. In this subsection we have

[4] The fall-through address for the trace-line terminating branch can be stored with the trace line in the trace cache, or in the branch predictor as the predicted fall-through address.

restricted ourselves to outlining the detection of a vectorizable trace line due to space limitations.

4.2. FRACTION OF INSTRUCTIONS VECTORIZED

We have enhanced the TW machine model to simulate the dynamic vectorization scheme. In this paper, we limit performance studies to determining the fraction of program instructions vectorized by the mechanism. The results reported in Table 8 are collected assuming unlimited size trace lines. The first observation is that a significant fraction of the dynamic instructions are captured in vector mode for all benchmarks except xlisp. For eqntott and espresso, the fraction is more than 60%. Next, the average loop limit (vector length) is fairly short, and about half the loop visits (the median loop limit) have vector lengths of 4 or less, except for eqntott with a median of 10 iterations. The maximum loop limit encountered is much larger. We draw the following conclusions. Phases of the programs around the loops with maximum loop limit will benefit from the vectorization scheme. Second, the trace cache can record the vectorized version of each trace line just as it records renaming information, and thus can reduce any vector detection overheads for repeated visits to short loops. Third, some loops (for example, the main loop in compress) repeatedly execute just two control paths within a loop a large number of times. This encourages the exploration of simple enhancements to the vectorization mechanism for capturing such loops.

4.3. PREVIOUS WORK

The CONDEL architecture[Uht92a] proposed by Uht captures a single copy of complex loops in a static instruction window. It uses state bits per iteration to determine the control paths taken by the loop and to correctly enforce dependencies. Two key differences exist between the CONDEL approach and dynamic vectorization. CONDEL captures the entire static loop body, which may not be possible if the loop body exceeds the implemented window size. We find several loops have large static body but much smaller dynamic size. Second, the CONDEL architecture does not fetch instructions from beyond the loop during loop execution. The dynamic vectorization scheme overlaps post-loop code's execution with the loop.

5. CONCLUSIONS

We have proposed a set of mechanisms to alleviate the problems caused by large instruction windows and limited register rename bandwidth in superscalar processors with large instruction fetch bandwidths. The mechanisms include partitioning of the instruction window, corresponding partitioning of the register file, and reuse of rename information to improve effective dispatch bandwidth. We find a resulting performance improvement between 1.5 to 3 times over a traditional superscalar processor for the SpecInt92 benchmarks.

Second, we observe that several loops in ordinary programs exhibit vector- like behavior at runtime. We outlined a hardware mechanism for detecting vector behavior and executing loops in vector mode. A significant fraction of the dynamic instructions are vectorized by the mechanism, up to 60% for two benchmarks. This encourages further investigation of dynamic vectorization even though the average and median loop limits (vector length) are small.

Overall our results show that capturing and exploiting dynamic behavior can considerably improve the performance of ordinary programs on superscalar processors. Further, our results also indicate that improved instruction dispatch mechanisms increase the penalties associated with disruptions to pipeline flow such as branch mispredictions, data cache misses, and instruction/trace cache misses. With improved instruction fetch and dispatch bandwidth and increasing speculation levels, we expect attention to shift from reducing the frequency of these disruptions to techniques for reducing the cost of disruptions. Previously considered techniques include prefetching for caches and fetching along multiple paths for control predictions. Squashing only control-dependent and (indirectly) data-dependent instructions on a misprediction is another previously identified direction for reducing misprediction penalty.

Acknowledgements

We thank Ravi Nair for the IBM RS6000 tracing tool. The first author would like to thank Jim Smith for his feedback on early versions of the dynamic vectorization idea.

REFERENCES

[Aus92a]
T. M. Austin and G. S. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ in *The 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[Bur96a]
D. Burger, J. R. Goodman, and A. Kagi, ‘‘Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors,’’ *23rd Int'l Symposium on Computer Architecture*, 1996.

[Con95a]
T. Conte, K. N. Menezes, P. M. Mills, and B. Patel, ‘‘Optimization of Instruction Fetch Mechanisms for High Issue Rates,’’ *22nd Annual Int'l Symposium on Computer Architecture*, June 1995.

[Dit82a]
D. R. Ditzel and H. R. McLellan, ‘‘Register Allocation for Free: the C Machine Stack Cache,’’ *Proc. Int. Symp. on Arch. Support for Prog. Lang. and Operating Sys.*, March 1982.

[Fra92a]
M. Franklin and G. S. Sohi, ‘‘Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors,’’ *25th Annual Symposium on Microarchitecture*, Dec. 1992.

[Fra92b]
M. Franklin and G. S. Sohi, ‘‘The Expandable Split Window Architecture for Exploiting Fine-Grain Parallelism,’’ in *The 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[Fra93a]
M. Franklin, ‘‘The Multiscalar Architecture,’’ *Ph.D. Thesis, University of Wisconsin-Madison*, 1993.

[Fra94a]
M. Franklin and M. Smotherman, ''A Fill-Unit Approach to Multiple Instruction Issue,'' *27th Int'l Symposium on Microarchitecture*, Dec. 1994.

[Fra95a]
M. Franklin and S. Dutta, ''Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors,'' *28th Annual Symposium on Microarchitecture*, Nov. 1995.

[Hao96a]
E. Hao, P-Y. Chang, M. Evers, and Y. Patt, ''Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures,'' *29th Annual Int'l Symposium on Microarchitecture (to appear)*, Dec. 1996.

[Hil84a]
M. D. Hill and A. J. Smith, ''Experimental Evaluation of On-Chip Microprocessor Cache Memories,'' *Proc. 11th Annual Symposium on Computer Architecture*, June 1984.

[Hwu87a]
W. W. Hwu and Y. N. Patt, ''Design Choices for the HPSm Microprocessor Chip,'' in *Proc. 20th Annual Hawaii International Conference on System Sciences*, Kona, HI, January 1987.

[IBM90a]
IBM, ''Special Issue on the IBM RISC System/6000 Processor,'' *IBM Journal of Research and Development*, January 1990.

[Lam92a]
M. S. Lam and R. P. Wilson, ''Limits of Control Flow on Parallelism,'' *Proc. International Symposium on Computer Architecture*, May 1992.

[Mel88a]
S. W. Melvin, M. C. Shebanow, and Y. N. Patt, ''Hardware Support for Large Atomic Units in Dynamically Scheduled Machines,'' in *Proc. 21st Annual Workshop on Microprogramming and Microarchitecture*, San Diego, CA, November 1988.

[Mit97a]
Tulika Mitra, ''Performance Evaluation of Improved Superscalar Issue Mechanisms,'' in *M.E. Project Report*, Dept. of Computer Science, Indian Institute of Science, January 1997.

[Pal96a]
S. Palacharla, N. Jouppi, and J. E. Smith, ''Quantifying the Complexity of Superscalar Processors,'' *Univ. of Wisconsin-Madison Technical Report*, vol. CS-TR-96-1328, November 1996, (Available at http://www.cs.wisc.edu/trs.html; a version to appear in ISCA'97).

[Pat85a]
Y. N. Patt, W. W. Hwu, and M. Shebanow, ''HPS, A New Microarchitecture: Rationale and Introduction,'' in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, December 1985.

[Pat85b]
Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, ''Critical Issues Regarding HPS, A High Performance Microarchitecture,'' in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, December 1985.

[Rot96a]
E. Rotenberg, S. Bennett, and J. E. Smith, ''Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching,'' in *29th Annual Int'l Symposium on Microarchitecture*, Paris, Dec. 1996.

[Rus78a]
R. M. Russel, ''The Cray-1 Computer System,'' *Communications of the ACM*, vol. 21, pp. 63-72, Jan. 1978.

[Smi84a]
J. E. Smith, ''Decoupled Access/Execute Architectures,'' *ACM Transactions on Computer Systems*, Nov. 1984.

[Smo95a]
M. Smotherman and M. Franklin, ''Improving CISC Instruction Decoding Performance Using a Fill Unit,'' *28th Annual Symposium on Microarchtecture*, Dec. 1995.

[Spr94a]
E. Sprangler and Y. N. Patt, ''Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme,'' *27th Annual Int'l Symposium on Microarchitecture*, Dec.1994.

[Tom67a]
R. M. Tomasulo, ''An Efficient Algorithm for Exploiting Multiple Arithmetic Units,'' *IBM Journal of Research and Development*, January 1967.

[Uht92a]
A. K. Uht, ''Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream,'' *IEEE Transactions on Computers*, vol. 41, July 1992.

[Wal91a]
D. Wall, ''Limits of Instruction Level Parallelism,'' *4th International Conf. on Arch.Support for Prog.Langs. and Op.Sys.*, April 1991.

[Wei95a]
Shlomo Weiss, ''Implementing Register Interlocks in Parallel-Pipeline, Multiple Instruction Queue, Superscalalr Processoors,'' *Proc. First Int'l Symposium on High Performance Computer Architecture*, 1995.

[Yeh93b]
T-Y. Yeh and Y. N. Patt, ''A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,'' *20th Int'l Symposium on Computer Architecture*, 1993.

[Yeh93a]
T-Y. Yeh, D. MArr, and Y. Patt, ''Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache,'' *Proc. 7th ACM Int'l Conference on Supercomputing*, July 1993.