Early Branch Resolution using a Fast Pre-Execution Core on a Dynamically Scheduled Processor

David M. Koppelman

Department of Electrical & Computer Engineering, Louisiana State University

koppel@ece.lsu.edu

Abstract

Branch resolution time is a significant factor in the performance of wide-issue dynamically scheduled superscalar machines. Some of that time is spent fetching, decoding, renaming, scheduling, and moving operands of a correct-path instruction after a branch misprediction.

An alternate organization is described in which a trace-cache like front end provides identical streams of partially renamed instructions to a processor having two execution cores. A main core completes the decode and renaming and passes instructions to conventional schedulers for dynamically scheduled execution. A fast core immediately executes at least some of the instructions, providing early branch resolution. Instructions not handled by the fast core are completed in the main core. The fast core can execute load instructions that hit the cache and one-cycle integer instructions, subject to restrictions such as register value availability and the availability of cached instruction information.

The fast core provides low-latency branch resolution and to some measure load prefetch, while the main core can manage a large window. The performance of such a system running SPEC-cpu2000 integer and floating-point programs was simulated. Despite limitations on instructions, register availability, and disambiguation, the system was able to resolve a significant number of branches, reducing the pipeline portion of resolution time by half and realizing as high as a 10% average speedup for very deep pipelines.

1. Introduction

One cost a dynamically scheduled system pays to achieve out-of-order execution of instructions from a large window (pool of un-executed instructions) is the number of stages an instruction must pass through from fetch to execution. So long as branches (and other control transfers) are correctly predicted this "cost" does not impact performance. On a misprediction however the flow of new instructions into the window stops until the first correct-path instruction traverses the stages, the amount of time is called the *misprediction penalty*.

In some dynamically scheduled processors the misprediction penalty is relatively short, in the MIPS R10000 it is about five cycles [18], in the PA-8000 it is about six cycles [5], and in the Alpha 21264 it is about seven cycles [10]. Assuming recent versions of these processors have the same misprediction penalties then larger penalties are correlated with higher clock frequencies and performance based on the SPECcpu2000 disclosures (The MIPS R14k at 500 MHz rates 500 SPECint2000, PA-8700+ at 875 MHz rates 875 SPECint2000, and Alpha 21264C at 1250 MHz rates 928 SPECint2000).

The move to higher clock frequencies and the need for larger instruction windows to tolerate load latency will both tend to increase the penalty. Recent gains in processor performance have come primarily through increases in clock frequency and many believe this trend has not yet

run its course. With larger instruction windows more time is needed for associative lookups in the processors' schedulers, increasing the number of stages in the misprediction penalty. Larger windows also increase the time needed for movement of data between distant functional units. The impact of these trends can be seen in the Pentium 4 which has both a high clock frequency (over 3 GHz) and a large window, 126 in-flight instructions [8]. For this it pays the price of a misprediction penalty of twenty cycles (in part due to the use of a trace cache).

Though deeper pipelining has improved performance, the misprediction penalty is becoming a larger barrier to improved performance. An alternative processor organization, the *fast-core processor*, (FCP) is described here which reduces the misprediction penalty by providing a much shorter path to execution for certain instructions.

The fetched instruction stream is passed both to the conventional decode pipeline in the *main core*, and to a *fast core* which may execute the instructions much sooner. The fast core can resolve branches and so initiate recovery much sooner than the long path, reducing the misprediction penalty.

The fast core is made fast by limiting the kinds of instructions that can execute there, imposing a fixed functional unit assignment, relying on cached dependency information, and limiting register forwarding. These restrictions limit the number of instructions that can execute in the fast core, to about 30% in the simulated systems. In addition to initiating branch misprediction recovery, results from the fast core are forwarded to the main core after spending a fixed number of cycles in the fast core.

Unlike other schemes for early resolution of branches [1,4,7,12,19], the FCP does not fork instruction fetch ahead to some future point. Instruction fetch is always on the predicted path, there is no duplication of fetch resources.

Some schemes for reducing branch penalty fetch, and perhaps execute, both sides of a branch, duplicating at least some resources for the two paths [15]. As stated above, FCP fetches only on the predicted path and avoiding wasted.

The remainder of this paper is organized as follows. Details on the fast core and front end are presented in the next section. The simulator and benchmarks are described in Section 4. Experiments are described and discussed in Section 5. Related work is in Section 6 and finally conclusions appear in Section 7.

2. Fast-Core Processor

2.1. Overview

Figure 1 shows the overall organization of the FCP. A trace-cache like front end provides an identical stream of partially renamed instructions to two cores, the *main core* and the *fast core*. The main core prepares instructions for conventional dynamically scheduled execution while the fast core is designed to execute instructions as soon as possible, especially those leading up to a branch.

Branches that resolve in the fast core can initiate recovery, yielding a reduction in the branch penalty. Instructions exit the fast core after a fixed amount of time, called the *sojourn*, at which time any values they produced are forwarded to the main core, reducing the number of instructions that must execute there. In the simulated systems the sojourn is chosen so the latest time an instruction can execute in the fast core is one cycle before its earliest execution in the main core.

2.2. The Fast Core

The fast core is essentially a simple dynamically scheduled processor. It consists of two or three register files, a set of integer functional units, and possibly a load unit. (The mix was chosen to resolve control transfers.) It executes instructions that have been partially renamed; the partial

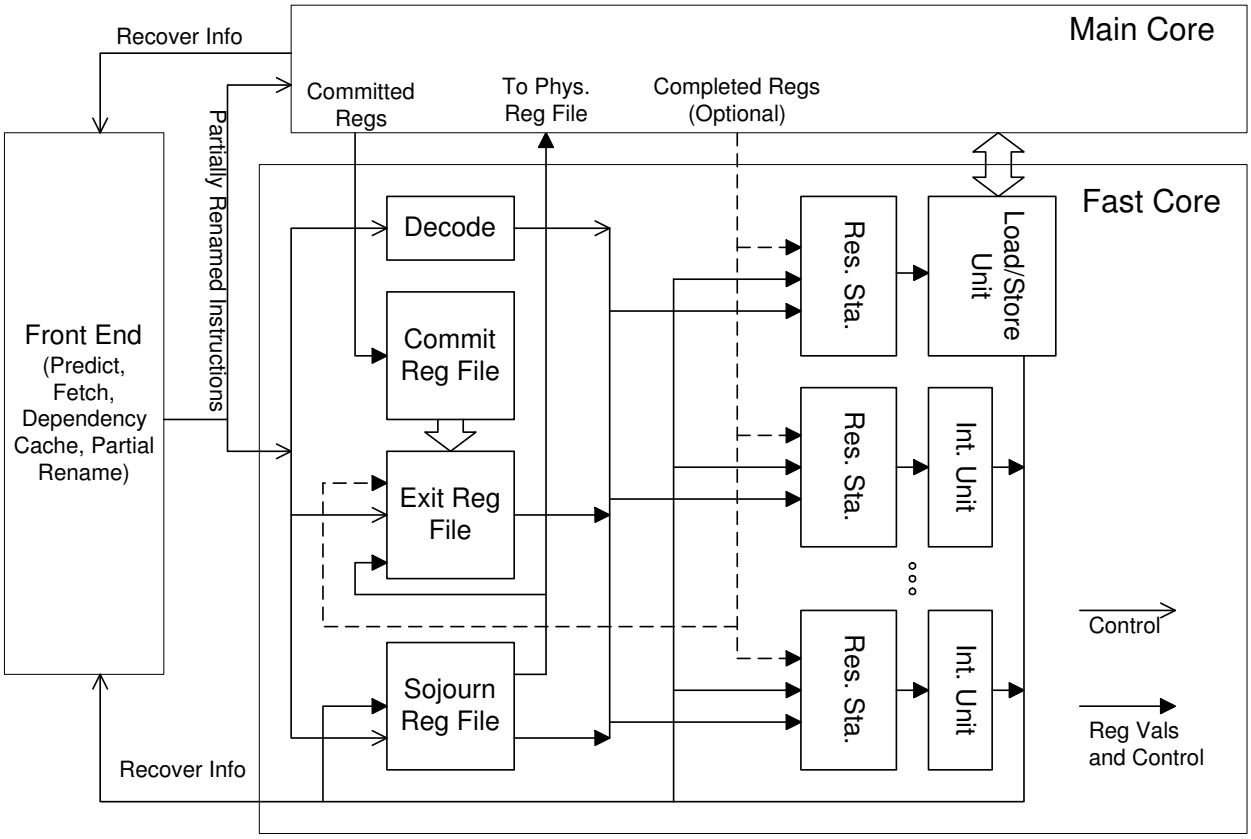

Figure 1. A fast core processor showing details of the fast core.

renaming produces an *index* for each source and destination register, see the section on renaming below. One register file, the index-addressed *sojourn register file* (SRF) holds results of instructions as they complete in the fast core. The other, the register-addressed *exit register file* (ERF) is written as instructions exit the fast core. The SRF is always properly updated, but the ERF is not and so some instructions may not be able to execute in the fast core. The fast core may also use a *commit register file*, updated when instructions in the main core commit. (*Commit*, also called retire, is the last step in the execution of an instruction; instructions are commit in program order to enable recovery.)

During their first cycle in the fast core instructions that had been renamed take operands from either the SRF or ERF; the ERF value is used if the operand index is out of range (indicating the writer has left the fast core), otherwise the SRF value is used. Instructions that were not renamed do not execute in the fast core. It is possible to retrieve an invalid value from the ERF, that indicates either that the writer has not completed or that it did complete in the main core but the ERF was not updated. An instruction reading such an operand also does not execute in the fast core.

During its second cycle in the fast core an instruction is copied to a reservation station at a functional unit and if the unit is idle it can start executing at that cycle. For maximum performance, the choice of functional unit is based solely on the instruction's position in the group of instructions provided by the front end.

When instructions complete they write the SRF; when they exit the fast core the result is copied to the ERF. The result, if available, is also written to the main core's physical register file using a mapping provided by the main core. The main core's scheduler can detect or compute which fast core instructions have or will execute and so can schedule any dependent instructions that do not execute in the fast core.

At its simplest the ERF is updated only by instructions exiting the fast core. This limits fast-core instructions to operands generated in the fast core, a significant limitation. Two variations were used to update the ERF using values from the main core, *Update* and *Recover*.

In the *Update* variation instructions update the ERF one cycle after they complete in the main core; the update completes if an *expanded index* stored in the ERF matches that of the instruction. The values are also bypassed, with the one-cycle delay, to waiting fast-core instructions.

The Update variation is costly because it requires write ports and bypass paths for each main core instruction that can simultaneously write back. The base configuration does not use the Update variation.

The *Recover* variation can update the ERF with a full set of register values when mispredicted branches (and other events requiring recovery) are resolved in the main core. In the Recover variation a *commit register file* is maintained, written by instructions as they commit (from the main core). The commit register file is copied to the ERF on a misprediction recovery if all instructions before the mispredicted branch commit before correct-path instructions following the branch reaches the main core. Otherwise, the ERF remains invalid. This recovery method is quite effective, providing nearly the performance of perfect recovery. (Note that since the ERF is updated as instructions exit the fast core a reorder buffer flush starting in the fast core does not affect the ERF.) Though Recover is effective at recovering registers on a misprediction it does not perform nearly as well as the Update variation at other times.

The Recover variation requires no extra ERF ports and has no direct connections to the fast core functional units, and so it is less expensive than Update and has little performance impact. It is used in most of the simulated configurations.

To simplify things and to limit instructions to those leading to a branch the fast core includes only integer ALUs and in some variations a connection to the load/store unit. As simulated, the fast core cannot execute floating-point instructions, any instruction requiring more than one cycle (except loads), any instruction updating more than one register (not including the condition codes), and store instructions. The result of a load instruction is only used if it hits the level-1 cache or its result is bypassed from a store in the main core, otherwise dependent instructions must execute in the main core. There is no way to bypass a value from a store in the fast core to a load; a dependence predictor prevents loads from executing ahead of unresolved stores (which includes all stores in the fast core). Instructions that were not retrieved from a BTC hit also cannot execute. (This rule could be relaxed for zero-source operand instructions.) Instructions which can execute in the fast core are called *candidates*.

2.3. Renaming

The FCP exploits a two-step, cached-dependency, register renaming scheme. Though such a renaming scheme is not necessary for FCP it simplifies the design and helps the realizability of the eight-way and wider systems for which FCP is targeted. The first step, *index generation*, maps register names to *indices*, essentially a serial number of the instruction writing the register. The fast core uses these indices directly while the main core, in a second step, maps them to physical register numbers. Critical paths in the two-step renaming hardware are short and are independent of the number of dependencies present.

The front end maps instructions in units called *block groups*; in the systems simulated these span two basic blocks. Destination registers (in effect instructions) are assigned indices sequentially. Cached information on block groups is used to map source registers, this is retrieved from a PC-addressed *block tree cache* (BTC). A BTC entry stores the register number and a *distance* for each source operand of each instruction in the block group. The distance identifies which instruction in the block group (if any) wrote the corresponding source register. A zero means the writer precedes the block group, a distance of one indicates that the immediately preceding instruction wrote the

value, etc. If the distance is non-zero it is used to generate the index, otherwise the index is retrieved from a register-number-addressed *register to index map* (RIM). A valid index from the RIM identifies the instruction that wrote the register, while an invalid index indicates that either the instruction is beyond the fast core's window or precedes the current run of BTC cache hits, whichever is shorter.

The register index map is updated using the block group's live-out registers (registers written but not overwritten); a RIM entry that is not written may be invalidated. An entry is invalidated if its index is in the range of indices to be assigned in the next cycle. Note that this requires an adder for each entry in the RIM, either to generate the new index (instruction position plus a base index) or determine if the old one needs to be invalidated. With a limit of 56 instructions in the fast core, only six-bit adders are needed, much less costly than a 64-bit ALU using carry-look-ahead logic.

The new RIM computed in one cycle is reused in the next cycle and is also saved for misprediction recovery.

The main core can use the indices to map source registers using two tables, a register-addressed *register map* (RMAP) (a conventional register map) and an index-addressed *index map* (IMAP). If the index for a source register is valid the IMAP is used otherwise the RMAP is used. The RMAP is updated one cycle earlier using the destination register index and a free physical register. If indices for source operands are not available then the processor falls back to a conventional register mapping scheme and passes dependency information to the front end for caching. On the simulated systems this mapping proceeded at half speed (*e.g.*, four instructions per cycle in an eight-way system).

2.4. Front End

FCP requires for its operation cached instruction dependency information. There are many ways of maintaining this, the chosen method is perhaps a hybrid between a *branch address cache* [17] and a *trace cache* [9]. A branch address cache is part of a multiple branch predictor; its predictions are fed to a multiported cache for instruction retrieval. In a trace cache instructions are retrieved from the trace cache itself, avoiding the extra step at the cost of storage for the trace cache.

In the system chosen here instruction information, though not complete instructions, is stored in the branch address cache, renamed a *block tree cache* (BTC). The BTC is smaller than a trace cache, both because unneeded instruction information is omitted and because information in interior tree nodes is shared by all paths through them (there would be a separate trace cache entry for each). Nevertheless a conventional trace cache (one predicted path per entry) could also be used.

Briefly, the BTC works as follows: Starting with a program counter (PC), an entry in the PC-addressed block tree cache is retrieved; it provides information on the tree of blocks reachable from the PC. That information includes the length of each block, the type of control-transfer instruction (CTI) at its end, and the CTI's target (if not indirect). A branch predictor, using information in the BTC and separate pattern history tables (PHTs) (indexed using information computed in the previous cycle) is used to predict a path through the tree of blocks. The path provides instruction cache lookup addresses, the PC for the next BTC access, and instruction information. For the systems simulated here a YAGS predictor is used. The systems simulated predict the target of an indirect branch in the next cycle (introducing a bubble) using a global-history-register–indexed table and predict returns using a return address stack. For more information see [3,6,11,17].

In the base systems simulated a BTC entry holds three nodes and is limited to 16 instructions (distributed any way), with at most 12 instructions from root to leaf, thus limiting rename to 12 instructions. Other configurations were simulated.

Table 1. Base Configuration Parameters

Base Parameters	Value
Decode Width	8-way Superscalar
Reorder Buffer	256 instructions
Front End Rename	12 insn per cycle (BTC hit).
Main Core Rename	4 insn per cycle (on a BTC miss).
ID to EX	8 cycles (1 cycle for **Ideal**).
BTC	2^{13} entries, height 2, 16 insn per entry.
Global Branch History	16 branches
Return-Address Stack	8 entries
Integer Units	4×2 clusters
Floating-Point Units	4
L1 ICache	256-B Line
L1 ICache	4-way, 256 kiB
L1 DCache	4-way, 64 kiB
L1 DCache Hit Latency	1 cycle
L1 ICache Ports	4.
L2 DCache	8-way,64-B Line, 256 kiB
L2 Hit Latency	11 cycles
L2 DCache Miss Latency	$\approx$ 100 cycles
Memory Units	4
Load/Store Queue	32 entries
Miscellaneous Configurations	**Value**
4-Way BTC	2^{13} entries, height 2, 16 insn per entry.
4-Way Front End Rename	12 insn per cycle (BTC hit).
16-Way BTC	2^{13} entries, height 3, 32 insn per entry.
16-Way Front End Rename	18 insn per cycle (BTC hit).

3. Evaluation

3.1. Simulator

The systems were analyzed using RSIM [13], a detailed microarchitecture simulator. Modifications were made to simulate fast core processor and other unrelated modifications were made; that is, they impact the reported performance of systems.

RSIM is a microarchitecture simulator which simulates a dynamically scheduled superscalar processor and memory system. The processor implements a subset of the SPARC V8 ISA [16]. Benchmark programs are compiled exactly as they are for a real system. Linking is identical except for the use of static libraries (though still the system's libraries, not specially prepared versions) and a special startup file. System calls are not simulated.

Dynamic execution is aggressive: The register map used for renaming is checkpointed when branches or jumps are decoded so that recovery can start when mispredicted instructions resolve. Exception recovery is initiated when the faulting instruction is ready to commit.

3.2. Benchmark Programs

The simulated programs come from the SPECcpu2000 suites, though using reduced input sizes to reduce simulation time, and other sources. The seven programs used are bzip2, gcc (cc1), gzip, mcf, perl, TEX, and mesa. Benchmark bzip2 is used to compress a copy of the GNU General Public License; gcc is used to compile (with O3 optimization) the integrate.c program in the gcc distribution, gzip is used to compress text, mcf uses the SPEC2000 test input, perl runs a script that analyzes a Web server log, TEX is run on Chapter 17 of The TeXbook, mesa uses the SPECcpu2000 reference input but at reduced size and frame rate. Benchmarks gzip, mcf, and mesa are compiled using the SPEC CPU2000 makefiles, using code from that suite. The code for

the other benchmarks was obtained from their standard distributions, compiled with optimization. Optimization was targeted to an UltraSPARC II processor, so scheduling would not perfectly match the wider-issue systems simulated here.

3.3. Configurations

The base system on which FCP is applied is a future-generation 8-way clustered dynamically scheduled machine using the front-end described above. Eight-way fetch/decode was chosen because most current machines are four way. The multiple branch predictor in the advanced front end provides sufficient fetch bandwidth and also the dependency information needed to rename eight instructions per cycle.

High clock frequencies make it difficult to bypass results between physically distant functional units and large windows make it difficult to quickly schedule instructions. To reduce these effects the integer functional units are clustered into two groups of four, and the floating-point units are in their own cluster. Each cluster has an instruction queue which can issue only to functional units in the cluster. Instructions are assigned to clusters in the decode pipeline based on the functional unit needed, and when there is more than one cluster, on availability of operands and issue queue space. Instructions are removed from issue queues when they are complete and there is no chance of re-execution (as there is with loads executing ahead of unresolved stores). There is a one-cycle delay when bypassing results between clusters. (The integer functional units in the Alpha 21264 are also clustered, though with a different issue queue (arbiter) organization [10].)

As described above, the earliest execution opportunity for an instruction in the simulated system is eight cycles after decode. This number is longer than any of the dynamically scheduled RISC processors, but shorter than the 11 cycles in the Pentium 4 (rename to execute, see [8]). The number seems like a reasonable extrapolation of the delays for the RISC processors to future implementations and is conservative when applied to deeply pipelined processors such as the Pentium 4. (The Pentium 4 translates its CISC instructions into micro-ops which are stored in a trace cache. The 11-cycle pipeline length does not include the IA-32 to micro-op translation stages.)

The number of integer functional units in the fast core of the FCP systems is the same as the cluster size, usually four. Issue queue size always refers to the size in each cluster, not the total for the processor.

A full list parameters for the simulated configurations appear in Table 1.

4. Experiments

4.1. Base Set

The performance benefit of a fast core is evaluated by comparing such systems to conventional systems using the same total number of ALUs, for roughly equal-cost systems and by comparing it to conventional systems using fewer ALUs, showing the potential to boost performance.

The base system to which FCP is compared, C8, has two 4-ALU clusters, for a total of 8 ALUs. That is compared to FCP systems also having 8 ALUs, (one cluster of 4 in the main core and four in the fast core), and FCP systems having 12 ALUs. The 8- and 12-ALU FCP systems with a memory unit are labeled **C4F4MR** and **C8F4MR**, respectively, and those without are labeled **C4F4R** and **C8F4R**, respectively. A system with the Update variation is also shown, **C8F4MUR**. (The "R" indicates the Recover variation and a "U" indicates the Update variation.) Comparisons were done for systems having 16- and 64-entry issue queues in each cluster. (The fast core has the equivalent of 14-entry issue queues.) A conventional system in which instructions execute one cycle after `ID` (versus 8 for the conventional system) is labeled **C-Idl**.

The CPI for each system is plotted in Figure 2, the percent speedup over **C8** appears at the top of the plot. The fast core systems are not effective when their 4 fast-core ALUs are "taken"

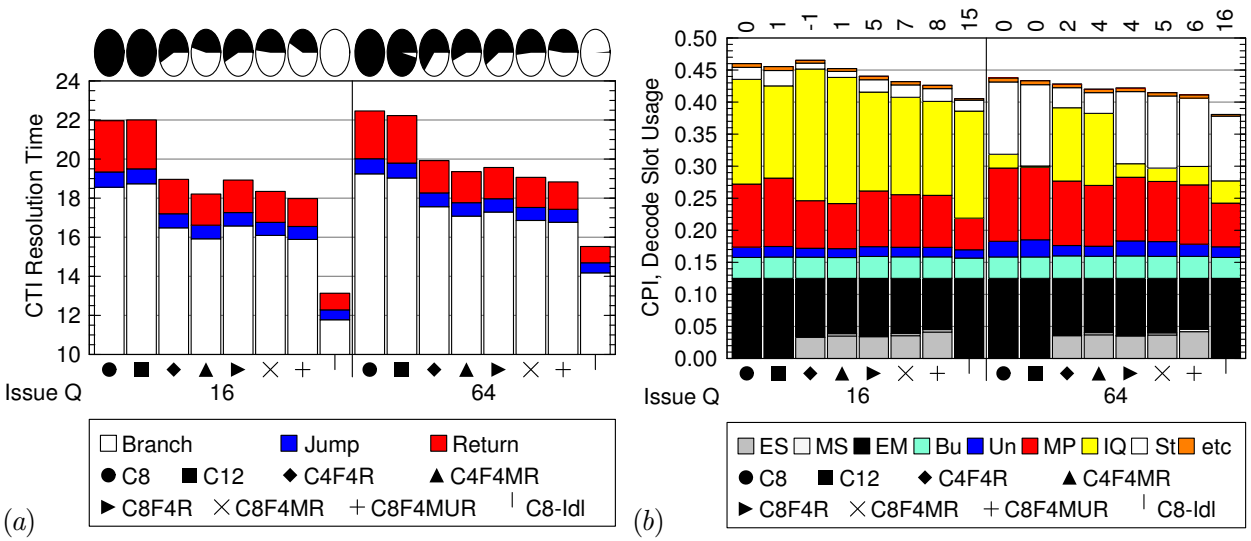

Figure 2. Branch resolution times (a) and performance (b) of conventional systems, C8, C12, an ideal system, C8-Idl, and fast-core processors in the base configuration with two issue queue sizes. (a) shows CTI (branches and jumps) resolution time (ID to EX), the pies show the fraction of pipeline

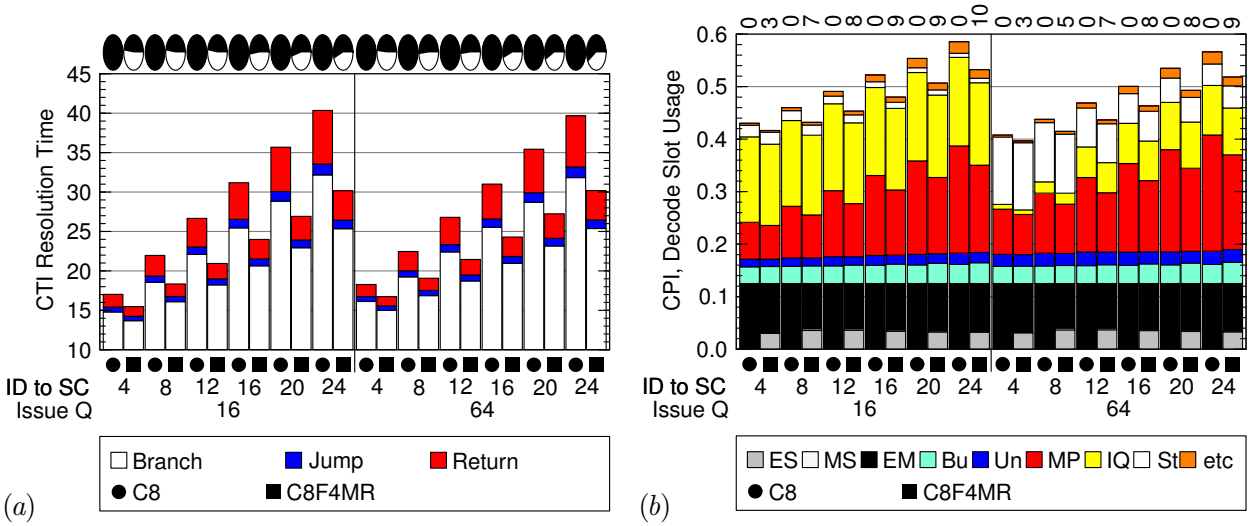

Figure 3. Branch resolution times (a) and performance (b) of a conventional system, C8 and and a fast-core processor, C8F4MR, on systems with varying pipeline delays from decode up to the earliest execution opportunity, ID to SC.

from the main core, **C4F4R** and **C4F4MR** show little or now speedup. However, for systems in which the main core "keeps" its eight ALUs, there is a 5% speedup without a memory unit and 7% with. The costly Update variation adds another point in speedup. In contrast, a conventional system with 12 ALUs enjoys only a 1% speedup (or nearly 3% speedup omitting one benchmark). The speedups on individual benchmarks vary, with three 10% or higher. The ideal system yields a speedup of 15%.

As the data showed, for roughly equal-cost 16-entry issue queue systems FCP yields little benefit, but if additional ALUs are to be added they are much more effective in a fast core than the main core. The situation is different with 64-entry issue queues. The equal-cost systems yield higher, though still modest performance improvement, while the higher cost systems, have a smaller margin over the lower ones.

FCP is designed to improve performance by reducing the impact of pipeline delay on the resolution of branches. In fact, roughly half of those cycles are eliminated, as shown by the pies at the top of Figure 2(b). The bars show the average resolution time for mispredicted CTI's, of which the pipeline delay is a small part.

Much of the improvement in pipeline delay on several benchmarks is due to procedure returns, despite the presence of a return-address stack. One reason is that gcc and other programs have subroutines that return to the callers, caller, confusing the return address stack. Branches also show some improvement.

There are several reasons why only about half the achievable reduction in pipeline delay is achieved. The primary culprit appears to be branch conditions depending on loads. Many loads do not execute because of the presence of stores in the fast core (it would be too time-consuming to insert them into the load /store queue).

Data from systems in which the pipeline depth was varied is plotted in Figure 3. The delay from decode to just before the earliest execution was varied from 4 to 24 cycles (the base is 8). Since instructions occupy reorder buffer slots staring at the first decode stage, the reorder buffer size was adjusted so all systems could hold the same number of instructions in an executable state. As expected, an FCP yields higher speedups with deeper pipelines, up to at 10% average improvement.

Due to several factors, the fast core will not be able to execute every instruction. In fact, the fast core is executing about one third of committed instructions, as shown by the segments in Figure 2(a);. The segments show how efficiently instructions are being executed by tallying the instructions that pass through *decode slots*, or the reason for their absence. The segments marked **EM** show instructions that will be executed in the main core; for the base processor this is the ideal execution time (that is, all other segments represent some kind of degradation). The segments marked **ES** show non-memory instructions executed in the fast core and **MS** show memory instructions executed in the fast core. (In FCP's, the three types of segments show the ideal execution time.)

The segments labeled **MP** (misprediction) show slots occupied by instructions that will be squashed. By resolving control transfers early FCP reduces the number of squashed instructions, this can be seen in the figure. As one would expect from the branch resolution times, a FCP still squashes a substantial number of instructions. Some of these are due to limits of the fast core, for example, the inability to bypass stores, but many branches wait on missing loads to resolve, something a FCP cannot help. (Loads are executed earlier, but so are most branches, so there is no net advantage.)

The instructions executed in the fast core do not need to be placed in issue queues in the main core, reducing the number of stalls due to full issue queues in systems retaining two main core clusters. This effect can be seen by examining the **IQ** segments, which show stalls due to a full issue queue. (Only the main core issue can fill, there are never enough instructions in the fast core to fill its issue queues.) The effect however is slight because the fast core only avoids instructions that would quickly leave the issue queues anyway.

A much larger effect is the increase in issue queue stalls when the main core has only one cluster (and issue queue). This effect dominates performance with 16-entry queues, but with 64-entry queues the issue queue stalls are replacing reorder buffer fills, shown by **St**.

Other performance limiters are unused decode slots (due to short basic block size or BTC misses) shown by **Un** (unused); reorder buffer fills (mostly due to cache misses) shown by **St** (stall); and miscellaneous flushes (exceptions, serialized instructions) shown by **etc**.

4.2. Fast Core Variations

The fast core is designed to be fast and inexpensive, requiring as few functional units and as simple a register recovery scheme as possible. The performance of some variations of differing costs

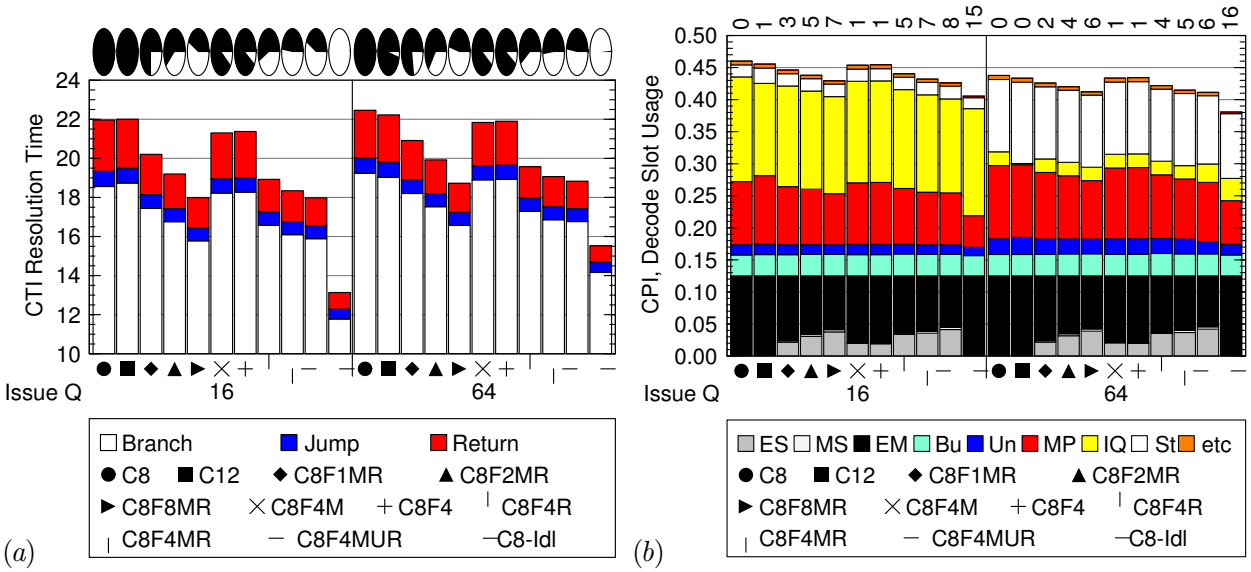

Figure 4. Branch resolution times (a) and performance (b) of conventional systems, C8, C12, an ideal

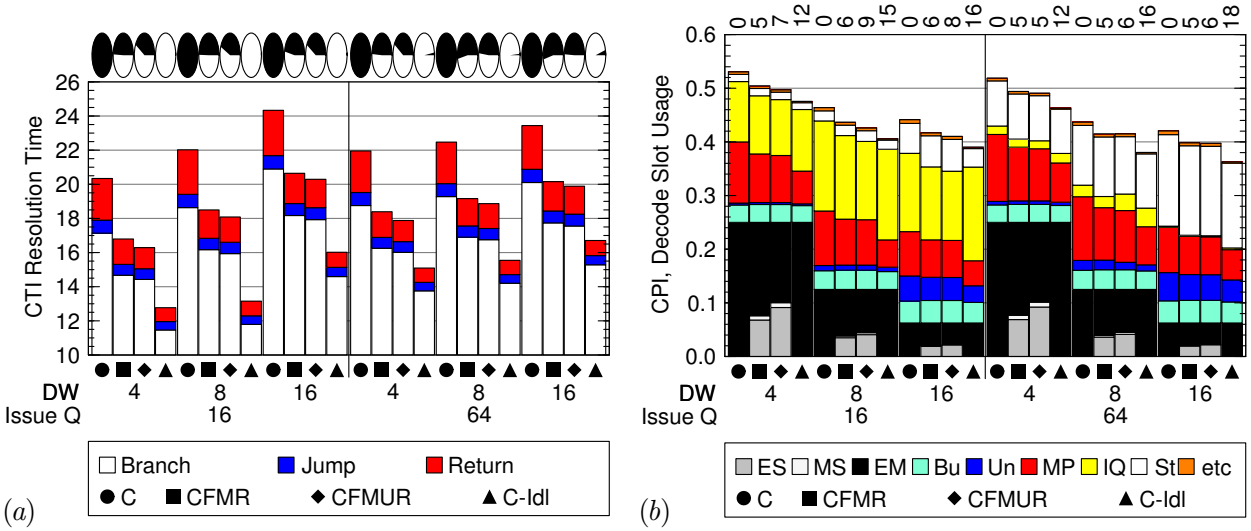

Figure 5. Branch resolution times (a) and performance (b) of conventional systems, C8, C12, an ideal system C8-Idl, and two fast-core variations on 4-, 8-, and 16-way processors (DW) using two issue queue sizes.

are shown in Figure 4. The first three FCP systems, **C8F1MR**, **C8F2MR**, and **C8F8MR** have 1, 2, and 8 ALUs in the fast core, respectively. The 4-ALU version, **C8F4MR** is also plotted. There are two-point improvements in speedup moving from 1 to 2 to 4 ALUs, but no improvement from 4 to 8. System **C8F4** cannot get register values produced in the main core, it barely yields any speedup at all, a memory, **C8F4M**, unit doesn't help. Comparing **C8F4R** to **C8F4MR** shows the benefit of the fast core memory unit, two speedup points. Allowing main-core instructions to send results to the fast core, **C8F4MUR**, adds another point of speedup, at the cost of a much larger bypass network in the fast core.

Figure 5 shows the performance of FCP on 4-, 8-, and 16-way superscalar systems. These systems all have depth-3 BTCs, something the 16-way system needs. Programs running on wider systems should have lower execution time but roughly the same number of branch mispredictions, and so the benefit of reducing the time needed to resolve these branches should increase. That

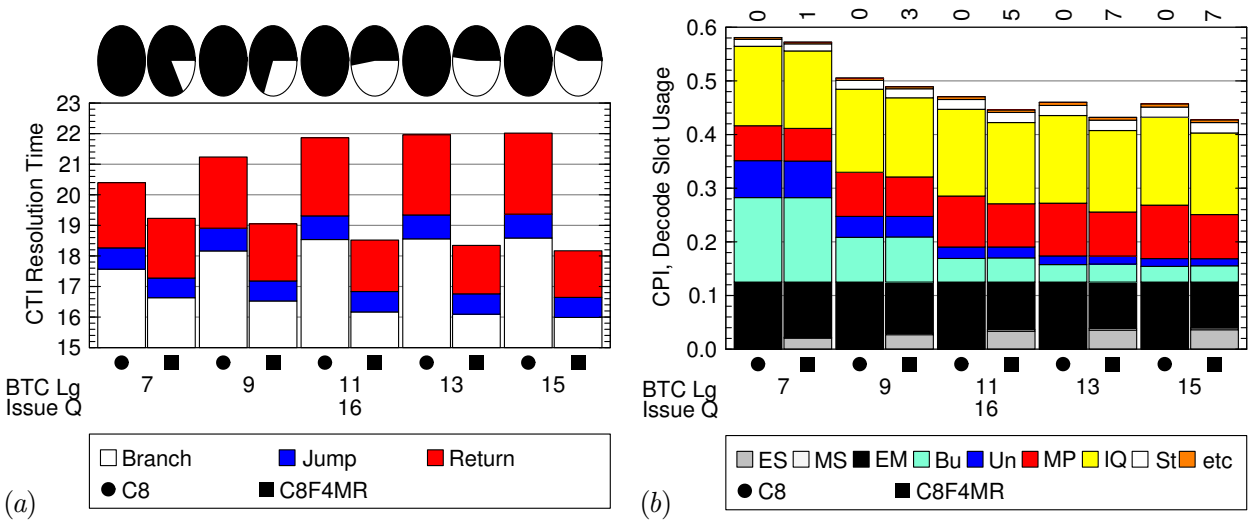

Figure 6. Branch resolution times (a) and performance (b) of a conventional system, C8 and a fast-core system, C8F4MR, on systems with different block tree cache sizes. BTC Lg is the log_2 of the number of entries in the block tree cache.

effect is seen from 4- to 8-way systems, but not to 16-way systems, perhaps because they are only slightly faster.

The fast core relies on cached dependency information in the BTC. Experiments in which the BTC size was varied to determine sensitivity to BTC size, the data are plotted in Figure 6. With a small BTC, 128 entries, the fast core can find few instructions to execute and so there is almost no improvement. Large sizes yield better performance.

5. Related Work

A FCP resolves branches early by executing them in a streamlined processing element. There has been much current interest in both early resolution of branches and partitioning processors into smaller and faster sections.

5.1. Early Branch Resolution

Techniques for early branch resolution rely on either running or jumping ahead of the main fetch stream to resolve, or at least speculate, a branch direction. The direction is communicated to the main fetch stream in time.

Farcy, Teman, Espasa, and Juan identify branches that are frequently mispredicted, as well as instructions producing its condition [7]. When these instructions are fetched they are executed using predicted values using the processor's existing functional units, and their results (other than the branch outcome) are not used by instructions in the main stream.

They exploit, in part, data prediction to reduce branch resolution time. This has the advantage of greater reduction in resolution time but also makes it more difficult to use results for the main stream. In contrast, FCP executes instructions without predicted values (except for performing loads ahead of unresolved stores), so results can be used by the main stream. Farcy's scheme requires advance identification of branches, something the FCP does not need to do (other than a BTC hit). The results reported by Farcy do not include execution times (since they consider only certain types of branches).

Annavaram, Patel, and Davidson, describe a system in which the front end pre-computes load addresses to be used for prefetching [1]. The loads and instructions generating their addresses are identified and later executed by a *pre-computation engine*. The fetch unit feeds the pre-computation

engine, taking advantage of the ability of fetch to get ahead of the main processor. No attempt is made to merge with the main program stream and early branch resolution is not investigated.

The techniques above rely on the main instruction stream being fetched. Others have looked at jumping ahead to prefetch loads and resolve branches. A set of instructions leading to a load, a *slice* is identified and fetched based on some trigger. Methods of finding the slice vary; examples of such schemes are in [4,12,19].

The difficulty in all of the schemes above is in good identification of code for advance execution. The only value is for prefetching or anticipating branches, so executing code which does not generate useful prefetches or predictions only consumes resources, potentially slowing down other operations. In contrast, the results of FCP's execution are used, potentially lowering the hardware requirements for the main core. There is no requirement for identifying code, eliminating such hardware or software mechanisms.

5.2. Fetching Both Branch Directions

Several investigators have looked at fetching both directions of a predicted branch, the schemes differ in how far one goes down the predicted wrong path.

In a scheme described by Pierce and Mudge fetch down the wrong path only goes as far as the instruction cache [14]. In constrast, Uht, Sindagi, and Hall [15] fetch, decode, and execute both branch directions, discarding the direction that turns out to be incorrect. Klauser, Paithankar, and Grunwald refine this *eager execution* using branch prediction confidence estimation []. Between the two extremes, Aragon, Gonzales, Gonzales, and Smith fetch, decode, and sometimes schedule, but do not execute predicted wrong path instructions [2].

Each of these schemes makes inefficient use of resources by having some parts of the processor process instructions that are guaranteed to be on the wrong path. Another limitation of these schemes is the number of branches they can speculate on. With branch resolution easily taking over 10 cycles on deeply pipelined systems, with 8-way fetch, and with branches occurring every five or six instructions, any system that goes down both branch directions would have to fan out in way too many directions to cover the resolution time. In contrast FCP resolves branches as early as possible.

6. Conclusions

A FCP design provides early resolution of branches while allowing for a large instruction window in the main core. The technique yields from 7 to 10% performance improvement, not fully realizing the 15% improvement of an ideal system. Perhaps the biggest impediment to performance is the execution of load instructions. Something closer to the full potential may be realized if a workable scheme could be found for bypassing stores to loads in the fast core without slowing the fast core.

7. References

[1] Murali Annavaram, Jignesh M. Patel, and Edwared S. Davidson, "Data prefetching by dependence graph pre-computation," in *Proceedings of the International Symposium on Computer Architecture,* June 2001, pp. 52-61.

[2] Juan L. Aragon, Jose Gonzalez, Antonio Gonzalez, and James E. Smith, "Dual Path Instruction Processing," in *Proceedings of the International Conference on Supercomputing,* 2002, p. 220–229.

[3] Po-Yung Chang, Eric Hao, and Yale N. Patt, "Target prediction for indirect jumps," in *Proceedings of the International Symposium on Computer Architecture,* June 1997, pp. 274–283.

[4] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen, "Dynamic speculative precomputation," in *International Symposium on Microarchitecture,* December 2001, pp. 306-317.

[5] David A. Dunn and Wei-Chung Hsu, "Instruction Scheduling for the HP PA-8000," International Symposium on Microarchitecture, 1996, pp. 298–307.

[6] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," International Symposium on Microarchitecture, December 1998, pp. 69-77.

[7] Alexandre Farcy, Oliver Temam, Roger Espasa, and Toni Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998, pp. 59-68.

[8] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal,* , 2001.

[9] Quinn Jacobson, Eric Rotenberg, and James E. Smith, "Path-based next trace prediction," International Symposium on Microarchitecture, December 1997, pp.14–23.

[10] R. E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro Magazine, March 1999, vol. 19, no. 2, pp. 24–36.

[11] Johnny K. F. Lee and Alan Jay Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer,* vol. 17, no. 1, pp. 6-22, January 1984.

[12] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi, "Slice-processors: an implementation of operation-based predictoin," in *Proceedings of the 15th International Conference on Supercomputing,* 2001, pp.321-334.

[13] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "RSIM reference manual version 1.0," Rice University Dept. of Electrical and Computer Engineering, August 1997, Technical Report 9705.

[14] Jim Pierce and Trevor Mudge, "Wrong-Path Instruction Prefetching," International Symposium on Microarchitecture, 1996, pp. 165–175.

[15] Augustus K. Uht, Vijay Sindagi, and Kelley Hall, "Disjoint eager execution," International Symposium on Microarchitecture, 1995, pp. 313–325.

[16] David L. Weaver and Tom Germond (eds.), "The SPARC architecture manual, Version 9," Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[17] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the International Conference on Supercomputing,* 1993, pp. 67–76.

[18] Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro Magazine,* pp. 28-40, April 1996.

[19] Craig Zilles and Gurindar Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the International Symposium on Computer Architecture,*, June 2001, pp. 2-13.