The Benefit of Multiple Branch Prediction on Dynamically Scheduled Systems*

David M. Koppelman
Department of Electrical & Computer Engineering, Louisiana State University
koppel@ece.lsu.edu

Abstract

Providing instructions for wide-issue systems is a challenge because the execution path must be predicted before branches are even decoded. Since basic blocks can easily be smaller than the issue width of near future systems multiple branch prediction is important.

An early solution to the problem is the multiple branch predictor (MBP) of Yeh, Marr, and Patt. A PC-indexed branch address cache (BAC) provides information on all basic blocks within a number, say 3, basic blocks of the PC; a path through the blocks is chosen by a global-history branch predictor.

The published evaluation did not include overall performance numbers, a shortcoming shared by many subsequent studies of similar systems. To fill this gap dynamically scheduled processors using a MBP or a conventional mechanism were simulated using RSIM. For comparison a multiple branch predictor based on superblocks, described by Reinman, Austin, and Calder was also simulated.

Compared to single branch prediction, 8-way superscalar MBP systems show speedup of 10% on some SPECcpu integer benchmarks on a resource-constrained system, the superblock predictor realizes an improvement of 8%. MBP tree depths of 2 and 3 show nearly equal performance. As one would expect, MBP systems fill more decode slots, but data show that much of the performance gain is due to providing more time for dependent integer instructions. Much smaller are gains due to early initiation of memory operations (except when using fast caches) and a less frequently empty reorder buffer.

1. Introduction

That instruction supply is important for wide-issue superscalar processors has long been recognized. Early limit studies found that execution rates of well over 20 instructions per cycle (IPC) were possible [15], though feasible rates would be much lower.

Recognizing that both these execution rates and the decode width of near-future processors exceed the basic block size of many programs, investigators developed instruction supply mechanisms that could fetch more than one basic block per cycle.

An early example is the multiple branch predictor of Yeh, Marr, and Patt, called an *Y-MBP* here. A PC-indexed branch address cache (BAC) provides information on the tree of basic blocks within a number, say 3, basic blocks of the program counter (PC); a path through the blocks is chosen, in one variation, by a global-history branch predictor [18]. Several other schemes have been described in which a tree of reachable blocks is stored [3].

The evaluation of the Y-MBP and other such branch predictors looked at their ability to supply instructions to a perfect execution engine, measuring an effective fetch rate (correct-path instructions fetched per cycle) [18]. Actual performance would be lower due to limited available instruction level parallelism (ILP) and load latencies.

A truer picture of performance would require simulation of a complete processor. Surprisingly such data on tree-type multiple branch predictors is lacking. Such data has been published on other multiple branch predictors, for example the *FTQ* of Reinman, Calder, and Austin [13,14] (called a superblock predictor here) and various trace caching techniques. (See Section 7 for further discussion.)

To fill this gap a system based on the Y-MBP, but with modifications appropriate for a dynamically scheduled system is analyzed here and compared to a conventional system using a YAGS branch predictor, [4], and to the superblock predictor. In one comparison the amount of storage needed by the BAC is subtracted from that needed by the instruction cache, providing some measure of the tradeoffs. The comparison also accounts for the longer branch penalty of the Y-MBP (due to the complexity of multiple fetch).

* Presented at the Workshop on Duplicating, Deconstructing, and Debunking, at the 29th Annual International Symposium on Computer Architecture, May 2002. Presentation slides available via http://www.ece.lsu.edu/koppel/pubs/bac_wddd_pr.pdf.

The goal of the analysis is not just to determine IPC but to identify performance limiters. Factors that limit instruction fetch (bubbles, squashed instructions, a full reorder buffer, etc) and instruction commitment (ILP, load latency, and a lack of instructions) are separately plotted.

Though other tree-type multiple branch predictors also lack execution time analyses, the Y-MBP was chosen because in a sense it is simplest and so a logical starting point.

Some might argue that multiple branch prediction is unnecessary because performance will be limited by memory latency to an IPC below the basic block size. This occurs in some of the simulations performed here, but others show usable speedup. Advances in prefetching and data prediction will reduce the limit on IPC imposed by memory, increasing the need for multiple branch prediction.

The remainder of this paper is organized as follows. Details on the multiple branch predictors is presented in the next section. Timing details for the simulated system follow in Section 3. The conventional system is described in Section 4. The simulator and benchmarks are described in Section 5. Experiments are described and discussed in Section 6 and their relation to other instruction supply mechanisms is discussed in Section 7. Conclusions appear in Section 8.

2. Multiple Branch Prediction

2.1. Overview

All instruction supply mechanisms starting with something akin to a program counter must identify a *fetch group*, a set of instructions to fetch (perhaps in the next cycle), and a program counter value to use in the next cycle. In conventional systems the fetch group must be contiguous, that is, only the first instruction can be the target of a control transfer (branch, jump, etc.). Multiple-branch-prediction schemes can generate non-contiguous fetch groups, to handle these the system would need a multiported cache or a trace cache to perform the fetch.

2.2. Branch Targets

A differentiating feature of MBP systems is how branch targets are stored. In conventional systems a branch target buffer [7,12] or next-line predictor might store branch targets, usually indexed with the address of the branch or some instruction that preceded it. In an MBP targets for multiple branches must be retrieved in a single cycle, precluding chained BTB lookups. Three approaches have been used. In the most general, used by the Y-MBP, a PC-indexed *branch address cache* BAC stores the tree of all blocks reachable from the PC to a certain distance [18], called the *order* here. Each tree node stores the length of the block, the type of *control transfer instruction* (CTI) at its end, and possibly a target. Predictors determine a path through the tree.

A 2^d-fold increase in size over a BTB is the price paid by an order-d BAC to avoid chained lookups. To avoid this size problem entries in the PC-indexed *Fetch Target Buffer* (FTB) used by the MBP of Reinman *et al* store information only on blocks reachable by not-taken branches [14]. An FTB entry is similar to a BAC entry node, except it describes a *superblock* that can span multiple not-taken branches.

Predictions made by the FTB are queued, smoothing irregularities in FTB entry size. With favorably occurring not-taken branches the FTB could keep pace with the Y-MBP. For the systems analyzed here though it could not keep pace, even with an order-2 Y-MBP.

Between storing an entire tree or just superblocks, an MBP might store common path segments. That is, a PC-indexed *path cache* might store the most common paths originating at the PC. Path selection would be an interesting problem since there are many ways to divide the execution paths into segments to store. The results presented here show that order-2 Y-MBPs provide sufficient fetch rate for 8-way systems, and so the advantage a path cache would have to be fewer BAC (or path cache) entries.

2.3. Branch Direction Prediction

As with target prediction, multiple branch prediction cannot make use of chained lookups. This rules out any scheme that requires precise branch history because only the address of the first block (used to find branch history) is known at lookup time and that block may be in another entry. Precise history is important for local history predictors [17,10] but is not needed for others such as bimodal and gshare [8].

Yeh *et al* do show how to precisely implement a global history predictor, GAg [18], in which a pattern history table holding 2-bit counters is indexed by a global history register. At the beginning of each cycle d lookups are made in a d-ported PHT. The first port provides one counter, for the first branch, the second port provides counters for the next two reachable branches (which are stored contiguously), etc. This scheme is precise in that the PHT contents are the same as a PHT in a conventional GAg system.

Reinman's MBP uses hybrid prediction; each FTB entry holds a two-bit chooser and bimodal counter, a GHR is also maintained; the chooser selects the more

Table 1. Branch Address Cache Entry Fields

Name, Size	Description
Tag, 17	BAC entry tag (part of address not used to index the BAC). The BAC is indexed using the address of the first instruction in the block at the root of the block tree.
Valid, 1	Set if entry valid.
Block Tree, $44(2^d - 1)$	Tree describing basic blocks within $d-1$ control transfers of the root. Space allocated for complete binary tree, $2^d - 1$ nodes.

successful predictor. Only the last branch in a superblock is predicted this way, the others are assumed not taken. Other predictors are discussed in Section 7.

For all the systems simulated here, including the conventional system, Y-MBP, and superblock, YAGS predictors were used [4]. In YAGS a bimodal predictor provides a base prediction which is used to select one of two GHR $\oplus$ PC-indexed *direction* PHTs, taken or not-taken. An entry in these tables holds a partial tag (portion of PC) and a two-bit counter. If the tag matches the base prediction is overruled; direction tables are updated on a hit or on a misprediction by the base predictor.

The conventional system uses YAGS as described. In the Y-MBP implementation the direction tables are indexed by the PC used to index the BAC exclusive-or'ed with a GHR shifted 0 to $d-1$ times, reading a d-ported PHT. With this scheme, the branch at the root of the block tree uses the YAGS lookup scheme while subsequent branches use the "wrong" branch address, but the correct GHR, to index the PHT. Each node in the block tree has a two bit counter used for making the base prediction. (Retrieving a counter from a BHT for each branch would not be possible in one cycle.)

In the superblock system only the last branch in a block is predicted, the PHTs are indexed using the GHR, updated for the not-taken branches, exclusive-or'ed with the superblock starting address. Unlike the system described by Reinman, a superblock is lengthened if the last branch was not taken eight consecutive times.

2.4. Other Control Transfers

In their original analyses Yeh *et al* predict only branches; Reinman *et al* also predict indirect jumps. A return address stack is used for returns and a *jump target buffer* JTB is used to predict other indirect jumps [14].

All systems simulated here use identical methods to predict indirect jumps. A return address stack is used for returns and a GHR-indexed JTB is used for other indirect jumps.

3. System Timing

Reinman *et al* evaluated their superblock predictor on an aggressive 8-way dynamically scheduled machine using two-level predictor tables with latencies chosen to match their sizes. One goal was to demonstrate that the predictor could run far enough ahead to usefully prefetch the second-level predictor tables and instruction cache. In contrast Yeh *et al*, who published six years earlier, focused for the most part on prediction accuracy, using fixed constants such as branch resolution time to estimate an effective fetch rate.

Here, the two schemes will be evaluated on identical systems, similar to Reinman's but without the multi-level predictor tables. The diagram below shows the stages an instruction passes through from fetch to commit. Bars indicate where an instruction's progress might be delayed (without stalling the pipeline) while waiting in a queue.

```
IF | CA AR | ID P1 P2 P3 P4 | P5 P6 P7 P8 EX WB | C
```

In `IF` the PC of the instruction (or its predecessor) is used to perform BAC and PHT lookups (or a JTB lookup is performed). A fetch group is constructed and placed in a queue while one item is dequeued (or bypassed) for each idle cache port. Cache lookup proceeds during `CA` (cache). Some time during `AR` (arrange) instructions arrive (assuming no miss), they are arranged in to program order and queued. Decode starts in `ID`; in the simulated system a reorder buffer entry is allocated in this cycle. Decode, rename, and the like proceed up to `P4`; `P5` shows the first step of dispatch, actual execution occurs in `EX`. The simulated system has a perfect load miss predictor and so instructions dependent on a load enter `EX` as soon as the data can be bypassed. Eight cycles after leaving `ID` an instruction is said to be *ripe*, indicating that it can execute (if dependencies are satisfied).

Stages `WB` and `C` are writeback and commit. Branch predictors are updated in `WB`; no attempt is made to recover or repair table entries for resolved branches that are later squashed, only the GHR is recovered. (Commit-time update did not work as well.)

Misprediction recovery (for branches and jumps) starts in `WB`. A checkpointed register map and GHR are used to restore system state. The `IF` of the correct-path instruction occurs in the same cycle as the `WB` of the resolving branch, for a minimum 13-cycle misprediction recovery time.

Table 2. Block Tree Node Fields

Name, Size	Description
Type, 4	Type of control transfer at end of block. Five bits needed to distinguish minor CTI variations, such as annulled branches.
Length, 10	Number of instructions in basic block.
Target, 30	Target address of control transfer at end of block.

Table 3. Base Con£guration Parameters

Common Parameters	Value
Decode Width	8-way Superscalar
Reorder Buffer	256 instructions
Return-Address Stack	8 entries
L1 ICache	256-B Line
L1 DCache Hit Latency	1 cycle
L2 DCache	8-way,64-B Line, 256 KiB
L2 Hit Latency	10 cycles
L2 DCache Miss Latency	$\approx$ 100 cycles
L1 ICache Ports	4.
ID to EX	9 cycles.
Global History	16 branches
Integer Units	8
Floating-Point Units	4
Memory Units	4
Base Configuration	**Value**
L1 ICache (MBP,super)	4-way, 64 KiB
L1 ICache (Conv)	7-way, 112 KiB
FTB,BAC	2^{13} nodes
L1 DCache (Conv)	4-way, 64 KiB
Large Configuration	**Value**
L1 ICache (MBP and super)	4-way, 256 KiB
L1 ICache (Conv)	7-way, 448 KiB
FTB,BAC	2^{15} nodes
L1 DCache (Conv)	4-way, 256 KiB

4. Comparison Conventional System

The MBP systems are compared to a system that uses a conventional instruction supply mechanism. Misprediction recovery is slightly faster on this system. The conventional system uses a next-line predictor to provide instruction cache lines to the decode logic and a YAGS [1,4,6] branch predictor to predict branch directions. As long as predictions are correct this system can fetch instructions each cycle (no branch bubbles). The next-line predictor is not particularly sophisticated.

The next-line predictor works by maintaining, in effect, a linked list of decode groups. A decode group is a group of n contiguous instructions, where n is the decode width. The pointers to the next group are stored in the instruction cache along with the lines; also stored is a two-bit counter used to predict whether the link should be modified. Using these predictions the fetch mechanism buffers two lines, and passes the correct one, if any, to decode.

Branch direction is predicted in decode using a YAGS predictor [4]. Indirect jumps are predicted in ID and there is a one-cycle bubble before the target is available (ID to ID). Like the MBP system, indirect jumps are predicted using a GHR-indexed jump target buffer. The timing is identical to the MBP systems starting at the `ID` stage.

5. Evaluation

5.1. Simulator

The systems were analyzed using RSIM [9], a detailed microarchitecture simulator. Modifications were made to simulate multiple branch prediction 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, though their impact on the execution of the chosen benchmarks should be small.

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. See the previous sections for timing.

5.2. Benchmark Programs

The simulated programs come from the SPEC suites, though using reduced input sizes to reduce simulation time. The six programs used are bzip2, gcc (cc1), gzip, mcf, perl, and TEX. Benchmark bzip2 is used to compress a copy of the GNU General Public License; gcc is used to compile (with O3 optimization) a program for finding hidden words in text, gzip is used to compress text, mcf uses the SPEC2000 test input (though is limited to committing 125,000,000 instructions), perl runs a script that analyzes a Web server log, and TEX is run on the GNU Emacs quick reference card. The number of committed instructions ranges from about 23 million for bzip2 to 125 million for mcf. Benchmarks gzip and mcf are compiled using the SPEC CPU2000 makefiles, using code from that suite. The code for the other benchmarks was obtained from their standard

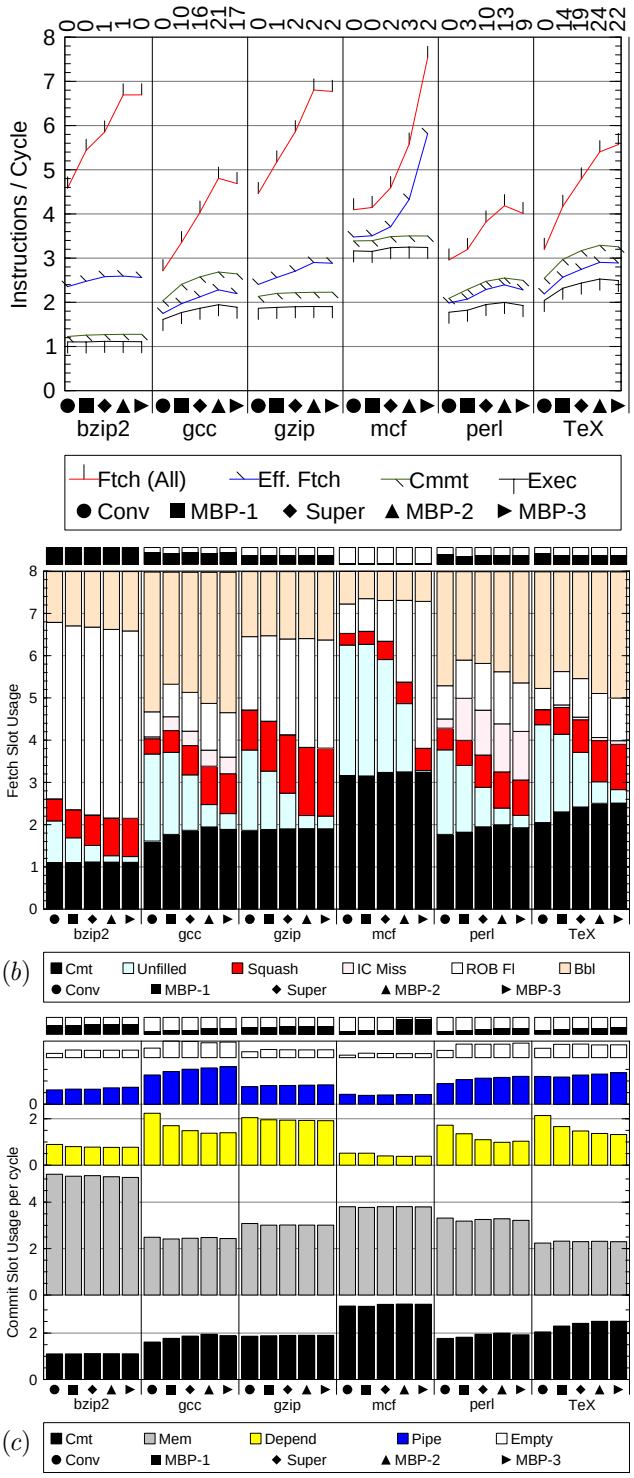

Figure 1. Performance of the base con£guration, shown by (*a*) fetch and commit rates with speedup along the top, (*b*) fetch slot usage with mispredict rate $\times 10$ along the top, and (*c*) commit slot usage with reorder buffer occupancy along the top.

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

5.3. Configurations

The performance results are presented for two main configurations, base and large. The base configuration is the most realistic; the instruction cache and BAC suffer a significant miss rate for the three larger benchmarks, gcc, perl, and TEX. The conventional system has a larger instruction cache so that the combined storage for the instruction cache and BAC are the same in the advanced fetch systems. Configuration parameters appear in the tables below.

The simulated systems were sized to represent a system realizable in the next ten years. A large instruction cache was chosen to focus attention on the fetch mechanism itself; the icache hit ratio is near 100%. The level-1 and level-2 data caches are 64 KiB (2^{16} B) and 256 KiB, respectively. Though they may seem small, for the problem sizes used they produce hit ratios of about 70% for level 1 and over 95% for level 2. There are an ample number of functional units and a 256-entry reorder buffer, see the tables below.

All systems use a 16-bit global history register. The YAGS predictors use 2^{16}-entry tables for the choice and direction tables; the direction tables use 8-bit tags. All systems have four cache ports (though the conventional system can only use one).

Superblock and Y-MBP systems of different orders but using the same amount of storage will be compared. The amount of storage will be given in nodes. An order-d Y-MBP BAC entry uses 2^d nodes (one node stores the BAC entry tag), an FTB entry uses two nodes. An order-d Y-MBP with a 2^m-entry BAC uses 2^{d+m} nodes. A KiB is 1024 bytes.

6. Experiments

Up to six systems were evaluated, the conventional one, the superblock system, and order 1 to 4 Y-MBP systems. In the figures' key an order-d Y-MBP system is labeled MBP-d. The Y-MBP and superblock systems are collectively called advanced supply mechanisms.

6.1. Base Set

The speedups of the advanced supply mechanisms on the base configuration are shown along the top of the plot area in Figure 1(a). The results show some performance improvement on the base configuration, MBP-2 provides the best average speedup, 10%, followed by MBP-3, super and MBP-1, at 9%, 8% and 4%, respectively. Running TEX the speedup was as

high as 24% for MBP-2. Assuming MBP-1 and super are of similar cost (they both predict one branch per cycle) the superblock predictor is the better of the two. Though table storage is the same for all systems (except the conventional system) MBP-2 and MBP-3 would be more costly because of the need to predict multiple branches per cycle. Ignoring that cost MBP-2 is best, with MBP-3 suffering due to BAC misses.

Though performance improvement is modest the MBP systems are effectively fetching instructions, as can be seen in the top series, **Ftch (All)**, which shows the *fetch rate*, the number of instructions reaching decode divided by the number of cycles in which the reorder buffer is not full. MBP-2 comes close to the maximum fetch rate, eight instructions per cycle on mcf and bzip2. The rate is lower on TEX, gcc, and perl due to instruction cache misses and the large number of indirect branches. The fetch rate of super falls between MBP-1 and MBP-2, indicating that the average superblock size is less than two blocks.

The bottom line in Figure 1 (*a*) shows the *execution rate* in instructions per cycle; it does not enjoy nearly as much improvement as fetch rate. Suspended between the fetch rate and the execution rate is the *effective fetch rate*, labeled **Eff. Ftch**, the number of committed instructions divided by the number of cycles the reorder buffer is not full.

Its distance below the fetch rate is determined by the number of squashed instructions, which in turn is determined by prediction accuracy, fetch rate and the time to resolve mispredicted control transfers. The distance between the effective fetch rate and the execution rate is determined by the fraction of time the reorder buffer is full, which is in turn a function of the *commit rate*. The commit rate, **Commit**, is the number of committed instructions divided by the number of cycles the reorder buffer holds ripe instructions. (Instructions that have been decoded long enough ago to have reached a functional unit, assuming they were ready to execute.)

The difference between commit rate and effective fetch rate and the sensitivity of commit rate to window size determine where the advanced supply mechanisms are effective. They help those benchmarks in which the commit rate exceeds the effective fetch rate, as one would expect. Briefly, the reason for the improved performance is the larger window size (more items in the reorder buffer). In the three well-performing benchmarks the increase in effective fetch rate is matched by an increase in commit rate and so they are limited by control transfer prediction accuracy. The other benchmarks are limited by insensitivity of commit rate to further increases in window size.

Figure 1 (*b*) shows the fate of instructions that passed through a *fetch slot* or reasons why an instruction did not; the plot is scaled to instructions per cycle. Segment **Co** shows instructions that passed through and were committed; its height is the execution rate, the same as **Exec** in (*a*). Segment **Un** (unfilled) shows slots that are unfilled because the fetch mechanism returned less than eight (in this case) but more than zero instructions that later ripen. This segment clearly shows the effectiveness of the advanced supply mechanisms.

Slots holding wrong-path instructions that ripen (and are later squashed) are shown by **MP** (mispredict). The number of squashed instructions should increase faster than fetch rate. The amount of time between when a branch is fetched and when it ripens is fixed by the pipeline length, so the number of fetched instructions during this time is proportional to the fetch rate. The time to resolve the branch *increases* with fetch rate since instructions the branch depends upon arrive closer to when the branch arrives. For that reason much of the additional fetch rate is wasted. For the systems simulated the mispredict and nonpredict rate is roughly the same for the different instruction supply mechanisms. That misprediction rate is shown in the segments along the top of the plot in (*b*). A full box indicates a 10% mis/non-predict rate (including indirect jumps, returns, and branches), an empty box indicates perfect prediction.

Slots holding wrong-path instructions that never ripen are labeled **SC**; also under this category are unfilled slots fetched with the wrong-path instructions and delays due to unpredicted jumps and next-line-predictor mispredictions. The slight increase with fetch rate is due to the same number of mispredictions occurring in fewer cycles.

Segment **IC** shows slots unfilled due to instruction cache misses. These are significant for gcc and perl, and also present for TEX, they partially explain why their fetch rates are lower than the ideal, (*a*) **Ftch (All)**, while filling most fetch slots, **Un**.

Finally, segment **St** shows the cycles in which the contents of a fetch slot did not advance due to a full reorder buffer or some other resource limit.

Potential speedup due to the higher fetch rate is not realizable when commit rate is not sensitive to further increases in window size, resulting in a more frequently full reorder buffer. This occurs for three of the benchmarks, bzip2, gzip, and mcf; but for the others there is a significant increase in commit rate, and so much of the potential speedup is realized.

Factors affecting commit rate are available ILP, load latency, and window size. The impact of a higher fetch rate on these factors can be seen in Figure 1(*c*) where commit slot usage is plotted. (The commit slots can

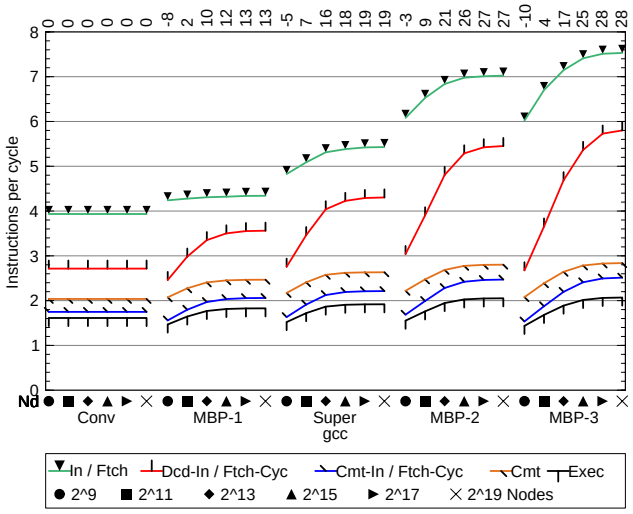

Figure 2. Performance v. BAC size of systems running gcc, perl, and TEX. BAC (FTB) size given in $\log_2$ nodes.

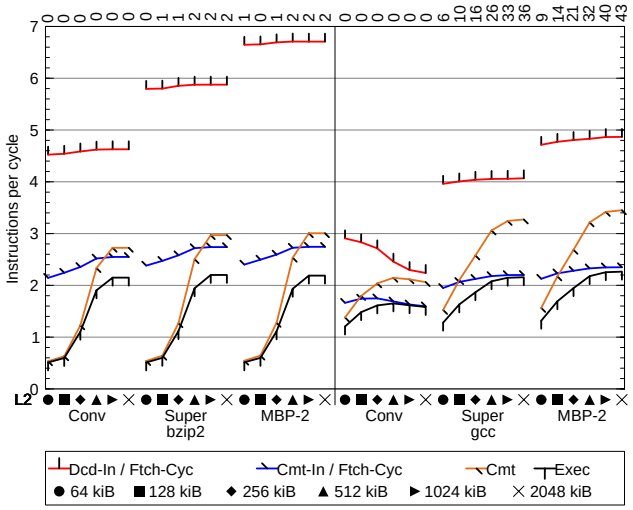

Figure 3. Performance on different L2 cache sizes of systems running gcc, perl, and TEX. Cache size given in $s = \log_2$ number of sets. L2 size is 2^{s+9} bytes.

be thought of as the n [possibly empty] entries at the head of the reorder buffer.) As with (b) the height of the lowest bar indicates IPC. The average number of instructions in the reorder buffer when a correct-path instruction arrives is plotted in the rectangles along the top. Segment **Co** (commit) indicate instructions that commit and **Em** (empty) indicate empty commit slots (as when there are less than n instructions in the reorder buffer), and segment **PS** indicate commit slots filled with instructions that are not yet ripe.

Segment **IL** (for available ILP limit delay) shows commit slots that are either filled with an instruction that uses a value produced by another instruction completing in the same cycle, or the instructions following that instruction. For example, suppose the reorder buffer in a 4-way system held the following four instructions in some cycle:

```
add r1,  r22, r33  ! Co:
sub r44, r1,  r55  ! IL:
xor r66, r77, r88  ! IL:
                   ! Em:
```

The `add` completes and commits in the cycle, but `sub` waits because of the data dependency, it is counted as an ILP delay. Note that all instructions in commit slots following the `sub` are also counted as having ILP delays, regardless of their disposition.

Segment **IM** (in memory) shows load blocking, that is, an instruction that cannot commit because it's an unfinished load, or an instruction in a commit slot blocked by such a load. Also included are instructions in other functional units, but these contribute only a tiny amount.

The performance gain due to increased fetch rate on the base configuration is realized primarily by an increase in commit rate as the instruction window is enlarged, as can be seen from the **IL** segments. The increase in reorder buffer occupancy of the well-performing benchmarks serves a useful purpose; for bzip2 and mcf it is a symptom of insufficient ILP.

Less of a factor is a reduction in the number of empty commit slots and reduction in miss latency impact. Empty commit slots are more of a problem in gcc, perl, and TEX because of their heavy use of indirect branches and instruction cache misses.

Disappointing perhaps is the small drop in blocking due to load latency. On average load instructions start over 20 reorder buffer entries higher up on order-2 MBP systems, providing at least a 2.5 cycle head start over the conventional system. That's only a small part of a 10-cycle L2 hit latency in the base system. Also, because of the faster order-2 MBP commit rate loads do not have that much more time before reaching the ROB head.

Address prediction is one way of getting loads to start earlier; load instructions on the base configuration on average moved down 30 entries from their time of arrival to when their source operands were ready; address prediction would provide another $3\frac{3}{4}$ cycles, or more if squashed loads are included. Reinman, in describing his supply mechanism, notes that predicted fetch address can be used by an address predictor to predict loads (before the load instructions are even fetched) [14].

6.2. BAC Size

The impact of varying the size of the BAC (and

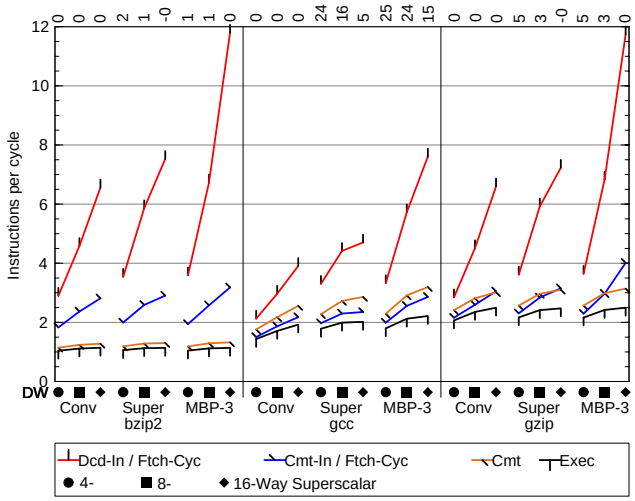

Figure 4. Performance with different decode widths of systems running bzip2, gcc, gzip, perl, and TEX.

FTB) sizes can be seen in Figure 2 where execution rates have been plotted for gcc, perl, and TEX for table sizes from 2^9 nodes to 2^{19} nodes. Data for the base configuration of the conventional system is plotted several times for reference. At the base size of 2^{13} nodes MBP-3 suffered because it had fewer BAC entries. At 2^{17} nodes it finally outpaces MBP-2, but by a small margin. This is to be expected since at a basic block size of 5 two blocks are sufficient to feed the eight instruction wide decode width.

At smaller table sizes the advanced systems suffer a slowdown over the conventional system, which has an unaffected next-line predictor. Interestingly MBP-2 and superblock predictors perform about the same despite the fact that MBP-2 must be suffering more misses.

6.3. Level-2 Cache Size

The impact of level-2 cache size is shown in Figure 3 where execution rates for two representative benchmarks, bzip2 and gcc, are shown. The advanced fetch mechanisms do little to reduce load latency and so at smaller cache sizes they have a low impact on performance. The small amount of available ILP in bzip2 limits the advanced fetch mechanisms to only a tiny performance gain at larger cache sizes. In contrast gzip2 at the largest cache sizes is fetch limited, as can be seen in the small gap between effective fetch rate and execution rate. (A small gap indicates that the ROB is rarely full.)

The gap between commit rate and execution rate is determined by how frequently CTIs are mispredicted and their position in the reorder buffer when resolved. The gap is small at small cache sizes for gzip2, indicating that CTIs resolve far enough from the ROB head to fetch, decode, and schedule correct path instructions before the CTI reaches the head.

6.4. Decode Width

On the eight-way advanced fetch systems examined performance was limited by prediction accuracy or available ILP. Moving to wider systems would do nothing to help prediction accuracy and would only marginally increase the window size (since branch prediction limited window size in eight-way systems). Performance improvement would come primarily from those portions of the code with a large amount of ILP and good branch prediction, such portions exist but occur to rarely to have much of a performance impact.

Experiments were run using superblock, order-3 MBP, and the conventional systems in 4-way, 8-way, and 16-way superscalar configurations. (In another set of 16-way experiments order 1,2, and 4 were also simulated.) The limited performance impact can be seen in Figure 4.

On wider machines the conventional system can also fetch more instructions per cycle, and so the relative advantage of the advanced fetch mechanisms *diminishes*. This is perhaps ironic since multiple branch prediction is seen as necessary for super-wide machines and yet provides the most relative benefit for narrower machines where it makes most efficient use of the limited number of fetch slots.

Figure 5 (*a*) shows the performance of MBP systems from order 1 to 4 on 16-way systems for the large configuration parameters. In (*b*) fetch slot usage is shown. The results show that for 16-way systems order-3 is best, on 8-way systems order-2 worked best. The superblock predictor averages only 3% speedup, outperforming MBP-1 which offers no speedup, but not doing nearly as well as MBP-2 to 4.

7. Related Work

7.1. Multiple Branch Predictors

The multiple branch predictor of Yeh, Marr, and Patt [18] was an early method of supplying instructions spanning multiple basic blocks to a superscalar processor. Their analysis focused on branch predictor performance and the prediction of indirect jumps was not considered. (Indirect jumps are handled by a single-block fetch mechanism in an earlier work [17].) The multiple branch predictor of Dutta and Franklin [3] also stores blocks reachable from an address, which they call a *tree-like subgraph*, but their predictor makes greater use of local history. Neither Yeh *et al* nor Dutta and Franklin simulate equal-cost systems to determine optimal tree depth nor do they provide performance

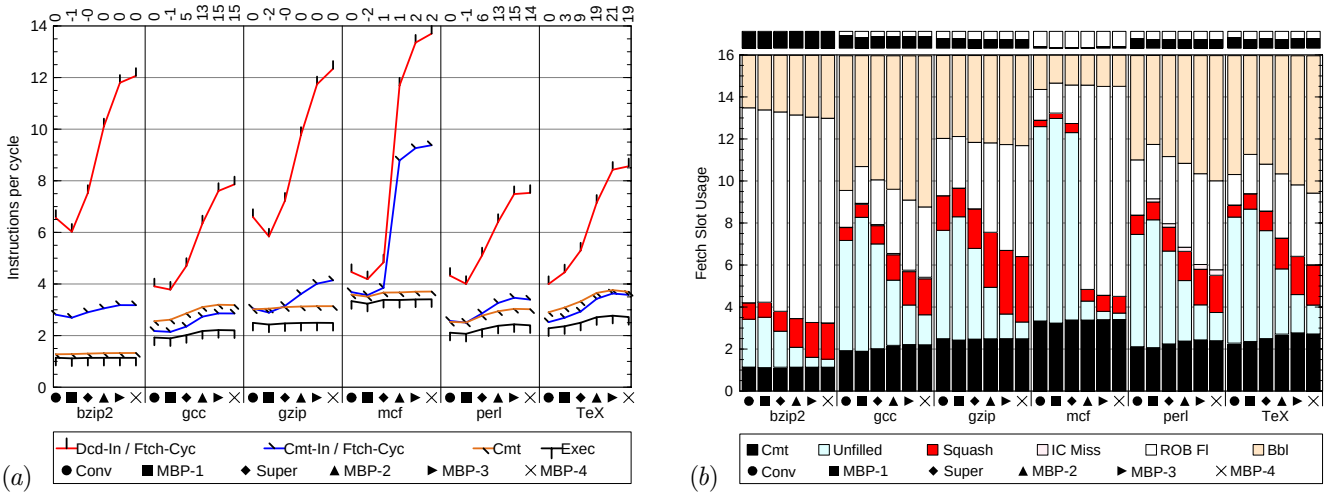

Figure 5. Performance on sixteen way machines.

numbers that accounts for the rate at which instructions can be executed.

Patel, Friendly, and Patt analyze several multiple branch predictors for a trace cache. The predictions are correlated with the start address of a trace: a PHT is indexed gshare-style, with the exclusive or of branch history and the trace start address. Several variations on what the PHT stores were analyzed, the best in the cost-equalized comparisons used the fewest counters, a counter for branches at distance 1, 2, 3, etc. (Rather than a tree of counters for all reachable blocks.)

Conte, Menezes, Mills, and Patel [2] analyze various dual-port instruction cache configurations; they find that the most flexible, using what they call a *collapsing buffer*, yields significant performance improvement. Such flexible multi-ported caches are assumed for the multiple branch prediction schemes discussed here.

Reinman, Calder, and Austin [13,14] describe an instruction fetch mechanism, the *Fetch Target Queue*, which is similar to an order-1 MBP, except that the single BAC node per entry stores information on several contiguous basic blocks, separated by highly biased not-taken branches. They show roughly 20% speedup over Yeh *et al Basic Block Target Buffer* [17] for small table sizes on 8-way systems, the speedup shrinks to a few percent when larger tables are used, indicating that the advantage is in compact table size. Similar results are obtained here over the order-1 Y-MBP which is similar to a BBTB.

Like the MBP as implemented here, Reinman *et al*'s FTQ is decoupled, allowing the predictor to run ahead of the fetch mechanism. They use the queued predictions to prefetch the cache.

7.2. Path-Based Trace Predictors

The path-based next trace predictor of Jacobson, Rotenberg, and Smith is, in effect, a multiple branch and indirect jump predictor [5]. A *trace*, in this context, is a segment of the dynamic instruction sequence that has been encountered and cached for later use. Traces are identified by a *trace ID*, the address of the first instruction and the outcome of contained branches. (An indirect jump must be the last instruction in a trace.) Traces are predicted by using a hash of the last several trace IDs to index a table that returns a predicted trace ID. Thus, both branch directions and indirect jump targets are being correlated with path history. The prediction accuracies attained, which also use variable history length and saved histories for calls, are very high [5].

Perhaps one disadvantage of this next-trace prediction mechanism is in the size of the history table needed. Each entry must store a trace ID, which must contain a substantial portion of an instruction address. A BAC also stores instruction addresses, however the number of times an address might appear in the table depends on the tree depth, which is small, whereas in a next-trace predictor the number of places a trace ID appears depends on the number of encountered paths which lead to it, which can be large. Multiple branch predictors predict only branch direction, and so can use compact pattern history tables. A JTB can be added, as in the systems analyzed here, without having to shrink the number of entries in the PHT.

Trace caches can also use multiple branch predictors [11], and there are many differences between the two unrelated to prediction. For example, systems using trace caches can have simpler instruction caches and decode units.

8. Conclusions

Multiple branch prediction mechanisms can attain higher performance than conventional instruction supply mechanisms on dynamically scheduled superscalar processors. The higher fetch rates presented in [18] lead to improved performance rather than running up against an ILP limit. Further, newer, more accurate branch prediction techniques can be used without sacrificing the ability to predict multiple branches.

The superblock predictor of Reinman *et al* is intended to get some of the benefit of a multiple branch predictor without the complexity of multi-read-ported prediction tables. Results here show that its performance is roughly half way between a single block predictor (MBP-1) and an order-2 multiple branch predictor.

An obvious tradeoff in tree type multiple branch predictors is the height of the tree (MBP order) v. the size of the forest (number of BAC entries). Results here show that the best size, when BAC storage is limited, is enough to fill the decode slots (decode width divided by basic block size), as one might guess. With a large BAC higher order yields only tiny improvements.

With the advanced fetch mechanisms, some of the benchmarks tested are limited by prediction accuracy, others by available ILP.

9. Acknowledgment

This work was supported in part by the National Science Foundation under Award No. CCR-0105478

10. References

[1] Brad Calder and Dirk Grunwald, "Next cache line and set prediction," in *Proceedings of the International Symposium on Computer Architecture,* June 1995, pp. 287–296.

[2] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the International Symposium on Computer Architecture,* June 1995, pp. 333–344.

[3] Simonjit Dutta and Manoj Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," in *International Symposium on Microarchitecture,* December 1995, pp. 258–263.

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

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

[6] William Johnson, "Superscalar microprocessor design," Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[7] 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.

[8] Scott McFarling, "Cache replacement with dynamic exclusion," in *Proceedings of the International Symposium on Computer Architecture,* May 1992, pp. 191–200.

[9] 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.

[10] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," *ACM Computer Architecture News,* vol. 20, pp. 76-84, October 1992.

[11] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yale N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers,* vol. 48, no. 2, February 1999.

[12] Chris H. Perleberg and Alan Jay Smith, "Branch target buffer design and optimization," *IEEE Transactions on Computers,* vol. 42, no. 4, pp. 396-412, April 1993.

[13] Glenn Reinman, Todd Austin, and Brad Calder, "A scalable front-end architecture for fast instruction delivery," in *Proceedings of the International Symposium on Computer Architecture,* May 1999, pp. 234–245.

[14] Glenn Reinman, Brad Calder, and Todd Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers,* vol. 50, no. 4, pp. 338-355, April 2001.

[15] David W. Wall, "Limits of Instruction-Level Parallelism," DEC WRL Technical Report, Nov. 1993, WRL–93/6.

[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 and Yale N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," International Symposium on Microarchitecture, December 1992, pp. 129–139.

[18] 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.