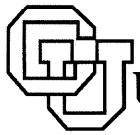
Static Methods in Hybrid Branch Prediction

Dirk Grunwald
Donald Lindsay
Benjamin Zorn

CU-CS-831-97



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Static Methods in Hybrid Branch Prediction

Dirk Grunwald, Donald Lindsay, and Benjamin Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-831-97 February 1997

University of Colorado at Boulder

Technical Report CU-CS-831-97
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Static Methods in Hybrid Branch Prediction

Dirk Grunwald, Donald Lindsay, and Benjamin Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Contact Author: Benjamin G. Zorn
Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: zorn@cs.colorado.edu

February 1997

Abstract

Recent microprocessors contain hybrid branch predictors that attempt to combine the predictions of multiple single-level or two-level branch predictors. The prediction-combining hardware — the selection mechanism — may itself be large, complex and slow. We show that the combination function is better performed statically, by a profiling or static analysis technique. An important advantage of our approach is that a branch site need only cause interference within a single component predictor. And, although prediction combination itself is now static, the actual predictions remain dynamic, so there is no specific risk of worst-case performance.

Static selection requires that information be stored in or with the program text. The most obvious implementation, dedicated branch opcode bits, would seem unavailable to pre-existing ISAs. We argue that a number of recent implementation techniques, such as code rewriting and decoded caches, are applicable. In various combinations, these techniques could be used to solve not only encoding problems, but pipeline problems. We also argue that our proposal addresses the scaling issues currently facing hardware designers.

Miss-rate results are presented which show a clear and consistent performance advantage. Using a cross-validation technique, we show that our static method is robust in the face of traces that it was not profiled on. These results are obtained with a realistic whole-system benchmark, and with realistic hardware sizes.

1 Introduction

Branch prediction is considered to be an important research topic because misprediction penalties have become a major impediment to high processor utilization. Branch predictions are used by deeply pipelined designs, so as to avoid pauses in the instruction prefetching. On a modern wide-issue processor, each one-clock pause means that up-to-N instructions were not issued. Worse, issue width is now similar to the size of a basic block, which means that a branch predictor must do about one prediction per clock, and in the near future, perhaps two per clock.

Research in this topic is not disconnected from industrial practice. In fact, recent microprocessors such as the MIPS R10000 [16], and the Alpha 21164 [6], contain extensive prediction hardware. More recent hardware is reportedly faithful to published predictor designs.

Recently there has been interest in hybrid branch predictors. The fundamental idea here is that different predictor schemes have different advantages. Hence, a combination of basic schemes might (at some cost) obtain all of their advantages. And, in fact, a hybrid predictor is known to be in the upcoming Alpha 21264 [9].

In Section 2, we will point out what some of these advantages might be. We will also give some idea of the suggested implementation techniques, and the cost and speed penalties that might be involved.

This paper proposes that the prediction-combining hardware — the selection mechanism — be replaced by a static choice that is made once per branch site. The most obvious implementation would be to dedicate one or more branch-opcode bits to this purpose. (Some alternative implementations will also be discussed.) This approach has a clear advantage in terms of hardware cost, and may have an advantage in terms of the predictor's critical path.

This paper shows that the prediction accuracy is acceptable. In fact, we will argue that the prediction accuracy can improve, because components can view less of the workload stream. And, in Section 5, we will demonstrate an improvement, relative to a comparable dynamic hybrid predictor. For example, one result shows a 4096-bit dynamic selection mechanism getting a 4.6% miss rate, while our static approach gets 3.4%. Much more importantly, we will show that our results are never worse than those of the dynamic selection mechanism, and this will be shown across a realistic range of benchmarks and configurations.

This paper is organized as follows. In Section 2, we will survey recent literature, and define some of the more important terms and prediction schemes. In Section 3, we introduce our proposal, and explore hardware and software implementation issues. In Section 4, we describe a group of sample hardware designs, and a benchmark with which to evaluate them. In Section 5 we present a variety of evaluations, and in Section 6 we summarize and discuss some possibilities for future work.

2 Previous Work

There is an extensive literature on branch prediction, to the point where a 1992 taxonomy [17] is already becoming dated. Typically, predictors use the program counter, and/or recent branch history, to index into an array of up/down saturating counters. The actual prediction is based on whether the counter contains a high or low value. Counters typically use two or three bits, so as to reflect the 'recent popular direction' for that subset of the branch stream.

A branch history is simply a short shift register, which receives one bit each time a branch is taken or not. A branch history may be global, in which case it is kept in a Branch History Register. A history may be kept per branch site, in which case a Branch Target Buffer ("BTB") is usually kept, with address tags, and typically with 2-way or 4-way associativity. An intermediate method uses a tagless Branch History Table, which permits different branch sites to be aliased to the same branch history.

The idea of hybrid predictors was proposed by McFarling [12]. He describes a series of simple predictors, most notably "gshare", which we show in Figure 1. The paper argues about why each simple predictor works, and concludes that to some extent each works in a different way. Hence, a "combining predictor" is suggested, which is a specific variant of the general scheme shown in Figure 3. The "combining predictor" contains two component predictors, both of which are operated normally. Additional hardware then uses the PC to select from yet another array of 2-bit counters. The selected counter's value indicates which component predictor has been the most successful lately for that branch site. Given this indication, the hardware simply chooses one of the two available predictions.

Update of a combining predictor, shown quite generically in Figure 4, involves changing the selected counter if one component predictor was right and the other was not. The component predictors are also updated, in whatever way is normal for each specific component.

The paper concludes, from performance on SPEC89, that combining predictors have better performance than any single predictor of the same hardware size. Note that the selection mechanism's array was the same size as the arrays of the component predictors, so the cost of that extra array was accounted for when normalizing performance versus hardware cost.

Chang, Hao and Patt [2] looked for good combinations of components. They did not want to try all possible combinations, in all possible hardware size ratios. So, they used a 1:1 ratio, and determined, per branch site, the effectiveness of each component. This allowed a static computation of the miss rate of an "idealized" hybrid predictor. The paper argues that this ideal is in fact not perfect, since it does not adapt to program dynamics. However, the method allowed them to choose the best pairing, which they determined to usually be gshare plus "PAs". ("PAs" [17] [14] is described in Figure 2.) They argue that the pairing is successful because gshare does well on site correlation, and PAs does well on successive uses of one site. They also suggest that some predictors have a quicker warm-up time (i.e. a smaller training cost).

The paper suggests several "two-level" implementations for dynamic selection mechanisms, and reports their performance on SPECint92.

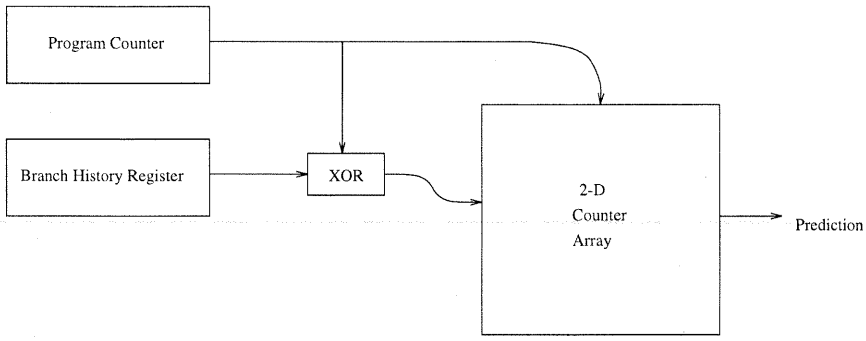


Figure 1: The gshare branch predictor.

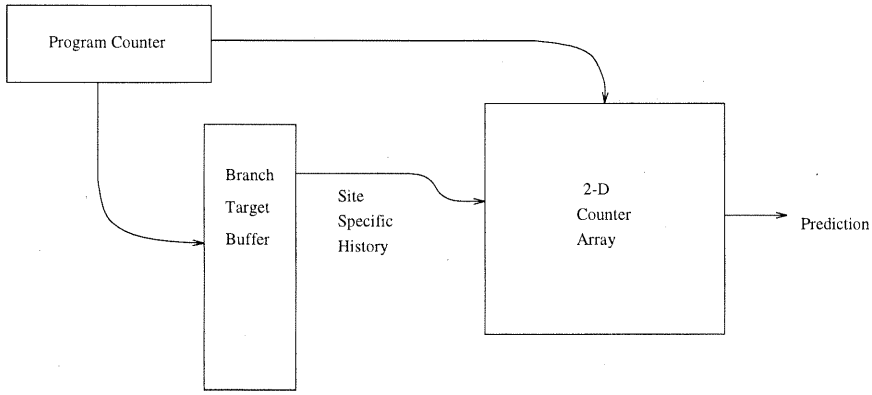


Figure 2: The PAs branch predictor.

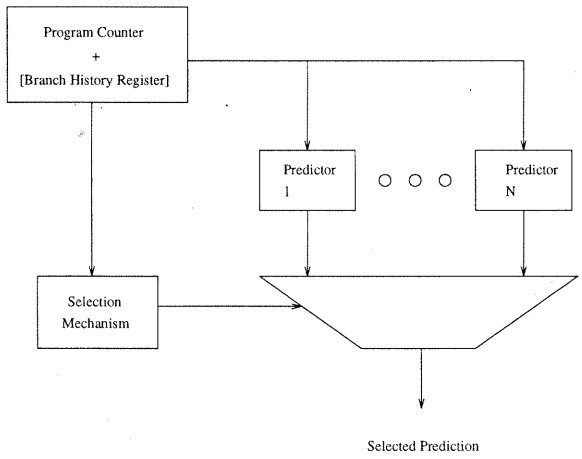


Figure 3: A generic hybrid branch predictor.

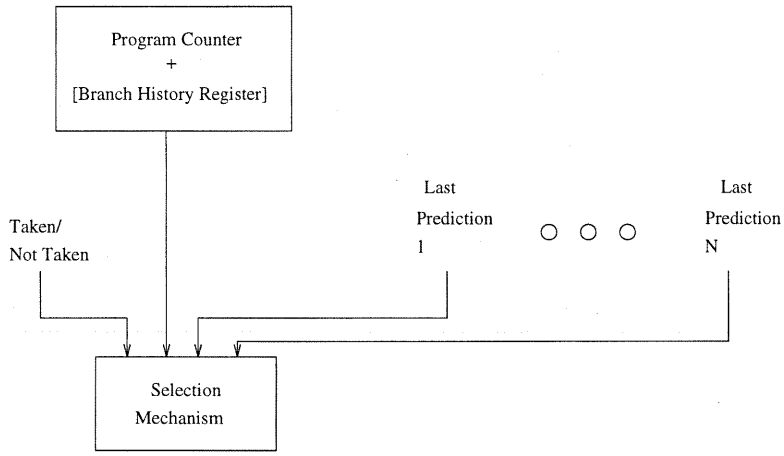


Figure 4: Updating the generic hybrid branch predictor.

In a more recent paper Evers, Chang and Patt [7] consider a larger selection of component predictors, and propose a hybrid predictor with not two components, but six. This "multi-hybrid" contains unequal sized elements, in the expectation that some will warm-up (train) more quickly after context switches. It also contains a component predictor which is specialized for predicting loops.

The basic problem with this concept is that the selection mechanism is expensive. To make a multiway choice, it cannot use anything so simple as one two bit counter. So, they propose having six two-bit counters in each line of a 2048-high BTB. Each counter rates a given component, and the highest-rated component wins. A priority scheme resolves ties, but update rules are necessary to prevent ties from becoming the common case.

The paper studies this predictor's performance under a load of context switches, which are simulated by periodic flushing of the predictor state. This simulation method is deprecated in Gloy, Chen and Smith [8], which compares results from this method against results from full-system traces. They argue (for example) that flushing makes large tables appear pointless, whereas in actuality they reduce aliasing between kernel and user branch sites.

A major difficulty of a 6-way hybrid is the rather long critical paths through the selection mechanism. Evers, Chang and Patt do suggest a way whereby updates can precompute the highest ranked component, reducing the work that must be done during that BTB line's next prediction. But in any case, it is clear that this whole approach has increased costs, in both hardware area and in time. The problem with any such scheme is to assess not only its effectiveness, but also its cost-effectiveness.

Instruction set support for branch prediction was part of the RISC revolution of the 1980's. In some cases, it was an explicit part of the ISA that programmers should expect "BTFNT" (Backward Taken, Forward Not Taken) prediction. The PowerPC [5] has a bit for reversing BTFNT. It could be argued that the loop

instructions of the IBM Power (and PowerPC) ISA permit excellent prediction of inner loops, including potential prediction of loop exit. (This is because there is a hardware register which is dedicated to loop counting, and a processor implementation could specifically detect when the loop is one cycle away from termination.)

Other ISAs such as SPARC and Intel i960 [13] allow compilers to provide a branch prediction, via a bit in the branch opcode. There are some minor variations on this in the literature. For example, implementing the classifying predictor of Chang, Hao, Yeh and Patt [3] probably involves having an "almost never taken" branch form, and an "almost always taken" form. (In this case, "almost never" means less than 5% of the time.) The predictor in the upcoming PA-8500 [11] keeps track, not of branch direction, but of whether or not the branch went in the direction indicated by the static hint. The HP designers argue that a conventional predictor suffers whenever a usually-taken branch and a rarely-taken branch are aliased to the same prediction counter. With the HP design, however, such situations will have little impact if each opcode bit was set correctly. The counter will most likely represent a correct prediction for both branches.

3 Static Hybrid Predictors

The basic idea behind our approach is that it would be nice to simply eliminate the dynamic selection mechanism. After all, that hardware component only exists in order to decide who to believe. A bit in a branch instruction can make just such a choice.

A simple way to compute such selection bits is by profiling. There is, for instance, literature which is positive about the degree to which profiles are predictive of subsequent executions [1]. Profiling was originally a compiler technique, but more recently, code rewriting systems, such as ATOM, have been used to generate profiles. And, some recent microprocessors have hardware which can be used to gather profiles of even real-time systems [4].

The most obvious place to store selection bits is in branch opcodes. This poses a problem for most pre-existing instruction set architectures. However, it is not a problem for all ISAs: for instance, upcoming VLIW designs. And, with code rewriting techniques, conventional ISAs can undergo adjustment. The selection bits can also be promoted into cache line headers, or trace cache headers, which makes them available at pipeline-convenient times. The selection bits can also be promoted into non-branch instructions, which are suitably coupled to the actual branch. Predecessors in the control flow graph are all suitable, and predecessors on a trace are also candidate locations. In particular, NOPs inserted by code scheduling could be quite convenient.

A potential performance drawback of the basic idea is that superficially it cannot track dynamic behavior, at least at a fine or medium grain. It has been argued [7] that this is important, because some programs may

go through phases of execution, each of which should be characterized. This argument is not enormously strong, because we are not, after all, proposing fully static prediction. Actual predictions are done by a component predictor, which is most likely dynamic, and therefore capable of adaptive behavior.

This negative argument is nevertheless a point which must be dealt with. In the data presented later, we are careful to compare our results to those of a reasonable dynamic hybrid predictor. Also, by cross-validation, we will show that phases of program execution do not seem to be a problem.

Static selection quickly leads us to a large number of topics. It is, for instance, an appealing idea because it would support an N-way selection mechanism as straightforwardly as a it supports a 2-way. If only the selected component is cycled, then the idea allows component predictors to have unequal or even variable prediction latency. It also suggests a simple way to obtain multiple predictions per clock, by "coloring" branch sites so that successive branches are unlikely to want the same component predictor. Without such freedom, it is difficult to see how the designers of the "billion transistor chip" will be able to spend very many of those transistors on prediction. It would surely be uphill to use a high-latency subsystem as a latency cure.

A more subtle performance issue is the question of pipeline staging. Examining Figures 3 and 4, we see that a hybrid predictor goes through three steps in sequence: (1) The PC and BHR flow into the components; (2) predictions emerge from each component, and a selection action chooses a prediction; (3) the resolved branch information is used in updating actions.

In fact, step 3 may occur much later (and may not occur before the next prediction). However, the critical question, here, is when information from the code stream must be used. In conventional systems, the answer is that the code stream affects step (3). Our proposed information could be used in step (1), so that only one component predictor need be cycled. If a parameter of the evaluation came from the code stream, then this pipeline timing would be a strong requirement of the scheme. However, this paper does not propose that, and thus our information is not required until step (2).

The above argument places branch prediction schemes into three timing categories. A given pipeline/cache design may or may not incur penalties for a type (1) predictor. However, since our proposal is of type (2), we believe that there is no strong impediment to an aggressive implementation of our scheme. Note that the prediction bits in various RISC ISAs imply that their designers planned type (2) implementations.

4 Experimental Design

4.1 Selection of a Configuration

The first test of our proposal is to select some predictors, select a benchmark, and obtain a profile.

In Section 2 we discussed several hybrid predictors. From these, we chose to implement the McFarling combining predictor. It has the advantage that the implementation details are clear, for any desired size. It has no free parameters, and therefore gives a clear point of comparison between papers. Also, the data in the hybrid predictor literature [12] [2] [7] show that it works quite well.

Chang, Hao and Patt [2] reported that their best predictor combination was PAs [17] plus gshare [12]. Both of these predictors have free parameters, but papers such as Sechrest, Lee and Mudge [14] report the most effective ratios between the parameters. It was decided that the PAs predictor would have a 256-high BTB, plus a one-dimensional array of 512 counters. Further, the BTB was given 4-way associativity, and an "update on taken" update policy. This policy means that if an update would cause a BTB eviction, and if the branch was Not Taken, then the BTB update is ignored. This policy choice is only sometimes advantageous; but in any case, it was used throughout, and therefore does not lead to any unfair comparisons.

The gshare predictor was given 2048 counters, organized as 2^7 by 2^4. Following McFarling's suggestions [12], the McFarling combining predictor also has 2048 counters.

Results are also reported with everything four times the size, to to test the stability of the configuration, and the stability of the profile. The total storage devoted to this larger size is extremely small, compared to some simulated configurations in the literature [2][7]. However, it is actually somewhat on the large side, compared to current hardware practice. For instance, the MIPS R1000 [16] uses a bimodal [12] (a.k.a. 2bC [2]) predictor of only 512 counters. The Alpha 21164 [6] uses 2048 counters in the same organization. As we have already pointed out, large predictors are necessarily slow, and even given unbounded area, an optimum size must exist.

For reference, we will also show the results for the component predictors, operated standalone. And, we will show the results of double-sized components. Specifically, we present gshare predictors that are 2^7 by 2^4, 2^8 by 2^4, 2^8 by 2^5, and 2^9 by 2^5. We present a PAs predictor with a doubled BTB height of 512 instead of 256. (The PAs has 512 counters in both cases, since PAs is relatively insensitive to the addition of counters.) Since the hybrid predictor uses components of approximately equal size, doubled components give some idea of the performance of alternative organizations having comparable overall costs.

4.2 The Benchmark Suite

The results here use the Ultrix portion of the IBS traces [15]. These traces have become popular because they are traces of an entire system, including the kernel, interrupt handlers, daemons, and an X server. They are also fairly accurate, although they are not perfect. (Discrepancy checking turns up a small set of problems, which can be attributed to buffer overflows during the original hardware monitoring process.) And, they are long enough to capture entire file system operations. As shown in Table 1, they vary from 40 million to 150 million RISC instructions.

Benchmark	Millions of Ops	% of Ops in Kernel	Conditional Branch Sites
groff	105	13.6	6,292
gs	118	13.3	12,761
jpeg_play	151	8.0	7,828
mpeg_play	99	21.8	5,571
nroff	130	8.3	5,230
real_gcc	107	12.7	17,300
sdet	42	98.7	5,284
verilog	47	22.6	4,613
video_play	53	69.9	4,575

Table 1: Charactieristics of IBS Suite

Their deficiency is that they are from a small, slow machine (a 16 MB MIPS R2000), so recent large, fast servers have a somewhat different I/O balance. For our purposes, that is not as important as the fact the traces avoid the small-footprint and user-only problems that have been noticed in program suites such as SPEC [14] [8]. The number of branch sites per trace varies from 4575 to 17300. The median distance between branches ranges from 3 to 6 instructions, but the median distance between re-reference of the same branch site varies from 17 to 250. That is the variation from trace to trace: different sites have median re-reference distances which span five orders of magnitude, and the distribution of those medians has a tremendous tail. The interbranch distance is not different between kernel and user state, but the site inter-reference distance has consistently higher means when the kernel state is included. All of these facts indicate that the IBS traces are entirely different from (say) SPECint92's "eqntott", where a mere five branch sites are responsible for 90% of the dynamic conditional branches.

4.3 A Simple Profile

Each trace was run through the simulated component predictors, and a count of wins (correct predictions) was obtained per component per branch site. Each site is then assigned to the component with more wins. Figure 5 gives a breakdown, showing the percent of dynamic branches which have been allotted to each component. These numbers give some idea of the imbalance between the predictors. The four-times-larger predictor makes slightly more use of PAs (which becomes more efficient at that size) but still represents a subsystem that could reasonably be implemented in the near term.

The "tie" category comes from sites whose two win counts were approximately equal. (In this case, "approximately" means that gshare's win count, at a site, was within 5 of the PAs win count.) The tie category was measured to capture the sites in the distribution tail, whose normalized dynamic weights are effectively in the part-per-million range. In this regime, the exact win counts do not have great significance for our purposes, and ratios of counts would have even less significance. So, the "fuzziness" of the count

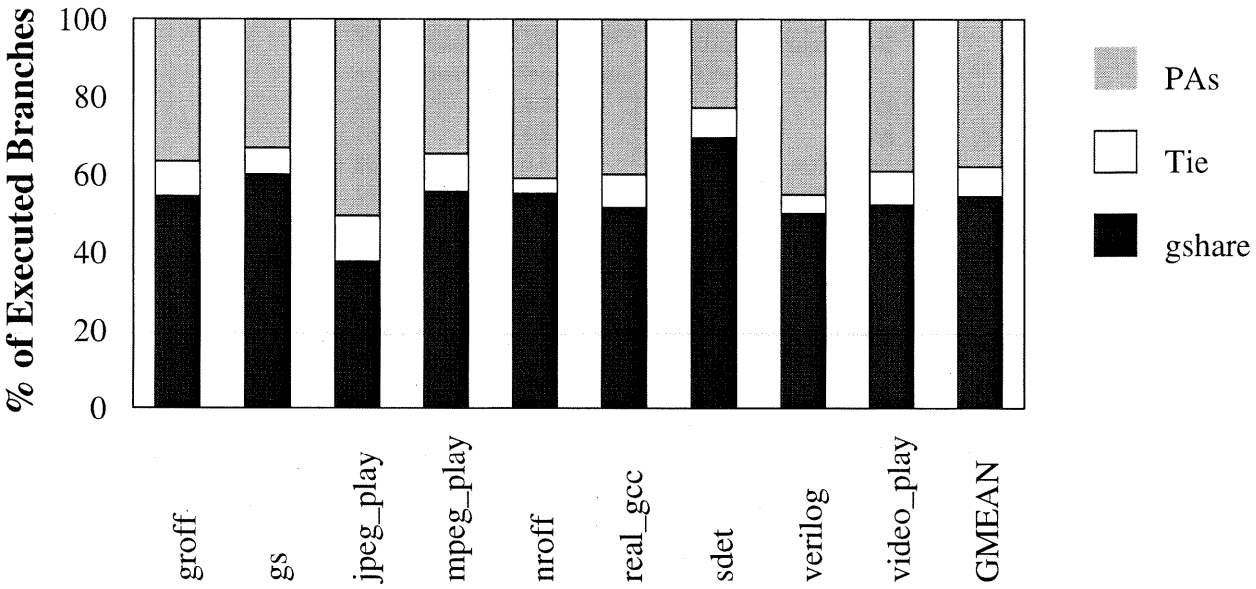


Figure 5: The profiled balance between the component predictors.

comparison was adjusted so as to capture a desired fraction of the dynamic weight. For the purposes of turning the profile into selection bits, the "tie" category was given to gshare. According to our measurements, gshare predictors are less perturbed by having too many branch sites. This is partly because gshare allows aliasing, and hence should have graceful degradation compared to a tagged predictor. However, it is also because gshare has a quite good miss rate when a site indexes a counter last indexed by a different site. Apparently shared gshare indexing patterns tend to be predictive for all the sites which share them.

Figure 6 shows how the McFarling combining predictor views the balance between the component predictors. The disagreement versus the profiled view is not major. On average, the McFarling combining predictor uses gshare 5% more than the profile thinks warranted. This figure bounds any difference which the dynamic selection might make.

5 Results

In this section, we will present simulation data for a number of predictors and policies. We begin by exploring several update policies, and choosing a specific variant of our predictor. We then present the miss rates of that variant, compared to several dynamic predictors. Next, we examine stability, first by a cross-profile study, and then by a cross-validation study. Finally, we examine some convergence properties and the importance of trace length.

The miss rate of our static hybrid predictor is shown in Figure 7 given as a comparison against the McFarling combining predictor. It is important to note that this is not a hardware-normalized comparison,

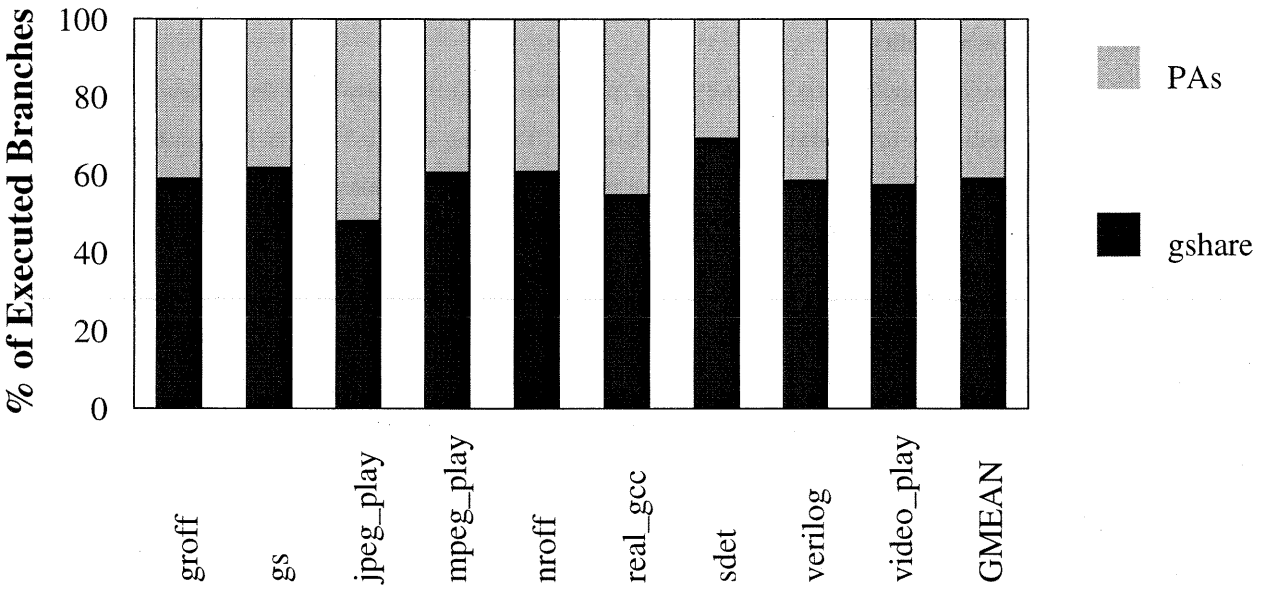


Figure 6: The component balance according to the McFarling combining predictor.

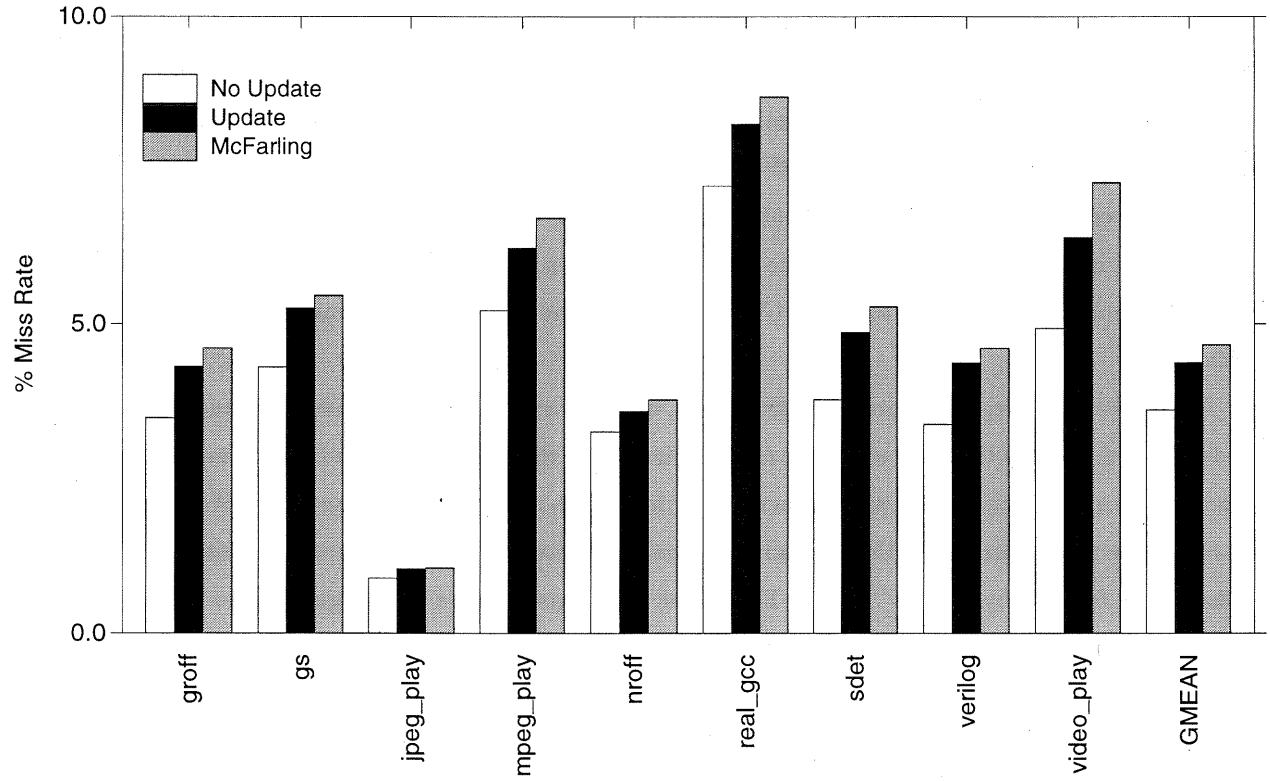


Figure 7: The effect of updating the component predictors.

since the combining predictor uses an additional 2^{11} counters. This means that we are using a comparison that is unfair to our own predictor.

Figure 7 shows two histogram bars for the static predictor. The "update" bar indicates the miss rate, if both component predictors are updated on every branch. The "no update" bar is for a subsystem where update is done only to the component predictor whose prediction is used. This policy is clearly and uniformly better, and it is this policy that is used in subsequent figures. Note that dynamic hybrids are unable to use such a policy. Since they may at any moment begin relying on any given component, they must keep all components "trained" at all times.

The gshare predictor uses a branch history register (BHR), which is simply a shift register containing "taken" booleans for the most recent branches. We tried the policy of not updating this BHR when the gshare component predictor was not updated. This policy turned out to be a bad idea. It would seem that gshare benefits from having this information.

We also tried forcing the use of gshare, whenever the BTB in PAs had an eviction. The theory here would be that PAs is about to make a "cold" prediction, and what could be worse than that? The answer is that gshare is (conditionally) worse than that: the policy is a dis-improvement. We might also note that larger BTBs have correspondingly fewer evictions, hence policies which deal with BTB eviction have diminishing effects with larger predictors.

Figure 8 presents the no-update static hybrid, as compared to five alternatives. Again, note that the McFarling combining predictor uses more hardware. The "gshare, doubled" bar shows the result of instead using only gshare, but twice the size (2^8 by 2^4 instead of 2^7 by 2^4). Similarly, "PAs, doubled" is the Pas component only, with a BTB height of 512 instead of 256. Finally, the rates of the components (operated standalone) is given. The ordering of the bars was chosen so that the geometric mean ("GMEAN") group would be monotonic from left to right. (Note that an arithmetic mean is an incorrect summary of a set of ratios [10].)

Figure 9 is similar, and represents the results when everything has four times as much hardware. The ordering of the bars is the same as in Figure 8, but note that the geometric mean bars are no longer monotonic from left to right. Essentially, the 4x increase in hardware resources has caused the PAs predictor to pull ahead of the gshare predictor, thus changing the balance between the components. In this case, the doubled gshare goes from 2^8 by 2^5 to 2^9 by 2^5 . The doubled PAs goes from a 1024-high BTB to 2048, but remains at 2048 counters.

This change in balance has an effect on the profiling process. To see the extent of the effect, simulations were done with cross-use of the profiles. By this, we mean that the 4 times larger static hybrid was operated in accordance with a profile obtained on the small static hybrid. And, of course, vice versa. The result for the small predictor with the big predictor's profile showed high stability. That is, about one third of its

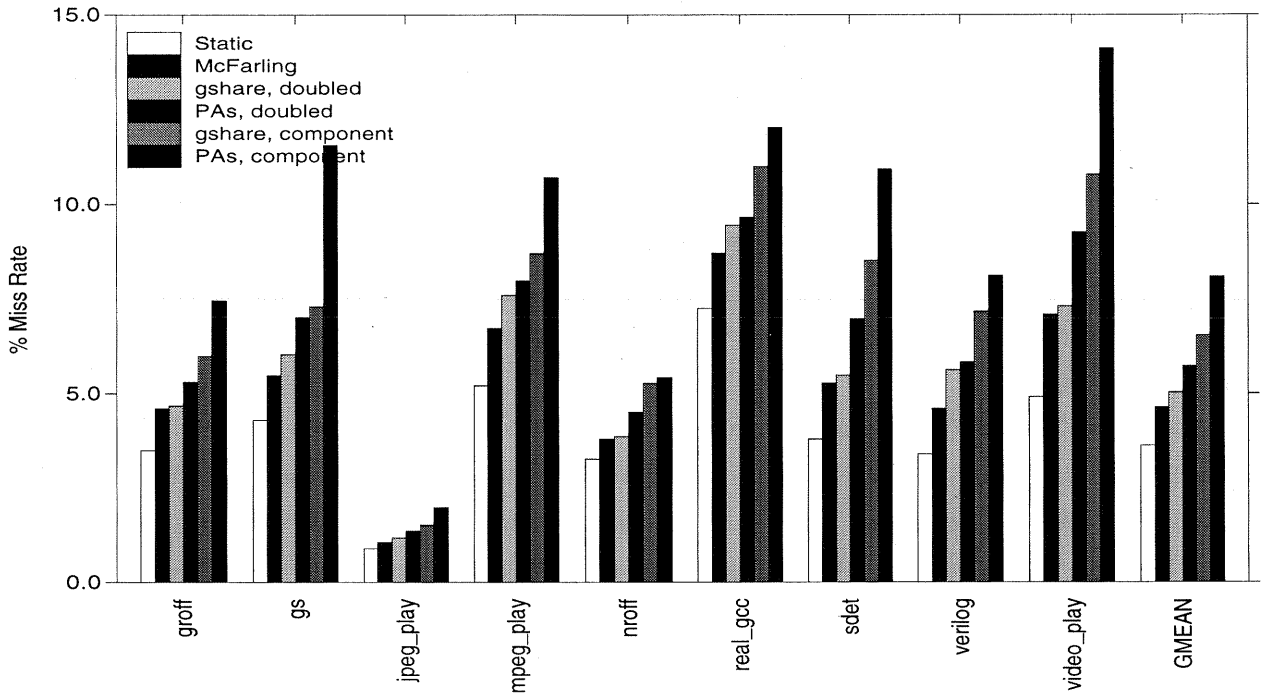


Figure 8: The Static Hybrid predictor versus alternatives.

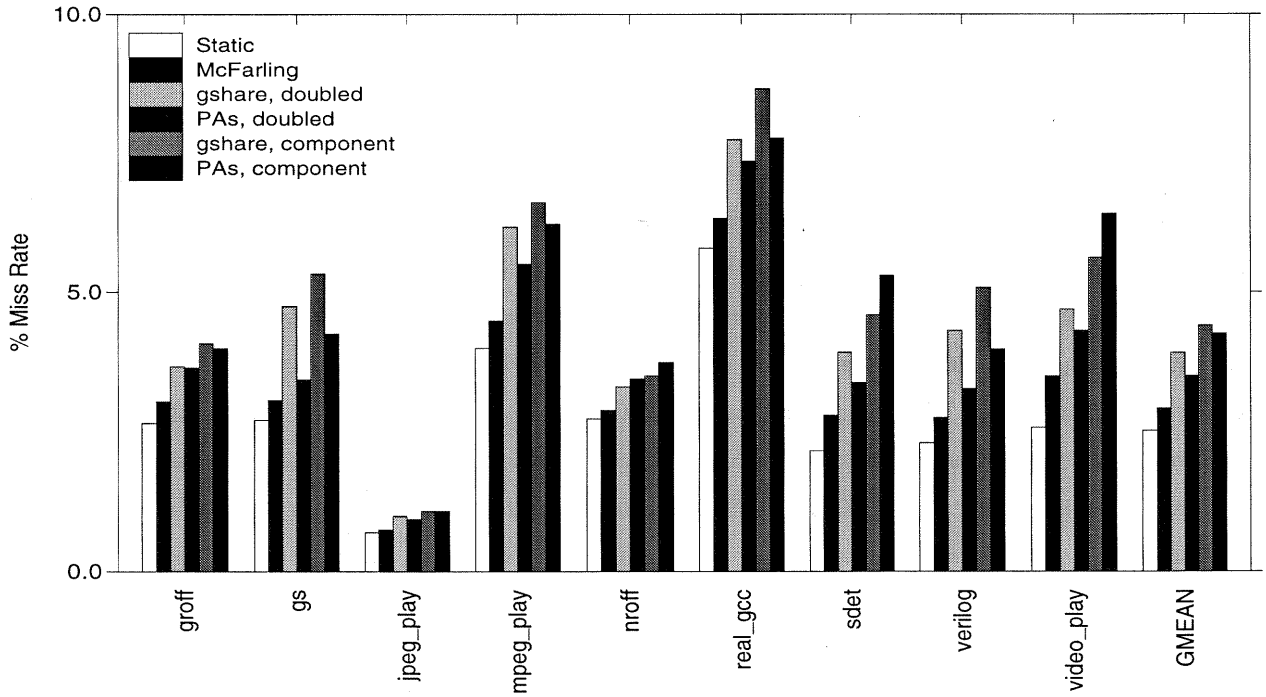


Figure 9: 4x Larger Static Hybrid predictor versus alternatives.

relative advantage (versus the McFarling predictor) went away. The reverse was not true. Running the big predictor with the small one's profile essentially erased the advantage over McFarling. This is not horrible — McFarling is, after all, pretty good, and as stated earlier, the simulated McFarling predictor uses 50% more hardware than our predictor.

It is clear that the large predictor's profile will assign more branch sites to PAs than the smaller predictor's profile would. A smaller PAs, with its shorter BTB, is relatively susceptible to thrashing. We conclude from this that PAs has a performance 'knee', and that component predictors with this property will cause profiles to be at least mildly affected by hardware scaling. However, no major penalty was encountered by any of the benchmarks.

It should be noticed that our experimental procedure involves a training period, followed by an evaluation period. With each trace, we have used the same data for both periods. If our procedure involved (say) a neural net, it would be reasonable to wonder if the procedure was susceptible to "over-training".

Accordingly, we would like to demonstrate that our procedure is stable in the face of previously unseen data. To do this, we divided each trace into two halves. For each trace, we generated two profiles, one from each half of the trace. We then evaluated each of these profiles on the other half-trace. The results of this "cross validation" effort are shown in Figures 10 and Figures 11. By "ab" we mean that training was done on the "a" half of the trace, and evaluation on the "b" half. The prefix "ba" is of course the reverse.

Cross validation causes a technical problem. Our profiling procedure is supposed to assign each branch site to one or the other of the component predictors. However, it is entirely possible that a given branch site will be used in one half of a trace, and not used at all in the other half. This means that a cross-validated evaluation step will encounter unknown sites. The evaluation process should have some policy or heuristic for dealing with these cases. If they were sufficiently rare, we could ignore them. However, they represent 2.4% of the dynamic branch count (as a geometric mean across the 18 half-traces). Worse, there is tremendous variability: unknown sites represent 47% of the branch count in "ba" mpeg_play.

We do not currently have a heuristic for assigning an unprofiled branch site to a predictor. Therefore, the evaluation process was performed three times, with three different policies in force. These policies are: assign unknowns to the PAs component; assign them to the gshare component; and finally, assign them semi-randomly to one component or the other. Figure 10 and Figure 11 show results for all three policies, and for comparison, show results of a McFarling combining predictor operating over the evaluation half-trace.

The figures also show the miss rate of our static hybrid predictor, profiled (only) on the evaluation half-trace. That is, the 18 values represent 9 "aa" values, and 9 "bb" values.

The cross validation figures show more variability than our other figures. However, the geometric means show a high level of consistency. Regardless of the unknown-site policy, cross validated prediction has generally excellent performance, getting a mean miss rate of 3.8%, versus the McFarling figure of 4.6%. Note

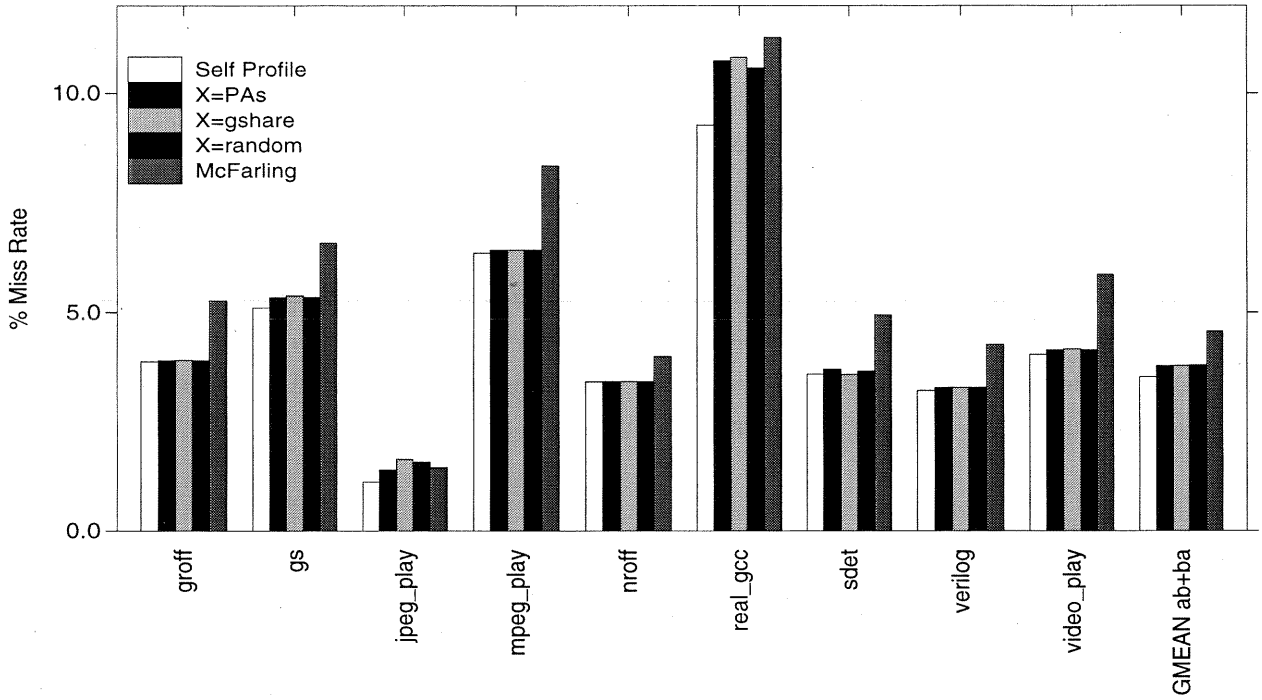


Figure 10: Cross Validation: AB

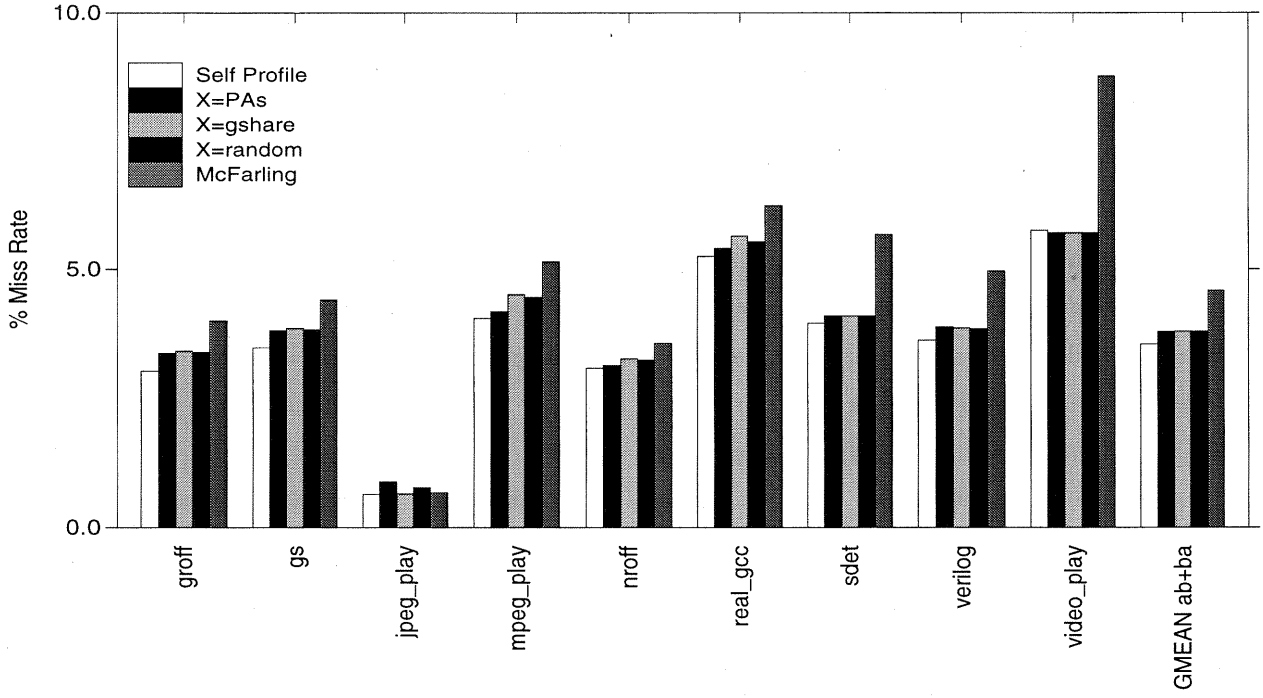


Figure 11: Cross Validation: BA

in particular that the enormous number of unknowns in mpeg_play in Figure 11 did not cause any noticeable problem for our predictor.

This result leads us to conclude that long traces are not required to produce a useful profile. This led us to investigate the effect of trace length. Specifically, we adjusted our simulator so as to produce a profile at any given distance from the front of a trace. A simple metric was applied to each profile, and graphs of this metric were drawn against trace length.

There is not enough space to present these graphs. However, they do consistently show convergence, and broadly speaking have three regimes. The first regime is the major training phase, and may take one million conditional branches. The second regime may take two to four millon branches, and has various smaller irregularities. The third regime is unlikely to have major features.

6 Conclusions

In Section 2 we described hybrid branch prediction, and the role and implementation of a selection mechanism. In Section 3 we introduced one contribution of this paper, which is that the selection mechanism could be performed statically, with a considerable saving in hardware. The approach was explored from the viewpoints of implementation, hardware speed, future extensions, scalability, and of compatability with pipelining. No unsurmountable problems were identified, and some interesting possibilities emerged.

A reasonable range of test hardware configurations was selected in Section 4, and shown to be balanced in Section 4.3. A realistic whole-system benchmark suite was described in Section 4.2.

In Section 5, the configuration was shown to have better performance than a more expensive hardware alternative. This section also introduced the second contribution of this paper, which is that static selection allows static elimination of state updates.

From the presented results, we conclude that static hybrid predictors are quite stable. They are unlikely to exhibit performance breakdowns, they appear scalable, they are able to cope with traces that they were not trained on, and they do not require long training periods. They also exhibit consistently good performance relative to well-regarded dynamic predictors. And, it should be noted, the performance was never worse than that of the more-expensive dynamic predictor.

One of the interesting things about this paper's approach is that it motivates a number of research directions.

As suggested earlier, the approach can be applied to imbalanced components. It can easily be extended to 6 or 8 components, and various interesting cases (such as predication or interrupts) could be catered to.

It may also be possible to define further kinds of static information, which reduce aliasing, increase parallelism, or decrease power consumption. And, all such information could be studied in the context of

compiler heuristics, rather than of profiling. On the other hand, hardware based profiling [4] and modern emulation/cross-compilation tools provide interesting system contexts for the gathering and use of static information.

7 Acknowledgements

The authors would like to thank the Colorado Advanced Software Institute and Hewlett-Packard Company whose award has funded Mr. Lindsay and other students. This work has been made possible by donations of hardware and software from Digital Equipment Corporatation and from Hewlett-Packard Company.

References

[1] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report TR96-06-02, University of Washington, June 1996.

[2] P.-Y. Chang, E. Hao, and Y. Patt. Alternative implementations of hybrid branch predictors. In *28th International Symposium on Microarchitecture*, pages 252–257, 1995.

[3] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *27th International Symposium on Microarchitecture*, November 1994.

[4] Thomas M. Conte, Burzin A. Patel, and J. Stan Cox. Using branch handling hardware to support profile driven optimization. In *27th International Symposium on Microarchitecture*, November 1994.

[5] IBM Corporation. PowerPC 601 microprocessor. Technical report. Online IBM RS/6000 Tech ReSource at http://www.rs6000.ibm.com/resource/technology/601.html.

[6] John H. Edmondson et al. Internal organization of the alpha 21164, a 300 mhz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[7] M. Evers, P.-Y. Chang, and Y. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *23nd Annual International Symposium of Computer Architecture*, pages 3–11, 1996.

[8] N. Gloy, C. Young, J.B. Chen, and M. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *23nd Annual International Symposium of Computer Architecture*, pages 12–21, 1996.

[9] Linley Gwennap. Digital 21264 sets new standard. *Microprocessor Report.* Article available online at http://www.digital.com/semiconductor/microrep/digital2.htm.

[10] Raj Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling.* John Wiley and Sons, Inc., New York, 1991.

[11] Gregg Lesartre and Doug Hunt. PA-8500: The continuing evolution of the PA-8000 family. Technical report, Hewlett-Packard Co. http://hpcc998.external.hp.com/computing/framed/technology/micropro/pa-8500/docs/8500.html.

[12] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[13] Glenford J. Myers and David L. Budde. *The 80960 Microprocessor Architecture.* Intel Corp., 1988. New York: Wiley-Interscience.

[14] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *23nd Annual International Symposium of Computer Architecture*, pages 22–32, 1996.

[15] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *22nd Annual International Symposium of Computer Architecture*, June 1995.

[16] Kenneth C. Yeager. The MIPS R10000 superscalar processor. *IEEE Micro*, pages 28–40, April 1996.

[17] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Annual Symposium on Computer Architecture*, pages 124–134, May 1992.