FLAP: Flow Look-Ahead Prefetcher

Rice University – ELEC 525 Final Report
Sapna Modi, Charles Tripp, Daniel Wu, KK Yu

Abstract – In this paper, we explore a method for improving memory latency hiding as well as increasing instructions per clock cycle in a processor. Our proposed architecture combines the techniques of branch prediction and load prefetching. If the processor regularly misses on a particular load, and if we can correctly predict the address of this load, then the prefetcher will mark the instruction for prefetching. Using the branch predictor to run ahead of the program instruction stream, the prefetcher will speculatively issue the load before the processor reaches it again. The main component we added to the baseline architecture is a load tracking table, which is located off the critical path. Using Simple Scalar, we simulated three SPEC2000 benchmarks and found that for applications with high cache miss rates, with an accurate address predictor, we can achieve large speedups.

I. INTRODUCTION

a. Motivation

Advances in modern processing techniques are creating a greater demand for data to be returned at low latencies, but as the gap between the performance of processors and memory subsystems widens, this is becoming difficult to do. For over a decade, architects and engineers have implemented enhancements such as out-of-order execution, VLIW compilers, expanded issue widths, highly-accurate branch prediction, and much more in order to increase microprocessor performance. However, these advantages can only be realized if the underlying memory system can provide data quickly enough to keep the processor busy. As main memory latencies increase to hundreds of processor cycles, memory accesses become extremely costly. For this reason, efficient prefetching techniques are necessary to help reduce main memory accesses that other methods cannot target. Figure 1[1] compares compute time against time spent waiting on memory; it is obvious that this problem cannot be ignored.

b. Hypothesis

We believe that we will be able to increase IPC and hide load-associated memory latency by predicting and prefetching loads far in advance of when the data is needed by the processor; this will be accomplished via the use of an accurate branch predictor to predict future program flow, and a load address predictor to issue expected loads in advance. Note that we are not trying to make the cache more efficient; we are simply attempting to hide as much memory latency as possible, especially on loads which are responsible for compulsory misses.

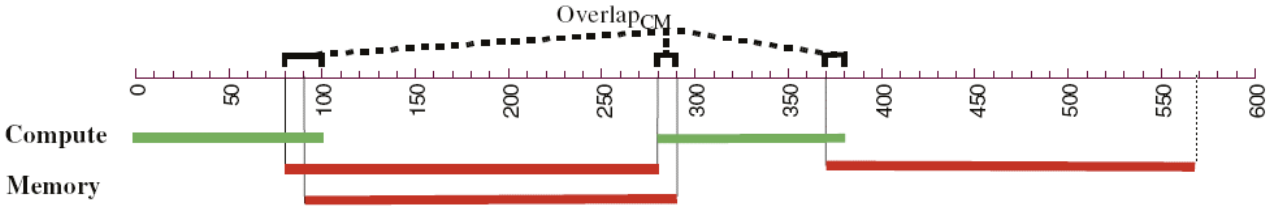

Figure 1: Example of Compute-Memory and Memory-Memory Overlap.

Benchmark	*Mem. Latency*	*L2 Miss Rate*	*Simulation Cycles*	*IPC*	*Possible Increase*
GZIP – smred.program 1	1	14.86%	1666287596	1.5638	-
GZIP – smred.program 1	200	14.86%	3993170848	.6526	140%
VPR – smred	1	0.71%	105134799	1.1510	-
VPR – smred	200	0.71%	111325030	1.0870	5.8%

Table 1: Effect of memory latency on IPC

			Total number of L2 cache misses with X prior correct branch predictions					
Benchmark	**Cache Size**	**L2 Miss Rate**	**0**	**1**	**2**	**3**	**4**	**5+**
GZIP-sm-red.program 1	64k	14.86%	581943	364786	594324	558057	302547	**11611031**
GZIP-sm-red.program 1	256k	0.58%	35607	19290	14169	13402	12198	**449825**
VPR – smred	64k	0.71%	13710	6729	4563	3433	2114	**8883**
VPR – smred	256k	0.13%	2715	977	577	407	228	**2182**

Table 2: Correct branch predictions before L2 miss

c. Preliminary Findings

Before implementing our design, we ran simulations with two SPEC2000 benchmarks to verify that memory latency was indeed a problem that affected IPC, which our proposed architecture has the potential to improve. We ran tests on GZIP and VPR with the memory latency set to one cycle vs. the memory latency set to two hundred cycles. The results are summarized in Table 1. On GZIP, which had a significant L2 miss rate, there is a potential for a 140% IPC increase. On VPR, however, since the miss rate was extremely low, the increase in latency had a minor effect on the IPC, but still showed possible improvement.

These tests were implemented with a cache size of 64KB, which we believe is reasonable, considering the small size of the input files we are using. We did not want the input to fit into cache.

Next, we tracked the total number of successful branch predictions prior to an L2 cache miss, which again illustrates the potential for improvement. We used an 11-entry gshare branch history table with 2K entries, and we compared the results with the smaller, 64KB cache, as well as with a larger size, 256KB. Table 2 verifies that the occurrence of many missing loads instructions can be predicted several branches in advance.

The remainder of this paper is organized as follows: Section II describes in detail the architectural specifications of the FLAP, Section III discusses which experiments and benchmarks we ran to evaluate our hypothesis, Section IV analyzes the results of these simulations, and we conclude with Section V, which presents a cost analysis and suggests possibilities for future work.

II. ARCHITECTURE

a. Load-Tracking Table

The additions we have implemented in the simulator are located off the critical path. The most important feature is the load-tracking table which is used to cache the PC of previously encountered loads and perform stride address prediction. The table is indexed by load

PC, and also contains the data required to make stride address predictions, as well as two confidence counters. The default table used is a 2-way, 4096-entry content-addressable memory that uses pseudo-random replacement. Based on several preliminary trials, we found this to be a good configuration.

All loads which miss in the L2 cache are put into the table. However, all loads are snooped; when a load instruction is executed by the processor, it is looked up in the table. If the load hits in the table, its stride prefetching data is updated. We have used a stride-based address prediction scheme capable of predicting memory address strides with 9 bit resolution. Through experimentation, we have discovered that almost all strided loads fall into this category. Once in the table, the load entry is tracked with two 2-bit confidence counters. One is used to see how many times the processor has missed on this load, and the other is used to see if the stride address predictor can return the correct address. This provides a safeguard against cache pollution in the L2. Therefore, once there have been four misses on the load, and the address has been predicted correctly three times in a row, only then will that load instruction be prefetched when encountered by the controller.

b. Controller

The other component of the FLAP is the controller. The controller is responsible for running ahead of the processor and querying its branch predictor. Since we would be sharing the processor's branch predictor but not it's PC, we needed to keep our own return address stack, which the controller also manages. Each cycle, the controller tries to advance its PC by several (4 by default) instructions. In order to accomplish this, each instruction must be checked in the BTB to see if it is a branch. If the instruction hits in the BTB, the branch predictor is queried to predict the branch. If the instruction is not a branch, the FLAP table is queried (the BTB and FLAP table queries can occur in parallel). PCs which hit in the FLAP are known loads, and if they pass the aforementioned confidence criterion, and would stride outside of a cache line boundary, they are issued. If a PC is neither found in the BTB or the FLAP table, then the instruction is ignored and the FLAP PC is advanced. We also use a run-ahead limiter which caps number of instructions the FLAP can run ahead of the processor's PC (100 is the default); this also helps to prevent cache pollution due to prefetches that are either too far ahead to be used before they get evicted from the cache or whose occurrence is incorrectly predicted by the branch predictor. The controller is also responsible for resetting the FLAP PC to the processor's PC every time there is a branch misprediction.

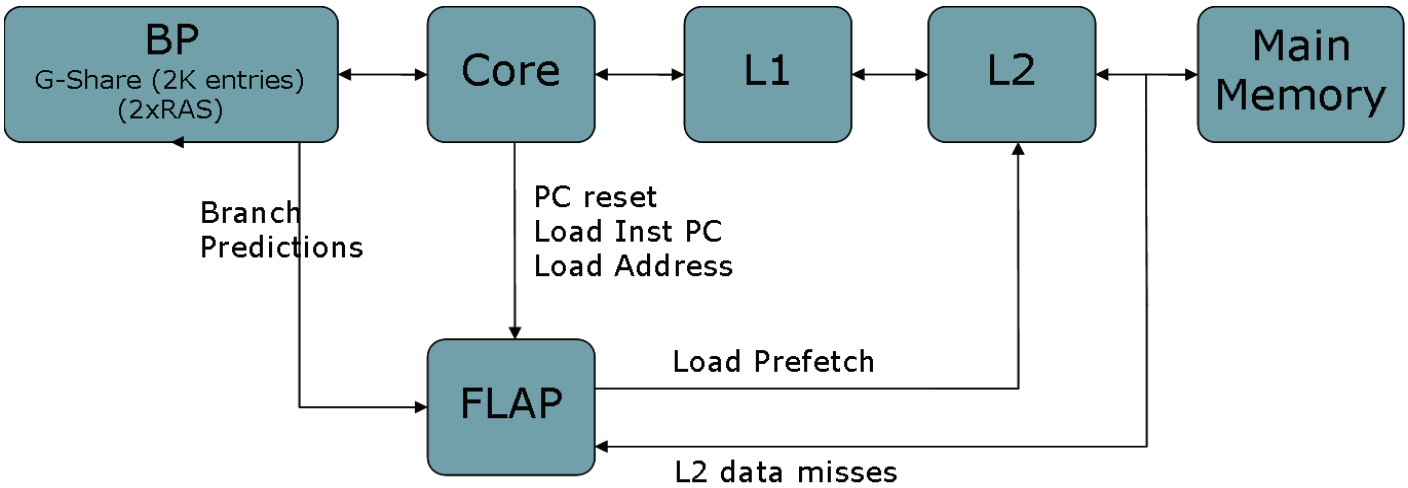

Figure 2: FLAP Implementation

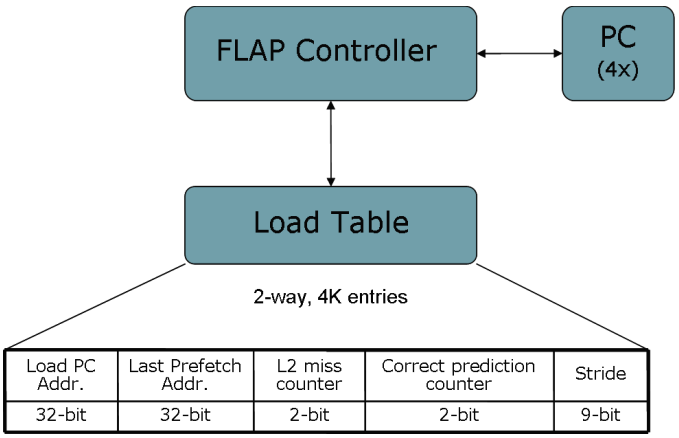

Figure 3: FLAP Components

III. EXPERIMENTAL METHODOLOGY

a. Simple Scalar

We implemented our architecture with SimpleScalar's sim-outorder source because we needed its features to access the branch predictor, snoop addresses, and issue loads. Since sim-outorder runs slowly, one of the drawbacks was that we needed to run our tests with small input data sets and therefore used small cache sizes so that the datasets would not fit into the cache. The FLAP table was implemented as an array, loads were monitored in the sim-outorder load instruction handling subroutine, and prefetches were issued through the L2 using the cache_access function.

There are many variables to the FLAP architecture – Table 3 lists the default values of our parameters and the different variations that we tested. The full set of simplescalar parameters used can be found in the appendix. We chose a good baseline configuration, and then proceeded to vary each parameter individually to see how it would affect the FLAP performance. Most of the changes had an effect on cache pollution, how much latency could be hidden, and the coverage and accuracy of the prefetcher.

b. Benchmarks

Three applications from the SPEC2000 benchmark suite are used to assess the architectural modifications we implemented. All three benchmarks belong to the integer component of the SPEC2000 suite: 175.vpr (VPR) which is used for FPGA circuit placement and routing, 181.mcf (MCF) which is used for combinatorial optimization, and 164.gzip (GZIP) which is used for compression. MCF has a high miss rate and highly-strided loads, and therefore stands to improve the most from the FLAP. VPR has an extremely small miss rate with very irregular loads, whereas GZIP falls in the middle with a modest miss rate and mildly strided loads. These factors contribute to the results shown in the next section.

Parameters	***Default***	***Variations***
No. of PC's to skip ahead/clock	4	2
Max instruction look ahead	100	30, 200
Table associativity	2-way	1-way, 4-way
Table size	4096 entry	1024 entry, 16384 entry
Counter width	2-bit	4-bit
FLAP location	Between L2 and main memory	Between L1 and L2
L2 cache size	64K	256K
L1 data cache size	16K, 4-way	---
L1 instruction cache size	16K, direct-mapped	---
Branch Predictor	G-Share with 2k entry	---
Address Predictor	Stride (9-bit)	---

Table 3: FLAP Parameters

IV. EXPERIMENTAL ANALYSIS

a. Base Configuration Results

In order to discover how effective the FLAP was at increasing IPC in an out-of-order superscalar processor, we ran several benchmark simulations. Figure 4 compares simulations of the unmodified version of sim-outorder, sim-outorder with the FLAP implementation, and an ideal situation where the main memory latency of all loads is one cycle.

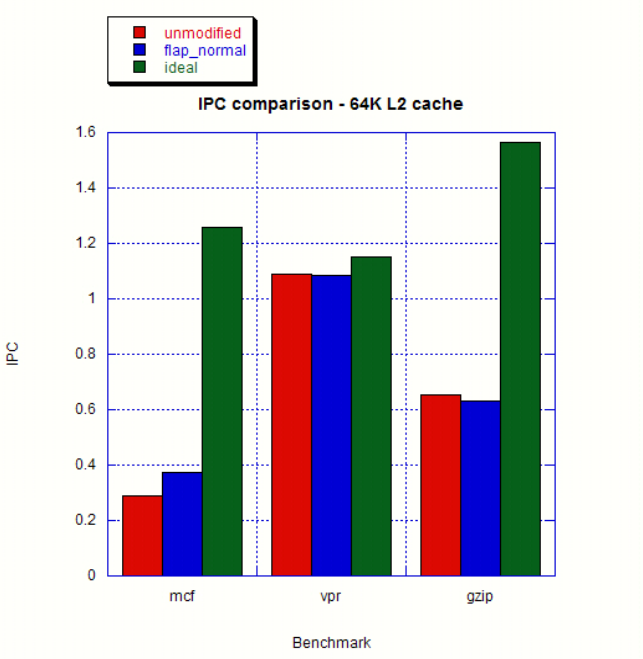

Figure 4: IPC comparison with 64K L2

Using a 64K L2 cache, VPR and GZIP experienced very minor performance degradations, which are due to cache pollution, but MCF experienced a 29% increase in IPC. To see if we could get rid of the loss in performance due to cache pollution, we ran these benchmarks again using an L2 size of 256K; these results are shown in Figure 5.

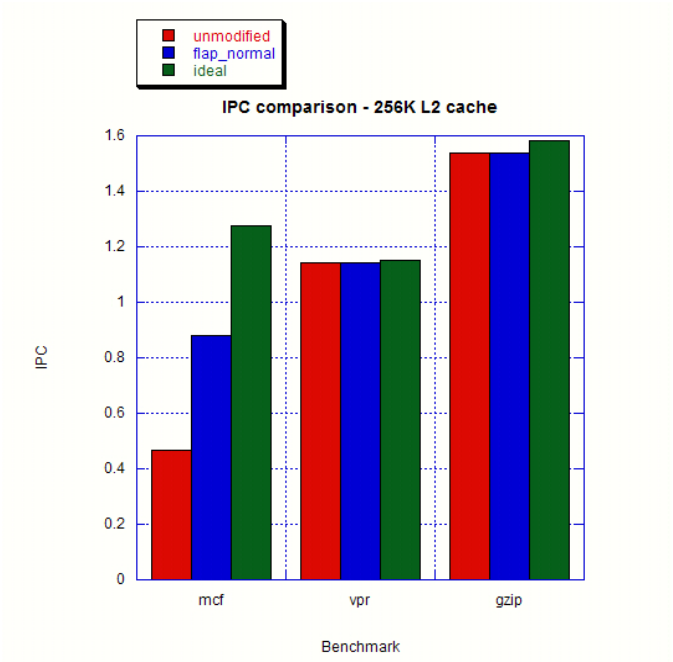

Figure 5: IPC comparison with 256K L2

This time there was almost no degradation for VPR or GZIP, and even the ideal version didn't do too much better. The most interesting thing to note is the 87% increase in IPC for MCF. The result shows that our technique can increase IPC, especially for processors with reasonably sized caches running applications with high miss rates and predictable load addresses. The results with MCF serve as a proof-of-concept for the FLAP. Using more accurate address predictors and better controls on cache pollution, it is obvious that the FLAP can benefit any program which suffers from memory latency related stalls.

b. Parameter Variation Results

In order to judge the effect of the myriad of FLAP parameters, we have simulated several FLAPs whose parameters have been varied from the base configuration. The following two graphs compare all of the variations of the parameters listed in Table 3, using a 64K cache. Figure 6 is for all three benchmarks, and Figure 7 shows a closer look at the results for MCF.

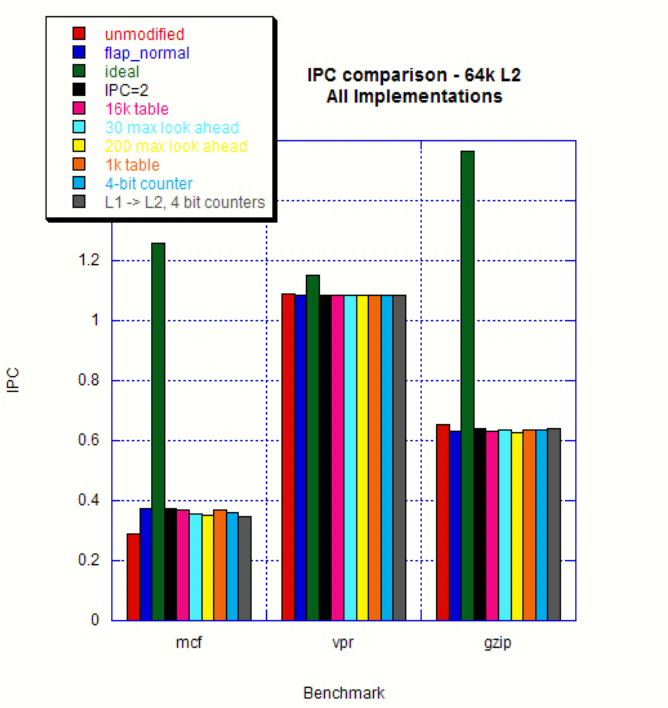

Figure 6: Comparison of FLAP variations

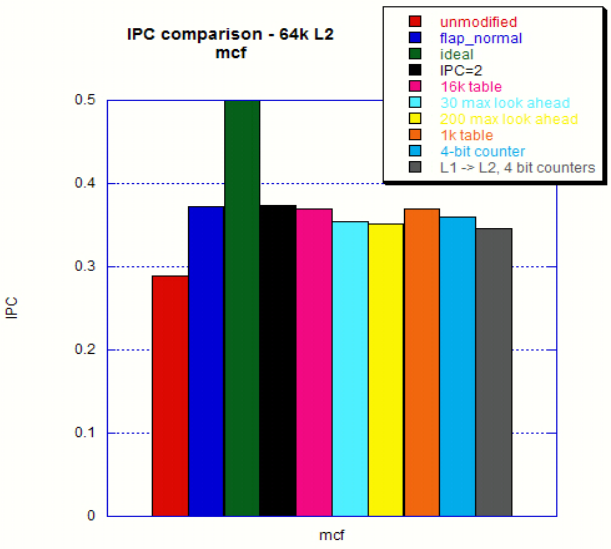

Figure 7: FLAP variations for mcf

Again, VPR and GZIP experience little or no performance change when each of the parameters is varied.

However, there are a few interesting things to note about the results shown in Figure 7. First, varying the size of the table has very little effect, revealing that there is only a very small amount of load instructions which cause the majority of misses. Therefore, a hardware implementation can achieve good results with a simple 1K entry direct-mapped table. Also important is the fact that decreasing the FLAP IPC from 4 to 2 resulted in a negligible drop in performance. A hardware implementation with a FLAP IPC of 2 would put less strain on the BTB, branch predictor, and FLAP table by requiring these structures to have less banking and read ports. However, a related parameter, the run-ahead amount, does have a substantial effect: too short of a distance doesn't hide enough latency; too large of a distance causes data to be prefetched into the cache either so early that it is evicted before it can be used, or for loads which are not actually encountered. This tension is a central issue with the implementation of the FLAP. Future designs must pay close attention to how far ahead of the processor they are and how confident they are that a given load will be reached. When the FLAP is moved between the L1 and L2, it doesn't perform as well, losing 7% compared to its default location, but it is still able to increase IPC. This is interesting considering the small memory latency of the L2 and the very small size of the L1.

c. Experimental Concerns

Our primary experimental concerns were time constraints and cache pollution. Because we were working with the slow-running sim-outorder, we were forced to use small data sets and therefore small cache sizes. With the 64K L2 cache, we saw the effects of cache pollution decrease performance on GZIP and VPR, as well as limiting the performance increase on MCF. Larger caches allowed for better FLAP performance. Another alternative which should be explored is using a separate buffer for the speculative loads to store data in. The buffer could be accessed for missing L2 loads, or, more aggressively, in parallel with the L2. This buffer would eliminate cache pollution problems.

Additionally, the FLAP's performance depends heavily on the coverage and accuracy of its address predictor. With a better address predictor, the FLAP will be able to prefetch

more useful loads. Tables 4 and 5 show the coverage (the percentage of missing loads the FLAP was able to prefetch) and accuracy (what percentage of prefetched loads the FLAP accurately predicted) for both 64K and 256K L2 caches, using the 9-bit stride address predictor. We use a stride predictor because of its ease of implementation and because it performed well according to the statistics provided[2].

We believe that the main reason why GZIP did not experience a performance increase is that, unlike MCF, its individual load instructions do not have very strided accesses, despite its fairly sequential memory access pattern. Looking at Tables 4 and 5, one can see that while the FLAP was able to accurately predict program flow, its stride address predictor was unable to accurately predict the addresses many missing loads. Therefore, if more accurate and higher coverage address predictors were used, other applications will also show performance improvements. Given more time, we would have liked to run more applications, use larger data sets, larger caches, use a separate buffer, and a higher coverage load-address predictor.

Benchmark	Coverage	Accuracy	Bandwidth Increase %	Branch Predictor Accuracy %
MCF	19.000	58.000	13.000	95.330
GZIP	0.0000	-	17.000	96.600
VPR	0.0000	-	6.0000	95.210

Table 4: Prefetch Stats with 64K L2

Benchmark	Coverage	Accuracy	Bandwidth Increase %	Branch Predictor Accuracy %
MCF	27.000	76.000	8.4000	95.660
GZIP	1.0000	0.70000	141.00	96.640
VPR	0.0000	-	0.37000	95.210

Table 5: Prefetch Statistics with 256K L2

V. CONCLUSIONS

a. Hypothesis Evaluation

For MCF we saw a large speedup, proving that the addition of the FLAP has the potential of significantly improving IPC. Our hypothesis, as stated in Section I, was validated for applications with high miss rates and highly strided loads. The FLAP was able to significantly increase IPC by hiding a large amount of memory latency. This was possible because the FLAP controller was able to use the BTB, branch predictor, and FLAP table to prefetch many loads far ahead of their execution. However, on applications which did not exhibit high miss rates, or whose loads were not predictable, there is no significant change in IPC. This can be seen especially for GZIP and VPR with a 256K L2 cache.

b. Cost Analysis

In order to conclusively determine whether the cost of the FLAP is worth its potential benefits, more experiments would need to be conducted. In considering its hardware implementation, the FLAP table for our base configuration is 4-way and 4096 entries. With 77 bits per entry, this yields a total capacity of 38.4KB. This is a small size, and our tests demonstrated that it can even be reduced to a direct-mapped 1024 entries (9.63KB) without loosing much performance. This is not very costly, especially when compared to the size of modern L1 caches. One major implementation concern is the branch predictor. Since we are using the processor's branch predictor but have our own PC, an additional return address stack is required for this implementation. Due to the number of accesses to the BTB (and potentially the rest of the branch predictor), the BTB may need to be multi-ported or multi-banked; we may even require a duplicate branch predictor to avoid access conflicts with the processor. To reduce the strain on these components, a lower FLAP IPC could be used (as seen in section IV, b), thus reducing the number of queries per cycle to the BTB, branch predictor, and FLAP table. We believe that a smart hardware implementation, especially with an improved address predictor, would be a worthwhile addition to any aggressive processor design which suffers from load-based memory latency stalls.

c. Future Work

There is still much more to study about the FLAP. Our results suggest that this concept has great potential; with further work, we believe all applications can show a positive improvement. In addition to applications with a larger data set, we feel that media applications could also benefit greatly from the FLAP. A larger L2 as well as a separate buffer for the speculative loads would reduce cache pollution and therefore allow for better performance. An improved branch predictor would allow us to follow the program flow more accurately, because (as we showed in Section IV) our results are heavily dependent on the accuracy and coverage of our address predictor. Also, using the correct look-ahead distance is very important, and having a better method to control this would be beneficial. Rather than just limiting the number of instructions we go ahead, it would be good to try limiting the number of branches ahead and pending load prefetches. This might provide a better way of balancing the need to hide latency while avoiding cache pollution and wasting memory bandwidth.

If these experiments are carried out, we are confident that with accurate predictors and larger caches, the FLAP can greatly enhance processor performance by improving memory latency hiding and increasing IPC.

REFERENCES:

[1] Yuan Chou, Brian Fahs, and Santosh Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism", *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[2] Glenn Reinman and Brad Calder, "Predictive Techniques for Aggressive Load Speculation", *Proceedings of the 31st International Symposium on Microarchitecture*, November 1998.

[3] Glenn Reinman, Todd Austin, and Brad Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery", *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[4] Tse-Yu Yeh and Yale Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History", *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

APPENDIX:

Full simplescalar configuration:

-fetch:ifqsize	4 # instruction fetch queue size (in insts)
-fetch:mplat	3 # extra branch mis-prediction latency
-fetch:speed	1 # speed of front-end of machine relative to execution core
-bpred	2lev # branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:2lev	1 2048 11 1 # 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:ras	8 # return address stack size (0 for no return stack)
-bpred:btb	512 4 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update	<null> # speculative predictors update in {ID|WB} (default non-spec)
-decode:width	4 # instruction decode B/W (insts/cycle)
-issue:width	4 # instruction issue B/W (insts/cycle)
-issue:inorder	false # run pipeline with in-order issue
-issue:wrongpath	true # issue instructions down wrong execution paths
-commit:width	4 # instruction commit B/W (insts/cycle)
-ruu:size	16 # register update unit (RUU) size
-lsq:size	8 # load/store queue (LSQ) size
-cache:dl1	dl1:128:32:4:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat	1 # l1 data cache hit latency (in cycles)
-cache:dl2	ul2:256:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat	6 # l2 data cache hit latency (in cycles)
-cache:il1	il1:512:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat	1 # l1 instruction cache hit latency (in cycles)
-cache:il2	dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2lat	6 # l2 instruction cache hit latency (in cycles)
-cache:flush	false # flush caches on system calls
-cache:icompress	false # convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat	200 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width	8 # memory access bus width (in bytes)
-tlb:itlb	itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb	dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat	30 # inst/data TLB miss latency (in cycles)
-res:ialu	4 # total number of integer ALU's available
-res:imult	1 # total number of integer multiplier/dividers available
-res:memport	2 # total number of memory system ports available (to CPU)
-res:fpalu	4 # total number of floating point ALU's available
-res:fpmult	1 # total number of floating point multiplier/dividers available