Combining Hyperblocks and Exit Prediction to Increase Front-End Bandwidth and Performance

Department of Computer Sciences Technical Report TR-02-41

Nitya Ranganathan Ramadass Nagarajan Daniel Jiménez*
Doug Burger Stephen W. Keckler Calvin Lin

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

*Department of Computer Science
Rutgers State University

Abstract

As processor issue widths increase, high-bandwidth parallel fetches across multiple branches will be necessary to prevent Flynn's bottleneck. In this paper, we show that by combining hyperblocks and exit predictors, high instruction bandwidths can be achieved in a streamlined, simple design that requires only a third of the predictions of a conventional predictor. Exit predictors identify the first branch in a hyperblock that will be taken with a single prediction, implicitly predicting all previous branches in the hyperblock as not taken. Hyperblock-based exit predictors reduce the number of predictions by 72%, while maintaining an average prediction accuracy of 94.7% (compared to 95.9% for the best conventional predictor we measured, a PAs-gshare *predictor). With this accuracy, a 250Kbit exit predictor achieves a mean 3.9 misses per thousand instructions (MPKI), fetching 91 useful straight-line instructions per prediction, on average. Since so many instructions are fetched with a single simple prediction, these predictors can be used for high frequencies, run-ahead prediction, initial prediction in multi-stage schemes, instruction prefetching, or power reduction. We conclude by showing that hyperblock exit predictors achieve higher accuracy than other high-bandwidth predictors at the same fetch bandwidth, and that these predictors can be used to achieve high ILP on advanced architectures such as Grid Processors.*

1 Introduction

Accurate branch prediction is essential for high performance in current-generation machines [33]. As clock rate improvements begin to reach hard limits [34, 10], the resulting drive for improvements in instruction-level parallelism (ILP) will demand more from branch prediction technology. Moreover, to achieve significant increases in ILP, large improvements in front-end bandwidth in general will be needed. Conventional front ends will be challenged to provide that bandwidth in future clustered solutions, since they require that all branches be routed through a centralized predictor. In this paper, we propose a series of schemes that couples aggressive predication to a range of new exit predictors, permitting large sequential regions of code, which may be physically distributed, to be predicted with a single predictor access.

Instruction predication converts control dependences to data dependences by merging multiple control flows into a single control flow, guarding instructions in the merged flows with predicates. Predication has been shown to improve branch prediction characteristics by removing hard-to-predict branches from the static schedule. A hyperblock is a region of predicated code that has a single entry point and no internal control flow, but potentially many branches that can exit the hyperblock.

Exit predictors were first proposed by Pnevmatikatos *et al.* [22] in the context of MultiScalar processors, and subsequently refined by Jacobson *et al.* [12]. Exit predictors perform only one prediction for all branches in a region of code (such as a MultiScalar task), predicting the first branch that will leave that region. By predicting only the branch that will exit a hyperblock, our exit predictor makes only one prediction per hyperblock, which may contain from tens to hundreds of instructions, with potentially tens of exit branches. Such a scheme, coupled with a BTB, can reduce the number of explicit predictions that need to be made by up to an order of magnitude. Such a scheme has the following compelling characteristics:

- *Ultra-high instruction fetch bandwidth*: For each predicted branch, more sequential instructions can be fetched before taking action on the next predicted branch (an average of 91).

- *Competitive prediction accuracy*: Since fewer branches are predicted, there are fewer conflicts in a particular size predictor, as only one of the branches in each hyperblock actually gets predicted; all others are implicitly predicted not taken. These exit predictors show accuracies superior to previously proposed predictors of equivalent instruction bandwidths.

- *Capability for distribution*: Since the exit predictor only needs to predict the sequence number of a branch in a hyperblock, it does not need to see the branch itself to make the subsequent prediction. Thus, instruction fetch can be heavily distributed in a clustered architecture; a necessary capability for high-performance future architectures such as highly clustered VLIW processors or Grid Processors.

- *A simple, streamlined design*: Since so many instructions can be fetched with a single prediction, the time between required prediction points is increased. The front end can exploit this aspect by (1) eliminating the need for complex overriding predictors, such as in EV6 [16] and EV8 [28], (2) predicting multiple branches ahead, which suffer multi-porting speed penalties [29, 25], (3) employing a slower, but more accurate, predictor [15], or (4) reducing the number of predictions to save power.

In this paper, we show that hyperblock-based exit predictors reduce the number of predictions made by 72%. We sustain an average prediction accuracy of 94.7%. With this accuracy, a 250Kb exit predictor achieves a mean 3.9 misses per thousand instructions (MPKI), fetching 91 useful straight-line instructions per prediction, on average. We show that our exit predictor is superior to other multiple-branch predictors in terms of accuracy at a given fetch bandwidth. We illustrate how exit predictors could be used in a VLIW processor front-end. Furthermore, we show that tuned exit predictors can enable high IPCs on block-atomic architectures such as Grid Processors.

We describe hyperblock construction issues, previous work on predication, and related work on multiple branch predictors in Section 2. Section 3 describes exit predictors that we propose, both traditional ones and perceptron-based. In section 4, we present a design space evaluation of exit predictors and also compare the accuracy and bandwidth of exit predictors to some of the best branch predictors proposed earlier. Section 5 demonstrates how exit predictors could be used in two different architectures: a VLIW processor, and a Grid processor. Finally, we draw conclusions in Section 6.

2 Background and Related Work

Predication was first implemented as vector masks in the Cray-1 [27]. The predication of individual instructions was first proposed by Allen *et al.* in 1983 [1], extended by Hsu and Davidson in 1986 [11], and first implemented in the wide-issue Cydra 5 [24]. However, it was Mahlke *et al.* [19] who first developed the modern notion of a hyperblock, which is the static abstraction that we use in this paper.

2.1 Hyperblock Construction

A hyperblock is a sequence of instructions, some predicated, that has no internal transfers of control. Hyperblocks have a single point of entry and may have multiple points of exit. Any exit through a branch instruction (including predicated procedure calls) is called an *early exit*. If none of the early exits in a hyperblock is taken, the control flow through the hyperblock is completely sequential. All early exits lead to the beginning of a new hyperblock. A hyperblock can be constructed by *if-conversion* of control flow paths. *Tail duplication* and *loop peeling* are further transformations applied to the hyperblock.

Large hyperblocks provide several benefits. First, by predicating hard-to-predict branches, predictability can be improved by reducing pressure on the branch predictor. Second, if early exits are rare, the front end can fetch across fewer taken branches, simplifying the design of a high-frequency high-bandwidth fetch unit. Third, large hyperblocks present improved scheduling opportunities for the compiler. The two primary disadvantages of large hyperblocks are a reduction of effective bandwidth because of the large number of useless instructions, and an increased pressure on execution resources.

2.2 Interactions of Predication and Prediction

Much previous work has examined these hyperblock construction trade-offs in the context of VLIW processors. Most recently, Knies *et al.* examine the benefits of predication for predictability and performance in an IA-64 implementation [5]. Knies *et al.* show that while *if-conversion* can potentially remove up to 29% of mispredictions on an Itanium processor (for SPECint 2000), predication-improved prediction is unlikely to cause significant performance gains. Our work differs in that our exit predictor/hyperblock combination has many fewer predictions to make.

In the mid-1990's, a series of papers examined the interaction of predication and branch prediction [18, 36, 4], but none of these considered branch predictors that exploited the specific control behavior of the hyperblock to improve front-end design. Mahlke *et al.* showed that if-conversion could reduce 27% of branches and 56% of mispredictions across their benchmark suite [18]. Tyson compared different predictor and predication schemes, showing up to 30% misprediction rate reductions for a number of predictors [36]. Chang *et al.* also studied the interactions between predication and speculation, showing that predicating hard-to-predict branches only can improve prediction accuracy for some benchmarks [4].

Other related work includes Simon *et al.*, which incorporates a deterministic order of predicates into branch predictor histories to improve prediction rates [32]. Havanki *et al.* studied the formation of non-linear regions called Treegions that could take advantage of predication [9]. August *et al.* studied the interaction between predication and other speculation [2]. Finally, Pnevmatikatos and Sohi [23] examined guarded (predicated) instructions in a dynamically scheduled superscalar microarchitecture concluding that full guarding can reduce mispredictions per thousand instructions to just under 4, but with a significant overhead of 33% more instructions fetched. Our work, different from the above, exploits the structure of a hyperblock to provide a long sequence of straight-line instructions with a single prediction.

2.3 Multiple Branch Predictors

Several predictors in the past have been designed to provide more than one basic block of instructions. Some of the schemes achieve this goal by predicting multiple branches in a single cycle (multi-branch predictors). Others make a single prediction for a few basic blocks of instructions (region predictors).

Yeh *et al.* [38], Seznec *et al.* [29] and Conte *et al.* [6] studied *multi-branch predictors*, structures that can fetch branch targets for multiple branches simultaneously. These predictors typically predict 2 or 3 targets at a time. The front-end design becomes more complex in these schemes compared to that in an exit predictor, since multiple targets have to be coalesced and sorted to fetch the correct instruction stream. Our work differs in that we predict only one target for a large, predicated region, and so we need to fetch only a sequential stream of instructions. In [28] and [37], multiple branches are predicted by maintaining as many saturating counters as the fetch bandwidth required, in each entry of the prediction table. We believe that hyperblock-based predictors do not require such a scheme since there are very few branches in the predicated code.

Region predictors attempt to predict, for a region of code, which branch will exit the region first. These schemes, including those proposed in this paper, attempt to provide a higher fetch bandwidth with fewer predictions. These predictors were first proposed for MultiScalar processors [22] and were called Control Flow Prediction Buffers (CFPBs). CFPBs typically limit the number of region exits to two, and CFPBs would predict one of the targets as the region's successor. Higher instruction bandwidth was observed, but prediction accuracies were expected to decline as the number of exits from a region increase. The authors also mention that coupling hyperblocks with control-flow prediction has potential to "enhance dynamic ILP". In another scheme [7], the authors predict one out of four possible paths in a tree-like subgraph of the control flow graph. The compiler has to construct such trees, and this becomes complicated for higher tree depths.

Subsequent work by Jacobson *et al.* describes more aggressive region-based predictors [12] that could predict up to four exits per region. Jacobson *et al.* proposed many of the techniques that we apply to hyperblocks in this paper, including local exit predictors, global exit predictors, path-based exit predictors, folding of exit histories, and hysteresis bits in the pattern history tables. They concluded that path-based task predictors were more effective than exit-based task predictors. The results in this paper indicate that when hyperblocks are used with exit predictors, the opposite is true: exit-based prediction outperforms path-based prediction. Another difference is that since the exit predictors in our work do not predict the target until the exit is computed, they require no block headers. The absence of block headers empowers hyperblock exit predictors to handle a much larger number of exits, limited only by hardware budget of the predictor.

Finally, we note that the goal of providing many straight-line instructions with a single prediction is similar to that of trace caches in dynamic superscalar environments [26]. Prediction of exits for traces is more challenging than for hyperblocks, since traces can change and thus use stale pattern history. Trace predictors [13] predict the address of the next trace directly without predicting the exit, hence they need to store the addresses in the predictor. Hyperblock exit predictors store only the exit numbers (which results in fewer storage bits) in the predictor and use a branch target buffer to generate the target prediction.

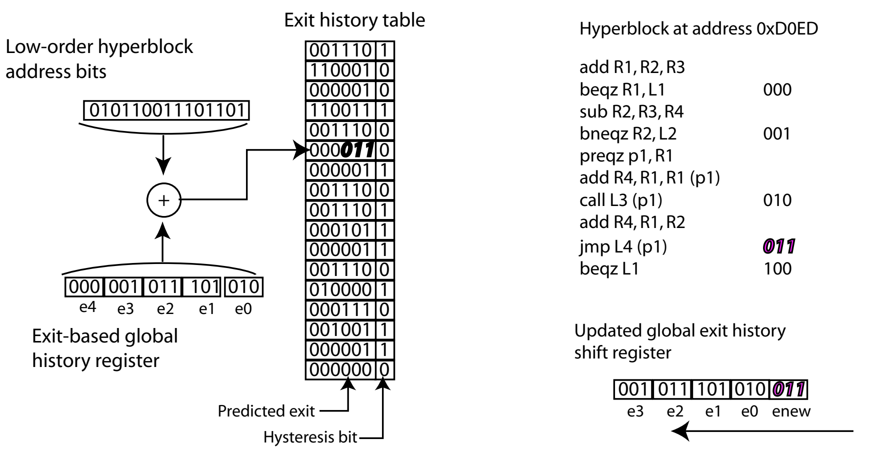
3 Exit Predictor Design

In this section we describe hyperblock-based exit predictors in detail. We design exit predictors based on both conventional schemes and neural techniques. Exit predictors based on conventional techniques have a simple and scalable design, and can make fast predictions, with accuracies close to some of the best traditional branch predictors. The perceptron-based exit predictor requires more time to make a single prediction, but provides higher accuracy than other high-bandwidth exit predictors.

3.1 Exit Predictors for Hyperblocks

Figure 1a shows a diagram of an exit predictor, a global predictor based on conventional branch predictor designs (the two-level approach proposed by Yeh and Patt [39]). The predictor produces a binary value that corresponds to the first branch in the hyperblock that will be taken. The first level of the table (the history register) differs from a traditional global history shift register by maintaining a history of exit numbers (the exit-based history) for each hyperblock, instead of per-branch taken/not-taken bits. Figure 1b shows an example hyperblock with five exits, each labeled with a three-bit binary identifier. Low-order bits of the hyperblock start address are XOR-ed with the exit history register, as shown in Figure 1a, which then index the exit history table.

The exit history table maintains, for a given pattern, the last exit number that was taken from a hyperblock.


Low-order hyperblock address bits
010110011101101
+
000 001 011 101 010
e4 e3 e2 e1 e0
Exit-based global history register
Exit history table
Predicted exit
Hysteresis bit
Hyperblock at address 0xD0ED
Updated global exit history shift register
001 011 101 010 011
e3 e2 e1 e0 enew

001110	1
110001	0
000001	0
110011	1
001110	0
000**011**	0
000001	1
001110	0
001110	1
000101	1
000001	1
001110	0
010000	1
000111	0
001001	1
000001	1
000000	0

```
add R1, R2, R3
beqz R1, L1              000
sub R2, R3, R4
bneqz R2, L2             001
preqz p1, R1
add R4, R1, R1 (p1)
call L3 (p1)             010
add R4, R1, R2
jmp L4 (p1)              011
beqz L1                  100
```

(a) 2-level global exit predictor (b) Sample hyperblock and updated register

Figure 1: Exit predictor example

In our example, the exit history table also maintains one bit of hysteresis to prevent replacement of a frequently taken exit with an infrequently taken one. The number of branches in a block that can be predicted is limited by the number of bits provided in the exit history table (6 bits in our example). By contrast, the exit history register uses only a few lower order bits of the exit.

Because the global history register can be longer than `log`$_2$`(size of history table)`, we split the history into several parts and XOR them to produce the final index into the history table; this is called *folding* of the history [12]. Before performing the XOR, each of these parts is shifted left by a few bits, if necessary, to prevent exits from lining up, resulting in cancellations that cause heavy conflicts in the exit history table.

After the exit is predicted, we use both the predicted exit and the branch address to determine the target. Lower-order hyperblock address bits are concatenated with lower order bits from the predicted exit and used as an index to a branch target buffer, the BTB (not shown in the figure). The BTB gives the target address of the branch i.e., the address of the next hyperblock.

Updates to the predictor are handled by replacing the exit in the history table entry (taking hysteresis into account) with the actual taken exit if the prediction was incorrect. The predictors shift in a few lower order bits of the exit into the first level exit history register. In the code example shown, the predicted exit is 3 (011), which is shifted onto the exit history shift register, as shown in Figure 1b.

This predictor organization has the following characteristics:

- A prediction that the n^{th} branch will be taken implies that branches 1 through $n-1$ will be not taken. Thus multiple implicit predictions are made with a single prediction, putting less pressure on the predictor state than a traditional per-branch predictor.

- Since the prediction generates the target of the hyperblock, the subsequent prediction can be started as soon as the previous prediction is complete and the history register is speculatively updated. Predictions can thus run ahead of the instruction fetch, similar to the Fetch Target Buffer [25]. Only one prediction is made per hyperblock, which facilitates high-bandwidth fetching for wide-issue machines.

- A hyperblock exit predictor requires no structures beyond what is needed by a conventional predictor, and it requires no ISA changes (assuming that predication is supported). The only restrictions are that (1) hyperblocks do not have more exits than are supported in the exit history table, and that (2) every hyperblock ends on a taken branch, for example, a "dummy jump" to the subsequent instruction. The second restriction allows the predictor to determine the end of the block with no ISA support.

3.2 Hyperblock Exit Predictor Design Space

The design space for the 2-level exit predictors is huge. Parameters include the length of the exits, the length of the history register, the amount of folding and shifting to get the index into the exit history table, the size of the first and second level tables, and the amount of hysteresis in the exit history table. As described later, we also explore tournament-style hybrid predictors that use a choice predictor to select from the result of two component predictors. We find that tournament predictors are consistently more accurate than individual 2-level exit predictors. Of course, the design space for tournament predictors is even more complex.

We reduced the search space by exploring some parameters individually while keeping the other parameters fixed. For example, the best values for the number of bits in the hysteresis and in the index to the local history table were determined and fixed. We found these values to be largely independent of the other parameters. We then used these results to explore other factors, such as the types of predictors and number and sizes of the exits. Section 4 gives a description of several predictors that we explored and their accuracies.

3.3 A Perceptron-Based Exit Predictor

We evaluate a *perceptron exit predictor* that uses neural learning [3] to classify patterns of local and global history as leading to one of several likely exit numbers. Perceptrons have been successfully used to predict branch directions [15]. The perceptron exit predictor, takes more time to make a single exit prediction, but is more accurate than the exit predictors previously described.

Perceptrons are vectors of small integer weights. The *excitation* of a perceptron is the dot-product of the weights and a vector of bipolar inputs. For the exit predictor, we use a combination of global and local history as inputs to an ordered set of a few perceptrons, called a *multi-perceptron*. Each perceptron in a multi-perceptron is associated with an exit number. To make a prediction, the hyperblock address is used to select a multi-perceptron from a table. The perceptron excitations are computed, and the exit number corresponding to the perceptron with the highest excitation is used as the prediction.

To update the predictor, we use the global and local history patterns to adjust the weights of each perceptron so that the perceptron corresponding to the correct exit number will yield a high excitation on that pattern and all other exits will yield a low excitation on that pattern. If the correct exit number is not associated with any perceptron, then it replaces the least-recently-used (LRU) exit number.

The dimensions in the design space for a perceptron exit predictor are global and local history length, number of bits for the weights, and number of low-order bits of exits to keep in the histories. We organize the multi-perceptrons as 8 banks of 48 perceptrons each, so that an access to the table of multi-perceptrons reads 8 perceptrons in parallel. The perceptron excitation computation is implemented similarly to a high-speed mul-

bench	amm	art	bzi	com	equ	gap	go	gzi	hyd	ijp	m88	mcf	mgr	par	swi	tom	tur	two	vor
# skipped (in billions)	3	1	1	6	4	1	0.5	7	1	0.1	1	1	1	1	1	1	1	1	1
# simulated (in billions)	1	1	1	1	1	1	0.1	1	1	0.1	1	1	1	1	1	1	1	1	1
# branches (in millions)	—	—	71.5	99.5	—	80.7	7.6	67.3	—	3.4	67.7	66.1	—	86.2	—	—	—	69.5	114.1

Table 1: Number of instructions and branches in benchmark simulations

tiplier, using a Wallace tree parallel adder to find the vector sum [15]. The update to the perceptron weights is fast [15] and off the critical path for making a prediction, so its delay component is less significant. Using CACTI 3.0 [31] and [8] we estimate the upper bound for the predictor delay to be 6.5 clock cycles.

Previous neural control-flow predictors are capable only of predicting whether a conditional branch is taken or not. This perceptron exit predictor generalizes this capability to predict one out of several possible outcomes. While we explore this capability for exit prediction, we note that this design is the first microarchitectural neural predictor design that can be applied to other general prediction problems, such as indirect branch prediction and value prediction.

4 Exit Predictor Evaluation

In this section we evaluate the exit predictors presented previously: exit predictors based on conventional techniques and perceptron exit predictors.

We use the Trimaran research compiler [35] which targets the HPL Play-doh ISA [**?**]. Trimaran supports profile-driven optimizations and predication. Our benchmark suite consists of 22 benchmarks: nine drawn from SPEC95 (*compress, go, ijpeg, m88ksim*,*hydro2d*, *mgrid*, *swim*, *tomcatv*, and *turb3d*), ten from SPEC2000 (*bzip2*, *gap*, *gzip*, *mcf*, *parser*, *twolf*, and *vortex*, *ammp*, *art*, *equake*) and three from the Mediabench [17] suite (*adpcm*, *dct*, and *mpeg2*). We do not present other benchmarks from the SPEC suites because the Trimaran compilations did not succeed with our configurations. We consider only the 11 SPECINT benchmarks for all the prediction accuracy results.

In Table 1 we list the details of the program phases we simulated. The first row in the table shows the number of instructions we skip before simulating, determined using the length of initialization given by the Sim-Point tool [30]. The second row shows the number of instructions we actually simulate. We simulated *adpcm* and *dct* to completion, and *mpeg* to completion after skipping 1B instructions. Because of compilation and simulator problems, we simulated only 100M instructions for *ijpeg* and *go*. Also we were forced to simulate *gap* and *vortex* during their long initialization phases, which require 123.5 and 29 billion instructions, respectively. The last row of Table 1 shows the number of dynamic branches encountered in the simulation window for each integer benchmark.

For measuring exit prediction accuracies, we used a custom trace simulator that takes branch addresses and targets from the Trimaran emulator for the phases we simulated, and compares exits and targets from the traces with the predictions generated by the simulator.

4.1 Benchmark Characterization

In Table 2, we show the average hyperblock characteristics for each benchmark, and means over both integer and the complete suite of benchmarks. The hyperblocks were generated with the default Trimaran hyperblock formation parameters. The static sizes (second column) of the average hyperblock range from 22 instructions (*parser*) to 336 instructions (*swim*). The dynamic sizes shown in column two of the table represent the number of useful instructions actually executed in each hyperblock, on the average. This number does not include instructions with false predicates or instructions in a hyperblock after the taken exit. Thus the dynamic sizes represent the average number of useful instructions that are fetched with each correct exit prediction, since only one prediction needs to be made per hyperblock. For all benchmarks, the predictor would thus need to be accessed at most every other cycle on a 16-wide machine, since the average number of useful, straight-line instructions typically exceeds 16. For many benchmarks the predictor would have ample opportunity to run ahead. For instance, for *ammp*, a prediction would need to be made only every 7 cycles, on average, on a 16-wide issue machine.

Benchmark	Static Hyperblock	Dynamic Hyperblock	Mispredicated Instructions	# Exits	# Exits Skipped	Predictions Eliminated
adpcm	51.1	30.7	5.0	6.7	2.4	70.6%
ammp	132.1	121.6	6.5	6.4	4.2	80.6%
art	83.3	80.5	2.6	6.3	5.1	83.7%
bzip2	68.3	55.9	3.2	6.4	3.2	76.3%
compress	24.7	21.6	0.5	3.1	1.3	57.3%
dct	188.0	172.2	0.0	2.5	1.3	57.3%
equake	41.9	32.6	0.0	1.9	0.8	45.9%
gap	35.9	27.3	0.7	3.7	1.4	58.4%
go	29.2	23.7	0.9	4.9	1.3	55.8%
gzip	66.8	37.2	11.3	5.3	1.7	62.9%
hydro2d	213.1	201.8	8.0	8.3	6.9	87.4%
ijpeg	80.0	73.3	3.8	3.2	1.7	63.5%
m88ksim	48.2	40.1	3.6	3.8	2.2	69.0%
mcf	39.7	29.3	5.4	2.8	1.1	53.5%
mgrid	180.2	180.1	0.0	1.1	0.0	4.1%
mpeg2	102.5	89.5	8.1	4.3	2.3	70.1%
parser	21.8	16.4	2.1	2.6	0.9	47.6%
swim	336.0	334.6	1.2	6.7	5.7	85.0%
tomcatv	211.2	206.7	3.1	5.4	4.3	81.2%
turb3d	177.0	154.1	0.0	4.2	2.7	72.7%
twolf	61.5	48.9	4.1	4.5	2.7	73.1%
vortex	37.5	30.4	3.6	4.5	3.3	76.7%
MEAN	101.4	91.3	3.3	4.7	2.6	72.1%
MEAN-INT	46.7	36.7	3.5	4.4	1.9	65.6%

Table 2: Hyperblock and exit characteristics of benchmarks

The fourth column shows the average number of mispredicated instructions per block. The fifth column shows the average number of exits per hyperblock which ranges from 1.1 for *mgrid* to 8.3 for *hydro2d*. The sixth column shows the average number of not-taken branches before the block exit, which is exactly the number of additional branches implicitly predicted as *not taken* by each exit prediction. Finally, the seventh column is derived from the sixth, and shows the fraction of predictor accesses that are eliminated by using an exit predictor instead of a conventional branch predictor. On average, across the suite, 72% of the predictor accesses of a conventional predictor are eliminated (66% for integer benchmarks), which also reduces the need for hierarchical predictors [14], makes a larger, more accurate predictor tractable, and reduces power consumed.

4.2 Design Space Exploration

We evaluated our exit predictor (based on conventional predictor design) over a wide range of design choices. The three metrics we use in our evaluation are *exit accuracy*, which counts the fraction of predictions in which the exit predictor selects the hyperblock exit correctly, the *predictor accuracy*, which counts the fraction of all branches that are correctly predicted, and the *predictor efficiency*, which measures the number of mispredictions for each kilo-instructions executed (MPKI). The predictor efficiency directly correlates with performance lost due to mispredictions; the EV8 designers estimated an approximate 4% performance loss per MPKI.

The primary design choice for the exit predictor is selecting the type of the predictor: global (global level-1 and global level-2 tables), local (per-block level-1 and global level-2), tournament (local and global with a choice predictor), or path-based as in [12]. For each of these predictor organizations, other parameters include, a) length of the encoded exit, b) length of the exit history, and c) folding of history bits. We tuned all predictors over a wide range of these parameters.

Table 3 shows the mean misprediction rates for several exit predictor designs. We fixed the size of all predictors to be approximately 250 Kbits. This translates to a total of 32K entries in the second level table(s). Each such entry has a 6-bit exit and one bit of hysteresis.

Most of the columns in table 3 are labeled *Xm,n*, where X indicates the predictor type: G for global, L for local, P for path-based, and T for tournament. The encoded exit length in bits is denoted by *m* and history length by *n*. Tournament predictors indicated by T have the same values of *m* and *n* for both the components. When the number of history bits is greater than the number of entries in level-2, we do shifting and folding of the history as described earlier.

In the table, `best-exit` refers to our current best exit predictor which is a 250Kb local-global tournament predictor. In `best-exit` both the local and global components have a 14-bit history with the local predictor using 2-bit exits and the global using 3-bit exits. The choice predictor uses a 12-bit global history to index into a table of 3-bit saturating counters which choose one of local or global predictors. The `best-exit` tournament predictor was selected as the best among 300 tuned tournament predictors which varied in history lengths, exit-sizes and component predictor types (global, local, path-based and path-based as in [**?**]).

The column `spec-updt` is the same as `best-exit` with “simulated” delays and wrong path predictions

bench	L2,15	L4,15	L4,30	L4,15 -noxor	G2,15	G3,30	G4,15	G4,30	P4,30	T2,14	T3,14	best -exit	spec updt	perf -tour
bzip2	21.8	19.6	22.2	20.4	19.2	23.8	19.1	20.4	22	18.4	18.5	18.0	18.6	13.6
compr	19.2	18.7	19.0	18.6	16.3	16.5	16.4	16.9	16.1	15.5	15.9	15.2	15.3	10.8
gap	1.1	1.2	1.1	1.2	2.7	2.7	9.3	2.7	2.7	0.7	0.8	0.7	4.1	0.4
go	21.7	17.9	22.3	24.4	17.9	20.1	17.1	18.5	17.2	17.6	15.7	16.3	19.1	9.5
gzip	18.2	17.0	18.9	17.3	14.4	13.8	14.8	13.7	15.4	14.4	14.3	14.3	14.4	10.6
ijpeg	11.6	11.7	11.6	12.8	10.7	10.7	11.6	10.6	11.6	10.1	10.9	10.1	10.9	7.2
m88ksim	3.5	6.2	2.9	11.4	7.8	6.7	8.9	6.8	8.0	2.0	3.0	2.3	4.2	1.6
mcf	11.8	10.9	12.2	11.6	8.9	9.8	9.1	9.2	9.1	9.1	8.9	8.9	9.1	6.1
parser	5.7	6.6	5.8	8.4	6.2	6.4	7.3	7.9	6.6	4.8	5.3	4.8	5.9	3.0
twolf	25.7	22.3	26.5	25.8	18.9	20.1	20.7	19.5	19.5	17.5	16.4	17.1	18.9	12.2
vortex	0.8	0.6	0.6	2.2	0.9	0.7	0.6	0.6	0.7	0.3	0.2	0.2	0.7	0.2
MEAN	12.8	12.1	13.0	14.0	11.3	11.9	12.3	11.4	11.7	10.0	10.0	9.8	11.0	6.8

Table 3: Exit prediction accuracy

modeled in our trace-based simulator. We speculatively update the global and choice predictor history registers, and also do a delayed update with random delay for all the prediction structures. Finally, `perf-tour` gives an upper bound on the accuracy achievable by `best-exit`. It uses an "ideal" choice predictor, which always makes the correct choice if either one of global or local give the correct prediction.

We make the following observations from the table. Global predictors are typically superior to local predictors and folding longer histories tends to reduce accuracy for local exit predictors (*L4,15* vs.*L4,30*), but increases it for some global predictors(*G4,15* vs. *G4,30*). XOR-ing with the address bits to reduce aliasing shows a big difference in accuracy (*L4,15* vs. *L4,15-noxor*). Some benchmarks benefit from local histories with shorter exit encodings (such as *L2,15*), typically, but not always, those benchmarks with fewer exits per hyperblock (parser, m88ksim). Those with more hyperblock exits typically benefit from more bits per exit (*bzip2*, *gzip*, *vortex*). Path based predictors were in general worse than global predictors, but better than local. The tournament predictors outperform the component predictors and performed best when local and global component predictors were used with different exit encodings for local and global. We also observe that speculative updates causes less than 2% reduction in accuracy. Finally, we note that a large gap exists between `best-exit` and `perf-tour` for several benchmarks (*bzip2*, *compress*, *gap*, *go*, *twolf*), but future work will determine whether that gap exists due to probabilistic coincidence or an insufficiently effective choice predictor.

4.3 Comparison to Traditional Predictors

In this section we compare hyperblock-based exit predictors with some of the best predictors proposed previously. We make a comparison of both the accuracy and the bandwidth provided by exit predictors with other predictors.

4.3.1 Accuracy Comparison

Since exit predictors explicitly predict fewer branches than branch predictors do, exit accuracies are not directly comparable to prediction accuracies from conventional branch predictors. To compare them directly, we convert exit accuracy into branch accuracy by counting each implicitly predicted not-taken branch before a hyperblock exit as a correct prediction. For example, if the predictor correctly predicts exit 4, then we count four correct branch predictions. When an exit predictor mispredicts, two cases are possible. First, if the taken exit is earlier than the predicted exit, then we incur one misprediction (for incorrectly predicting the actual taken exit). Second, if the predicted exit is not taken, we use a second exit prediction (which is available at the same time as the first) so long as it indicates a later exit than the first incorrect prediction. If the second prediction is also incorrect, then we predict that the hyperblock will fall through, and that none of the conditional or predicated branches will be taken. Thus, in the worst case, the exit predictor can incur three mispredictions on a single hyperblock. For the second prediction, the tournament exit predictor gives the prediction from the component that was not chosen and the perceptron exit predictor gives the prediction from the perceptron with the second highest excitation.

With this metric, we compare the misprediction rates of the `best-exit` predictor with several other branch predictors in figure 2. We present results for some of the best conventional predictors we measured. All predictors were driven by the same instruction traces. Results are shown for the 11 SPECINT benchmarks we simulated. We graph the accuracies of a gshare predictor, that takes 256Kbits [20], a PAs-gshare hybrid (284Kb) [20], a perceptron branch predictor (256Kb) [15], the `best-exit` predictor (250Kb) described earlier and the perceptron exit predictor (264Kb). We tuned the conventional predictors aggressively, simulating all possible (address, history) pairs for a fixed size for the tables and choosing the configuration with the best overall mean for each predictor. For the PAs-gshare hybrid, we assumed “full gshare” for the gshare component (an equal number of history and address bits).

For the perceptron branch predictor, we used the global/local hybrid of the perceptron predictor as described in [15] and tuned for a 256 Kbits hardware budget. The best configuration for a 263 Kb perceptron exit predictor uses 48 multi-perceptrons, 51 bits of global history and 20 bits of local history kept in a table of 1024 local histories. Both global and local histories store the lowest three bits of exit numbers. The perceptron weights are 9-bit signed integers. Each multi-perceptron keeps 8 perceptrons, and each perceptron uses 3 bits to keep track of the LRU exit number.

Our results show that the exit predictors attain accuracies close to the best conventional branch predictor accuracies. The average misprediction rate of `best-exit` is 5.3% on the average. The best conventional branch predictors we measured were the PAs-gshare hybrid and the perceptron branch predictor. PAs-gshare incurs an average misprediction rate of 4.1% and perceptron incurs 3.7% across the 11 benchmarks. The tournament exit predictor performs almost as good as PAs-gshare on 3 benchmarks (*gap*, *m88ksim* and *vortex*), but worse on several others, notably *compress* and *go*. The results for the perceptron exit predictor show lower misprediction rates compared to the `best-exit` predictor but on the average (4.7 %), is higher than that of PAs-gshare.

The rightmost set of bars represent the mean MPKI (predictor efficiency), which are directly related to the misprediction rates, scaled only by block sizes. The exit predictor incurs 3.9 mispredictions per KI, the perceptron exit predictor incurs 3.5 MPKI, while PAs-gshare incurs 3 MPKI and the perceptron branch predictor incurs 2.7 MPKI.

Figure 3 shows how per-branch exit prediction accuracy of scaled `best-exit` compares to those of conventional predictors as the size of the level-2 tables is varied. The total size taken by all the level-2 table entries is plotted on the x-axis with the y-axis giving the mean misprediction rates. We compare three predictors: gshare, PAs-gshare and exit (tournament). At all hardware budgets, PAs-gshare has the best prediction accuracy, while

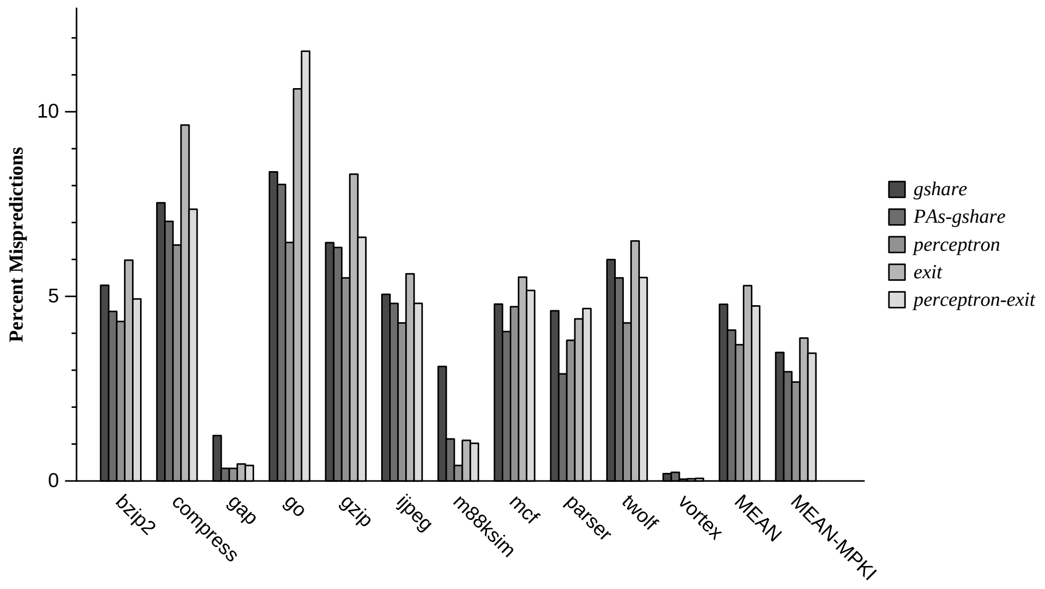


Figure 2: Individual Branch Predictor Misprediction Rates

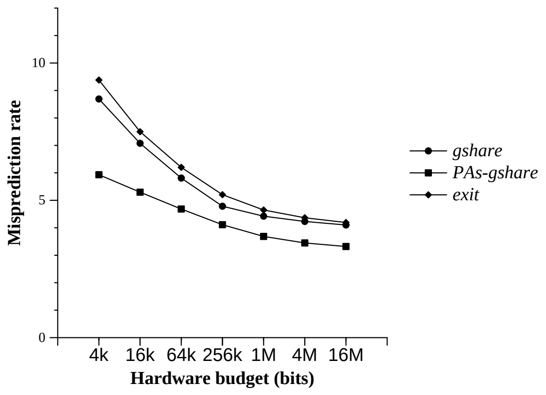


Figure 3: Predictor size vs. Misprediction Rate

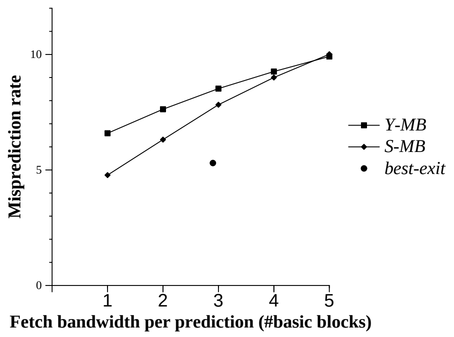


Figure 4: Fetch Bandwidth vs. Misprediction rate

the tournament exit predictor performs almost as well as gshare. The curves start flattening out when we reach large sizes, specifically around 1M bits for the exit predictor. We note that the exit predictors have almost one-fourth of the number of entries at a given capacity than the conventional predictors, since the size of a second level table entry is 7 bits (compared to 2 or 3 bits for the branch predictors). How much the exit length can be reduced without a significant loss in accuracy, or severe limitations on hyperblock formation, is a topic for future work.

4.3.2 Bandwidth Comparison

In Figure 4, we compare three predictors capable of delivering multiple basic blocks with a single prediction: the exit predictor `best-exit`, Yeh *et al.*'s [38] multiple branch predictor *Y-MB* and Seznec *et al.*'s [29] multiple block-ahead branch predictor *S-MB*. We use the Gshare predictor to model Seznec's block-ahead concept. The exit predictor uses 250Kb of state while the other two predictors use 256Kb. We do not model tree-like multiple branch predictors as described in [7] because that scheme cannot work with hyperblocks.

The x-axis in the graph shows the number of basic blocks fetched per prediction and the y-axis shows the misprediction rates. The single point at a bandwidth of almost 3 basic blocks gives the misprediction rate of `best-exit`, which is 5.3% on the average. Since the exit predictor makes just one prediction per hyperblock, it cannot be configured to predict a fixed number of basic blocks per prediction. As shown, the *Y-MB* predictor achieves accuracies close to the exit predictor when the bandwidth is 1 basic block, but is worse as the bandwidth increases. Further, it should be noted that this scheme requires an exponential number of ports (with respect to the number of basic blocks predicted) in the second level table. The *S-MB* gshare predictor performs better than the exit predictor when the number of blocks is 1, but becomes worse thereafter. Also, *S-MB* requires as many ports as the number of blocks to be fetched. In contrast, best-exit has only one port to each second level table keeping the design simple and scalable.

4.4 Summary

The results described in this section show the following. Careful tuning of the predictors are required for superior performance. For predicated hyperblocks, a tournament exit predictor achieves a high prediction accuracy, competitive with the best traditional predictors, while doing so at a very high bandwidth. A high-bandwidth instruction memory, such as a distributed I-cache, when coupled with exit predictors enable the design of aggressive front-ends demanded by high-ILP machines. The exit predictor offers several advantages: simple implementation like conventional 2-level predictors, run-ahead prediction, fast instruction prefetching, and power savings through reduced predictor accesses. The extra time available between predictions can also be used by slower predictors, such as the perceptron exit predictor to deliver higher prediction accuracy.

5 Performance Evaluation

In previous sections, we described exit predictors and showed how well they predicted control flow in typical programs. In this section, we measure their effect on overall performance. We first describe their application on the Grid Processor, a high-ILP core requiring a high-bandwidth instruction supply. We show that the fast front-end enabled by exit predictors, coupled with a scalable substrate, provides the highest demonstrated IPCs to-date. We then describe how exit predictors can be used in conventional VLIW processors, and compare their performance against traditional branch predictors.

For both the GPA and VLIW machines, we use code schedules from Trimaran, compiled for an 8-wide VLIW machine. We assume infinite register sets with no spills or restores. We model a memory hierarchy with 64KB instruction cache and 64KB data caches, and 2MB secondary cache, but no page translation tables (TLBs). We model speculative updates of branch predictors and recovery upon commits. We do not, however, model wrong path pollution of the predictors and caches. We assume a BTB with 4K entries for address prediction. Finally, we assume a perfect return address prediction.

5.1 Performance on GPA

Grid Processor Architectures (GPA) are an emerging family of technology-scalable architecture that use a fully decentralized core and a block-atomic model of execution [21]. The execution core is composed of an array of ALUs, each with several reservation stations that together provide a large distributed instruction window. Onto this array, large blocks of statically scheduled instructions are mapped and executed in dynamic dataflow fashion. To sustain a high instruction throughput, the distributed instruction window is backed by a fully partitioned register file and distributed caches.

The large instruction window of a GPA must be fed by an uninterrupted and high-bandwidth supply of instructions. Conventional predictors are a poor match for these distributed front ends. By predicting every branch, they require the entire instruction fetch stream to be funneled through a single centralized unit. At future process generations, the poor scalability of wire delays makes such centralized units—and instruction distribution from these units to the ALUs—prohibitively slow. Exit predictors avoid this limitation because, they do not need to see the actual branch itself to make a prediction, thus enabling the design of highly simplified and truly distributed front-end architectures.

In the GPA, the instruction cache is distributed into several banks, one bank per row of the grid. The predicted target address of the next hyperblock is broadcast to every I-cache bank. Each bank fetches and distributes instructions scheduled in its associated row. The distributed I-cache enabled by exit predictors thus provides a high-bandwidth supply of instructions to the execution core.

To measure the performance of exit predictors on the Grid Processor, we schedule the hyperblocks obtained from Trimaran using a custom instruction scheduler. We simulate an 8x8 array of ALUs, each with 128 reservation stations, providing an effective issue width of 64 instructions and a distributed instruction window of 8K instructions. The instruction cache is partitioned into eight 8KB, 2-way set associative banks with an access latency of 2 cycles. We assume a 0.5 cycle ALU-to-ALU communication delay and oracular load/store forwarding, in which loads are never held longer than necessary at the load/store queues. We assume that exit mispredictions incur a penalty of 10 cycles.

Figure 5 compares the performance, measured in instructions-per-clock, of our 250Kb `best-exit` predictor and the 263Kb perceptron exit predictor. We show results only for a subset of the benchmarks; other benchmarks exhibit similar results. For each benchmark, we show three bars; the left-most and the right-most bars represent realistic configurations of the `best-exit` predictor and the perceptron predictor. They are assumed to have a latency of 3 and 6 cycles respectively, between successive predictions. The middle bar represents the `best-exit` predictor but assumes a latency of 6 cycles. As shown in the figure, the GPA achieves an average

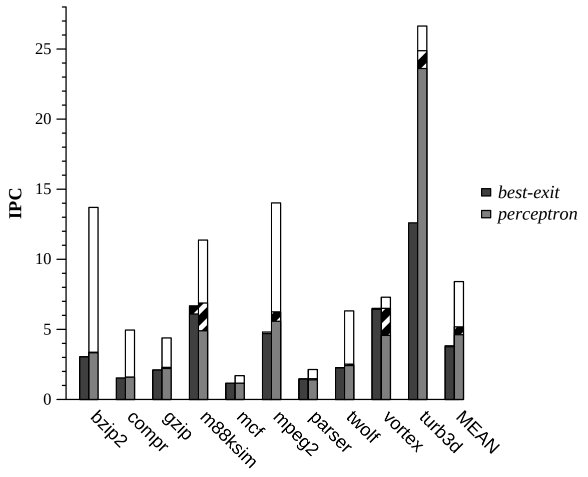


Figure 5: Exit predictors on GPA

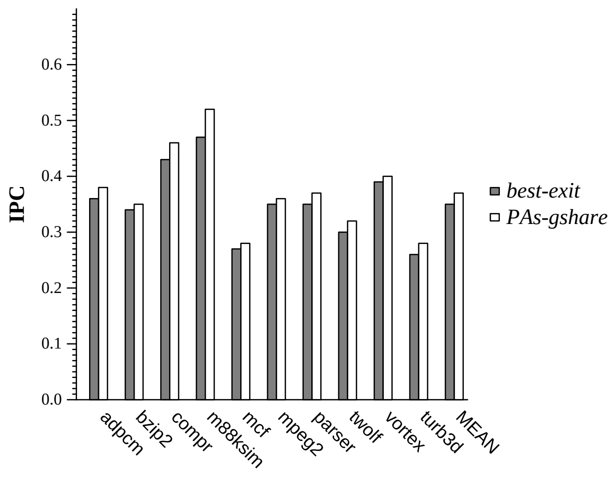


Figure 6: Exit predictors on VLIW machines

instruction throughput of over 4 for the `best-exit` and over 5, when the perceptron predictor is used.

On most benchmarks, the performance as seen from the first two bars is largely insensitive to the latency between successive predictions, when the `best-exit` predictor is used. Potential exists to exploit this latency-tolerance in one of two ways. First, the additional latency could be used by a slower but more accurate exit predictor, such as the perceptron predictor, resulting in improved performance. This behavior is evident in two benchmarks: *mpeg2* and *turb3d*. Second, a fast exit predictor such as the `best-exit` could run-ahead and predict instruction addresses to prefetch, thus improving I-cache performance. We leave the latter experiment for future work.

We notice that on some occasions (*m88ksim* and *vortex*), longer latencies between predictions can degrade performance significantly even with superior prediction accuracies. On such instances, perhaps the best prediction mechanism would be to use the fast `best-exit` predictor overridden by a slow perceptron, combining the advantages of high accuracy and high bandwidth prediction. We also leave this study for future work.

5.2 Exit Predictors for VLIW Architectures

While the previous discussion demonstrated the performance of the exit predictor on a specific architecture, this section shows how they can be used in conventional processors. Recall that by predicting which one of several branches will be taken, exit predictors implicitly predict the prior branches as not taken. We exploit this property to develop a branch predictor for conventional processors.

When the exit predictor is presented with the current fetch address (`fetchPC`), the predicted exit number and target address are saved in two registers. Successive instructions are fetched by adding a constant offset to `fetchPC`. As instructions are fetched and pre-decoded, every branch that is fetched increments a counter, `pred_ctr`. When `pred_ctr` reaches the saved(predicted) exit number, `fetchPC` is set to the predicted target address and further fetches are initiated from this address. Meanwhile, the next prediction is made with the new `fetchPC`. Similarly as branches are executed, a second counter, `exec_ctr`, is incremented. When a taken branch is encountered, the prediction tables are updated. A misprediction is flagged if the `exec_ctr` does not match the value in `pred_ctr`. Notice that this approach does not require any changes to the ISA. It only requires that the instruction stream be well demarcated into hyperblocks by appropriate branch instructions.

We implemented the above predictor based on the `best-exit` predictor for a VLIW machine, and compared its performance with the *PAs-gshare* predictor. For this evaluation, we simulated the code schedules obtained from Trimaran on a custom VLIW simulator. The VLIW machine that we compiled for and simulated had separate 2-way set associative, 3-cycle, 64KB caches for instruction and data, and a 2-way, 13-cycle, 2MB unified L2 cache. The arithmetic units were configured to have the same latencies as the Alpha 21264. We provided the PAs-gshare with 284Kbits of state, and the `best-exit` with 250 Kbits. Branch mispredictions incur a latency of 3 cycles on both the predictors.

Figure 6 shows the performance measured in IPC from using PAs-gshare and `best-exit`. The left bar shows the IPC when PAs-gshare is used to predict every branch, while the right bar shows the IPC when the exit predictor is used in place of the PAs-gshare. As shown, PAs-gshare performs better in every benchmark, achieving improvements of over 10% for *m88ksim*, and 5.5% on average. This result is not surprising, since the prediction accuracies as calculated in Section 4 were higher for the PAs-gshare predictor.

The small performance losses of `best-exit`, however, is coupled with significant reductions in the number of predictions, a potential power saving technique. Alternately, a run-ahead exit predictor can setup a pipeline of fetch addresses that can be used to either directly fetch from or prefetch into instruction memory. Thus with no changes to the ISA and only modest hardware additions, an effective high-bandwidth supply can be enabled which can potentially overcome the performance losses due to reduced accuracies. We leave the evaluation of these optimizations for future work.

6 Conclusions

As technology constraints force the move to wider-issue processors, significant demands will be placed on front-end architectures. In particular, it is becoming increasingly difficult to fetch a steady stream of instructions at modern clock rates. To meet these challenges, designers have already shown a willingness to sacrifice some accuracy for simpler implementations [28]. In this paper, we have presented exit predictors as enabling technologies for future high bandwidth fetch engines.

We have shown that by applying exit prediction to large, predicated hyperblocks, dramatic improvements in front-end bandwidth can be achieved at only small losses in accuracy compared to tuned modern predictors. Averaged over our benchmark suite, hyperblock exit predictors can eliminate 72% of the predictions, and fetch 91 useful straight-line instructions, on average, per prediction. Such high-bandwidth predictions creates the potential for further optimizations. The extra time allowed by exit predictors can be exploited to conserve power, to enable slower and more accurate cascaded predictors, or to build fast prefetching engines and run-ahead predictors. Furthermore, by predicting only the sequence number of the taken branch, exit predictors avoid the need to see the branches themselves to make a prediction, which enables the design of high-bandwidth distributed front-ends.

We found that the best hyperblock exit predictor configuration, based on conventional designs, was an EV6-like tournament predictor, in which the local and global histories used a different balance between exit length and number of exits in the history. This 250Kb exit predictor achieves a mean prediction accuracy of 94.7%, and incurs 3.9 MPKI on average across our integer benchmark suite. We showed that such accuracies, while competitive with traditional predictors, are significantly higher than other high-bandwidth predictors at equivalent fetch bandwidths. We evaluated a perceptron-based exit predictor that exploits the longer time available to make predictions and offers higher overall accuracy of 95.3% (3.5 MPKI).

We quantified the performance improvements offered by our tuned exit predictors on Grid Processor Architectures. The high bandwidth offered by the exit predictors coupled with a distributed execution core provides superior performance, sustaining mean throughputs of over 5 instructions per cycle. We also showed an application of exit predictors to conventional VLIW machines.

The potential remains for further improvements in accuracy. With much larger hyperblocks and careful predication, it is possible that we might be able to remove many hard-to-predict branches and predict the hyperblock exits more accurately. The interaction of hyperblock construction strategies and exit predictability is a promising direction for future work.

References

[1] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei W. Hwu. Integrated predicated and speculative execution in the impact epic architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–327, 1998.

[3] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[4] P. Chang, E. Hao, and Y. Patt. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[5] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 182–191, Dec. 2001.

[6] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 333–343, June 1995.

[7] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 258–263, Dec. 1995.

[8] Y. Hagihara, S. Inui, A. Yoshikawa, S. Nakazato, S. Iriki, R. Ikeda, Y. Shibue, T. Inaba, M. Kagamihara, and M. Yamashina. A 2.7ns 0.25um CMOS 54 $\times$ 54b multiplier. In *Proceedings of the IEEE International Solid-State Circuits Conference*, Feb. 1998.

[9] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 266–276, 1998.

[10] M. Hrishikesh, K. Farkas, N. P. Jouppi, D. Burger, S. W. Keckler, and P. Sivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[11] P.-T. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[12] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, Feb. 1997.

[13] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.

[14] D. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.

[15] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4), November 2002.

[16] R. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[18] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Hyllenhaal, D. M. Hallagher, and W. mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 217–227, 1994.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25st International Symposium on Microarchitecture*, pages 45–54, 1992.

[20] S. McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[21] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 40–51, 2001.

[22] D. Pnevmatikatos, M. Franklin, and G. S. Sohi. Control flow prediction for dynamic ilp processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993.

[23] D. Pnevmatikatos and G. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 120–129, 1994.

[24] B. Rau, D. Yen, W. Yen, and R. Towle. The cydra 5 departmental supercomputer. *IEEE Computer*, pages 12–35, Jan. 1989.

[25] G. Reinman, B. Calder, and T. Austin. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th International Symposium on Computer Architecture*, June 1999.

[26] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings on the 29th Symposium on Microarchitecture*, pages 24–34, 1996.

[27] R. Russel. The cray-1 computer system. *CACM*, 21:63–72, Jan. 1978.

[28] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 295–306, May 2002.

[29] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, 1996.

[30] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, Sept. 2001.

[31] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[32] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techonology*, Dec. 2001.

[33] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[34] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, May 2002.

[35] Trimaran : An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[36] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 196–206, 1994.

[37] S. Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 94–103, 1997.

[38] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 67–76, July 1993.

[39] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, 1994.