Evaluating the Potential for Combining Compile-Time and Run-Time Information in the CDG Cache*

Steve Dropsho
dropsho@cs.umass.edu

Charles Weems
weems@cs.umass.edu

Computer Science Department
University of Massachusetts
Amherst, MA 01003-4610

Abstract

We see an opportunity to improve overall performance by intelligently managing both the cache and prefetch using static structural information from program compilation during run-time. Such information enables better use of cache resources and promotes future-based (e.g., dependence graph lookahead) decisions on how to allocate resources rather than history-based (e.g., LRU replacement policy, branch history table). In this paper we outline the benefits of such a system and we verify the essential condition for enabling this approach: the existence of instruction run-lengths in applications that are necessary to support a sufficient level of prefetching. As evidence we present data showing the SPEC95 benchmarks from the persepective of instruction run-lengths.

1 Introduction

Much literature [HS89, GHPS93, RBS94, CB92, Jou90] is devoted to improving cache performance. Most designs rely strictly on run-time information, either from the instructions or from historical behavior such as branch target addresses. Some architectures allow the compiler to supply information or hints in the instructions about the control flow, such as the direction a branch is most likely to take [Int94]. The dynamic stream running on the hardware provides localized data-dependent information that the compiler can't know, but the compiler has global structure information that the hardware is not aware of. With ever-increasing pressure on the cache design to

*This research was supported in part by a grant from the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract number N00014-94-1-0742.

effectively decouple the processor core from off-chip memory, it is natural to consider the potential impact on cache performance if we combine knowledge of the static program structure with the dynamics of a running program.

While the above applies to both instruction and data streams, in this paper we discuss only the instruction stream and outline a cache memory system that we call the **Control Dependence Graph (CDG) Cache**. Some of its interesting qualities include space utilization and hit rates comparable to fully-associative organizations from a direct-mapped RAM, and a cache controller integrated with branch prediction free of aliasing on branch information. This paper presents data from a study to explore whether codes exhibit the behavior to support such a design. The key factor in its feasibility is the size of the run lengths of applications to support the necessary level of prefetching. The run length is the number of instructions between unanticipated changes in control flow. We present a view of the Spec benchmarks in terms of their distribution of run lengths and discuss the implications.

2 Related Work

In microprocessor systems with limited IO bandwidth and the relatively slow but cheap off-chip memories, reducing delays due to memory accesses is essential for peak performance. The idea is to fetch information into cache before the processor needs it and keep it there, as best as possible, if the same code is to be accessed again. A typical microprocessor memory system consists of multiple pieces: cache management policy, cache RAM, branch prediction and target buffer hardware, and, currently, sophisticated instruction prefetch hardware. Each piece has been explored extensively. We present representative work here.

2.1 Cache Management Policy

Cache management hardware controls how data is added to and removed from the cache RAM. Many policies have been explored [HP96]: direct-mapped memory, fully-associative memory, and memory with LRU and other forms of replacement. The simplicity of the direct-mapped cache allows its access time to be faster [Hil88] than an associative cache for a given overall cache size, though performance may actually degrade due to a smaller hit ratio.

2.2 Branch Prediction

Branch prediction hardware attempts to correctly guess the direction a data-dependent branch will follow prior to its condition being resolved. The simplest systems do no prediction of conditional branches. The next level of prediction uses target buffers [CHP97] for *unconditional jumps* and *subroutine calls* where the target address is not available from the instruction binary, but rather a data value in a register. The target buffer is a cache of target addresses typically indexed by the address of the branch instruction.

Some processors accept hints from the compiler in the form of a taken/not taken bit in the instruction encoding [Int94]. The processor always fetches along the path determined by the bit

making this a *static* policy. A common heuristic is the **backward-taken-forward-not (BTFN)** policy that always follows the target path on backward branches and falls through to the next consecutive instruction on forward branches.

In contrast, dynamic branch prediction policies have the capability to alter the branch preference during run-time based on previous behavior of the branch. The **bimodal** predictor is a single level predictor that uses a two-bit saturating counter to determine the prediction path. Every time the branch is taken the counter is incremented up to its maximum value, and when not taken decremented. The **global branch predictor** is a two-level predictor that uses the outcomes of the last N-branches as an index into a table of two-bit saturating counters, based on the expectation that the behavior of a branch correlates with the path of control up to that point. Jouppi and Ranganathan [JR97] compare the performance of both the bimodal and global predictors as well as proposed hybrids. An important result by Chen et al. [CCM96] showed that the global predictor is an approximation to an optimal predictor in data compression. Their empirical studies showed that the approximation is quite good, leaving little room for improvement in the absence of aliasing.

Dynamic predictors typically store state and target addresses in shared tables or caches. To save space no tags are used to disambiguate between shared locations. Thus, totally unrelated branches can modify the same state bits or use unrelated cached targets. This conflict between branches is called aliasing and almost always impacts performance negatively [SLM96, MSU97]. Many of the papers on branch prediction involve schemes to reducing aliasing [ECP96, MSU97, JR97].

2.3 Instruction Prefetch

Guarded Sequential Prefetching [XT96] is a proposed software technique where the compiler gives hints when to prefetch the next sequential block. The compiler sets a guard bit in the last instruction of a prefetch sequence. The processor prefetches along a path until it encounters an instruction with its guard bit set.

The simplest hardware prefetch mechanism is next-line prefetching proposed by Smith [Smi82]. The Markov Prefetcher [JG97] uses a Markov predictor that dynamically learns from past program behavior the most probable cache lines to prefetch. Wrong-Path instruction prefetching [PM96] prefetches the target line into cache when a branch is *not* taken, in the hopes that on a subsequent pass the branch will require the target. These techniques use only run-time information to guide prefetching.

Lee et al. [LBCG95] compare system performance across prefetch strategies with varying degrees of aggressiveness. Their conclusions show that neither very aggressive nor pessimistic prefetch policies are superior. The best prefetch policy depends on system parameters such as latency.

3 CDG Cache

We are exploring ways to improve processor performance by exploiting compile-time information about program structure to more efficiently manage the limited bandwidth and high latency associ-

ated with accessing memory. Our approach is to explore what can be accomplished if the processor has access to the structural information generated by a compiler in conjunction with the dynamic information obtained at run-time. We use two common data structures to represent this information: a **control flow graph (CFG)** [Muc97] for sequencing information and a **control dependence graph (CDG)** [Fer87] to explicitly represent dependencies between basic blocks. These graphs are generated by most compilers for program optimizations, e.g., code motion, loop unrolling, dead code removal, and branch deletion.

A control flow graph explicitly represents possible transitions to and from each code **basic block**, which is a straight-line single-entry, single-exit sequence of code. The control dependence graph is derived from the CFG to represent the essential control flow relationships in a program. Using the CFG and CDG together gives a concise picture of program structure.

Additional information can be associated with each basic block such as virtual address, size, cache residency index and status, and branch history data. The following example illustrates the potential of using this data.

Figure 1 shows the CFG of a simple program and Figure 2 is its corresponding CDG. In this program the start node always jumps to A. Node A optionally branches on TRUE to G near the end or to node AA on FALSE. Nodes B, C, D, E, and F make up a loop with B as its head. The CDG readily shows us what blocks are to be executed given the current program counter. For example, if the FALSE edge out of A is taken then the CDG shows that AA, B, C, and F are all *guaranteed* to be executed and one of the two nodes D or E depending on the branch outcome at C. The loop back on node F shows that on a TRUE condition we loop back to the head and must execute the same group of blocks. (This conditional check at the end of the loop represents the semantics of a *repeat* loop.)

An intelligent cache manager can use this information to its advantage. Once the condition at AA is known a prefetch unit can retrieve blocks B, C, and F directly. The prefetch of F is intriguing in that the processor is able to look beyond the sequential set of branches to key control flow points in the program and has the option of ignoring the resolution of C's branch to fetch F. The processor can also use the block size information and cache residency status to control how blocks D and E are prefetched. The management unit can assess the cost given system parameters and may decide to fetch both blocks if the cost warrants (e.g., they are small relative to access overhead), fetch only the block missing in cache, fetch neither, or fetch the most likely to be accessed or least costly. The additional information enables intelligent, dynamic prefetch and replacement management policies.

3.1 Benefits of the CDG Cache

The CDG provides static structural information about the program. Once it is in place it also provides the opportunity to attach additional dynamically acquired information. For example, adding branch history data to the CDG avoids the branch aliasing that Sechrest et al. [SLM96] show can undercut the potential gains of advanced global correlation prediction schemes for large programs. Of course, this comes at a cost which we address below.

Given that the processor has global information about the program, it now has more options

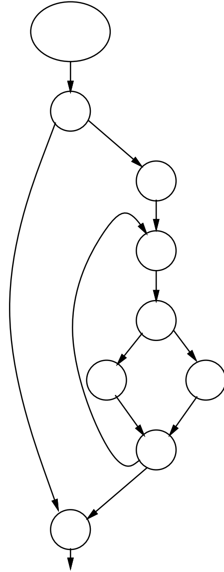

Figure 1: Control Flow Graph (CFG)

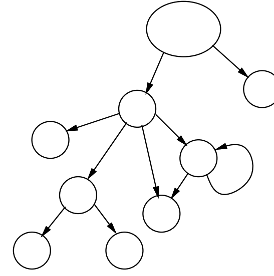

Figure 2: Control Dependence Graph (CDG)

for prefetching effectively. Size information can help in deciding a replacement strategy. For example, loops that do not fit in cache would thrash in a direct-mapped or LRU associative cache as the excessive instructions in an iteration displace earlier references from cache. In very large caches this may not be a problem, but some real-time systems divide the cache into small partitions [Mue95, Kir89] and some of the SPEC95 applications have static paths that are thousands of instructions long. The Fortran application fpppp has a *single* basic block that is 6189 instructions long, almost 25KB. Having access to size information the processor may decide not to encache that entire block in the primary cache so the overall hit rate and performance improves. Such a policy allows peformance to degrade gracefully in larger loops, avoiding a sharp performance drop due to thrashing. These examples are future-based replacement strategies that differ sharply from history-based strategies like LRU.

With the combination of the CDG and dynamic information we can differentiate between fetching for the cache and fetching for the pipeline. When we fetch for the cache, once a loop is completely encached the processor can start fetching beyond the loop in anticipation of its finish. The processor fetches each block in the loop *once*. In contrast, fetching for the pipeline continuously requests each instruction from the cache until the loop terminates and only then does it look beyond the loop. Clearly, decoupling the two types of fetch alleviates latency pressure on fetching from slow off-chip memory and provides time for the prefetch unit to compute its successive accesses.

The CDG Cache looks ahead in the program flow. This allows the fetch unit to run significantly ahead of the pipeline and provides slack that we can use to radically restructure the basic cache design. With reliable run-ahead fetching the processor would be able to afford a multi-cycle, pipelined cache structure that exhibits hit rate behavior similar to a *fully-associative cache while achieving access speeds of a simple, direct-mapped RAM*. Figure 3 shows how pointers can be used to implement this. Note that once the first hit of a basic block is detected, subsequent fetches in the block need not check tags, and can be read from memory as a stream.

When the processor brings in a basic block it gets a RAM location from the cache manager hardware to store the instruction data. The basic block information is updated with the index into the RAM. When fetching an instruction from a block the index is read from the CDG structure and pipelined to the cache RAM read hardware. There is one lookup per block of instructions. This, in effect, implements a mapping table between the basic block in the CDG and the RAM, providing flexibility in block placement. Though the access is easily pipelined, the two-cycle latency is a penalty. Reliable lookahead capability amortizes this cost over the number of references in the stream. Blocks in the cache have pointers back to the CDG structure to reset residency bits upon replacement.

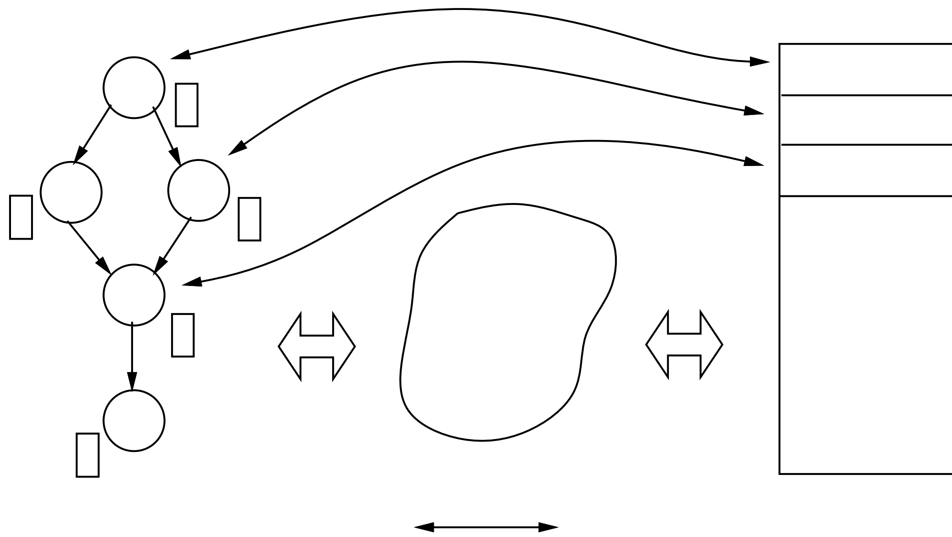

Figure 3: CDG Cache

3.2 Implementation Considerations

We are exploring the cost/benefit tradeoffs of this approach by first estimating the *potential* for processor performance improvement, through simulation. Once we have determined that the potential performance benfits are sufficient, we will consider in detail the implementation cost issues involved in practically building such a design. One issue is the space required to store the program structure information. The CDG cache structure will replace the traditional cache, cache tags, associative matching hardware, branch target buffer, and branch history tables in today's microprocessors. That considerable amount of silicon will be replaced by a direct-mapped RAM, storage and logic for the CDG, and intelligent management logic for the cache. To save space on the CDG implementation we are looking to a hierarchical CDG [Fer87]. Thus, the on-chip CDG storage becomes a cache for the full CDG structure with meta-blocks representing non-local pieces of the application (e.g., whole procedures represented by one node) possibly having enough information to hide the overhead of bringing in its detailed representation. Caching the CDG structure also implies overhead in startup time.

4 Run-length Requirements

Lookahead fetching is only beneficial if the prediction/prefetch logic can consistently and accurately predict long streams of instructions. What constitutes *long* is relative to a particular system's memory access structure. A system with very long access times and small cache will require much longer run-lengths than a system with a memory access time of a few cycles. To decide if we are justified in pursuing the CDG cache we explored the run-lengths of the SPEC95 [SPE96] suite. The results present a novel view of the benchmarks and we believe they justify the approach.

4.1 Experimental Method

On a DEC Alpha [ERP95], we instrument the SPEC95 codes with the shorter test data set and an MPEG encoder application with ATOM [SE94] and measure the run-lengths on a simulator that implements the various prediction strategies. Each branch has its own prediction state, if applicable, so aliasing of target addresses is not an issue as it would be in a design with a fixed size branch target buffer. This removes the negative effects on run-length due to resource limitations [SLM96].

To reduce the size of the recorded output we average run-lengths over intervals of 10,000 blocks (roughly 40K-80K instructions per interval depending on the application). The exceptions are m88ksim and vortex where we averaged over 1,000 block intervals due to their short test cases. An interval of 10,000 blocks averages the run-lengths across regions of code that would likely take less than 1 ms on a typical 200 MHz machine, a fairly small time slice for a process to run. While the averaging can obscure mixes of short and long run-lengths, this would imply that long runs are much longer than the average, providing more time to handle prefetching the interspersed short runs.

Branches are broken down into the following categories:

1. **Unconditional**: Always taken; target known from offset
2. **Return**: Always taken; target predicted from call stack
3. **Branch Subroutine**: Always taken; target known from offset
4. **Jump Subroutine**: Always taken; target determined from offset.
5. **Jump**: Similar to Jump Subroutine.
6. **Jump Subroutine Coroutine**: Always taken; target linked through call stack. (Never used.)
7. **Conditional**: Outcome data-dependent; target known from offset.

Jumps and jump subroutine calls in the Alpha provide a hint in an offset field of the lower 16 address bits of the target. This allows a prefetch to start a cache access. The target address is actually supplied via a register. We presume the compiler knew the jump address if the lower 16 bits match the address of the next instruction executed. If there is a match we consider the jump as being correctly predicted. Most such branches are known; however, in jump tables and function pointers in C the target will not be known.

5 Results

Figures 5 through 22 plot the different applications. There are four data sets per plot. One is most applicable to real-time where the need for predictable timing dictates that the fetch unit stalls on all unconditional branches, jumps, and jump subroutines (whose targets are unavailable on the Alpha), called **No Predict**. Another follows all branches where the target is known at compile time, i.e., most jumps and jump subroutines, called **Known Target**. A third adds the capability of statically predicting backward branches taken and falling through on forward branches, called **BTFN**. The final scheme is the **bimodal** technique using two-bit saturating counters. Note that some plots are have a logarithmic y-axis to show the complete range of values.

The first observation is that more sophisticated schemes improve on simpler ones as expected. What is interesting is the generally large increase in run-length when going to the bimodal technique. Figure 5 demonstrates the clear superiority of the bimodal predictor on the MPEG application with run-lengths consistently over 100 instructions. Dips in the plots of predictors show regions of code with (relatively) many branches that vary between taken and not taken. Dips in static predictor plots that correspond with peaks in the bimodal plot are regions of many forward branches that are consistently taken. Regions where the static and bimodal schemes match have code that doesn't exploit the potential of the better prediction scheme, e.g., few branches or highly unpredictable branches.

Another observation is that certain codes have distinct phases. MPEG shows three distinct regions. The first region processes user input parameters- command line flags and setup file information- and has many IF clauses. The second region computes a fast discrete cosine transform (DCT) via a sizable loop of branch-less code. In this region the run-length is over 1000 instructions using the two-bit branch predictor. The third region does the post-DCT computation of quantization and RLE Huffman encoding. While the loops in this region are large there are many IF clauses that conditionally run code depending on out of range values and arithmetic sign. Static predictors do poorly under these conditions and the run-lengths average around 10 instructions in the first and last regions. The two-bit dynamic predictor consistently has run-lengths over 100 instructions.

Figure 4 lists the overall run-length averages for the applications. Across the benchmarks the bimodal predictor experiences run-lengths over 49 instructions long and often over 100. Of the benchmarks tested the MPEG encoding phase, compress, perl, ijpeg, applu, apsi, fpppp, mgrid, swim, turb3d, and wave5 had average run-lengths near 100 or above. While go, li, m88ksim, vortex, hydro2d, su2cor, and tomcatv had run-lengths ranging from 50 instructions to 100. We find these results encouraging and feel they justify continued research on the CDG Cache design.

6 Conclusion

In this paper we propose research with a radically different approach to instrution cache design, the CDG Cache, to explore the full potential of combining static information generated at compile time with run-time information. Previous work has limited integration of these two types of information, but much greater integration poses interesting possibilities. We believe a CDG Cache offers unique

Application	No Predict	Known Targets	BTFN	Bimodal
su2cor	5.9	6.2	8.0	49.3
go	7.1	8.0	16.7	51.8
li	5.4	7.6	11.7	58.9
tomcatv	6.0	6.5	8.3	59.5
hydro2d	6.2	6.9	9.3	61.9
vortex	6.4	8.3	12.5	78.6
m88ksim	7.1	8.4	11.4	78.7
perl	6.1	7.3	11.3	95.6
applu	29.7	29.9	42.8	117.7
ijpeg	14.3	16.1	28.0	129.5
mpeg	10.8	13.1	20.4	142.1
apsi	21.8	24.7	35.4	304.9
turb3d	32.4	34.1	41.7	377.6
compress95	14.7	17.7	21.4	393.4
fpppp	73.9	83.7	128.3	652.0
wave5	12.6	18.1	27.0	1756.3
mgrid	68.8	69.2	72.0	2104.7
swim	35.9	54.2	54.8	4779.4

Figure 4: Overall Average Instruction Run-Lengths

advantages such as hit ratio levels of a fully-associative cache from a direct-mapped RAM, using global information to guide prefetching, and the potential for *future-based* replacement strategies rather than the current *history-based* approach. The overhead incurred in such a structure requires that programs have relatively long instruction run-lengths to amortize the costs. We have observed that the SPEC95 benchmarks and MPEG satisfy that condition.

7 Future Work

We are completing a simulator to further test the ideas presented here. Much work is to be done in understanding the variety of dynamic conditions that occur and what policies to apply. The additional information in the CDG allows us to subclass situations and possibly manage each one in a unique way. Examples include small vs. large loops, small vs. large conditional blocks, and nested conditionals. Also, we are exploring implementation issues to make a CDG Cache design, or some variation, a practical alternative to traditional designs. Of course, this work will be expanded to data accesses by extending the CDG cache to a Program Dependence Graph (PDG) Cache that includes data dependences. Finally, our approach evolved from a study on adapting RISC architectures for use in real-time by minimizing variance in execution time. We feel such a design can be used to improve predictability in processors while still providing good performance, and will continue to explore this aspect of the research.

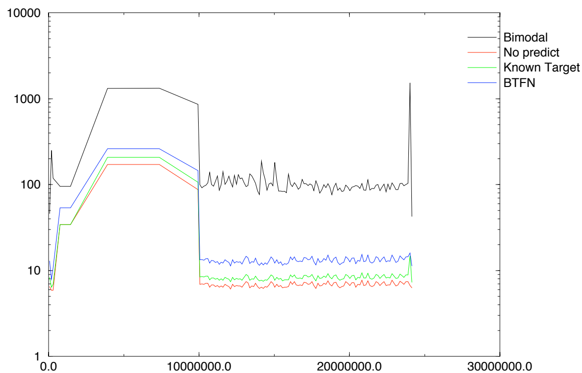

Figure 5: MPEG

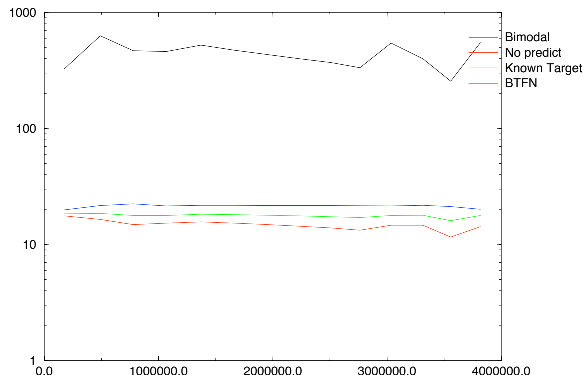

Figure 6: COMPRESS95

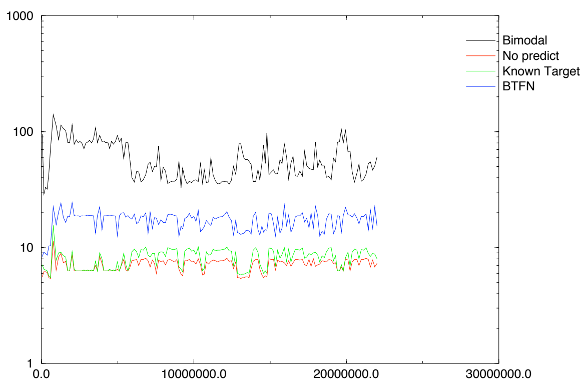

Figure 7: GO

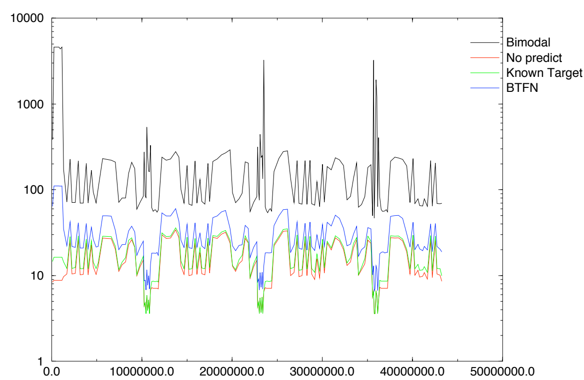

Figure 8: IJPEG

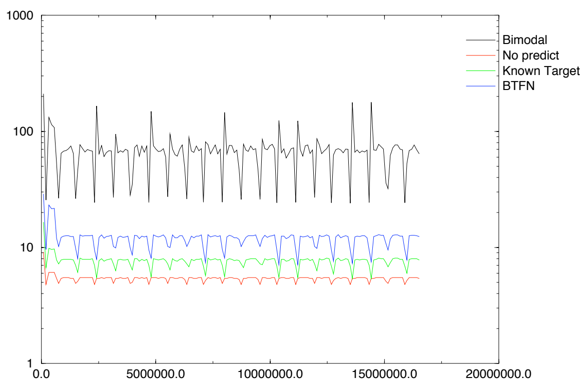

Figure 9: LI

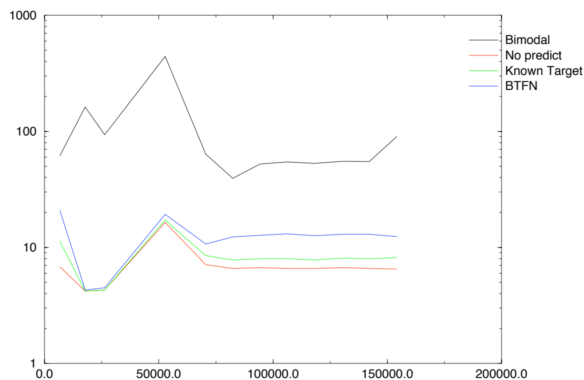

Figure 10: M88KSIM

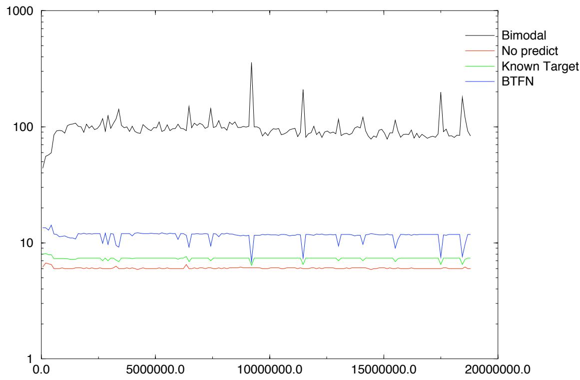

Figure 11: PERL

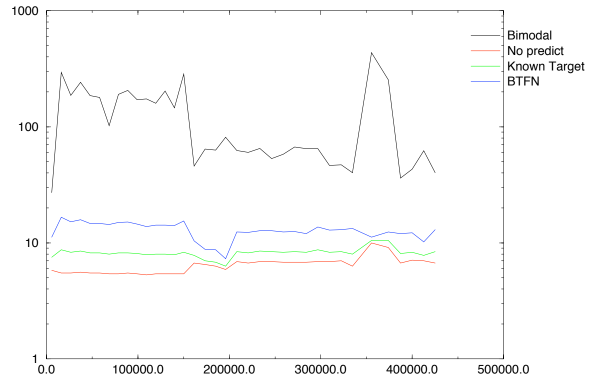

Figure 12: VORTEX

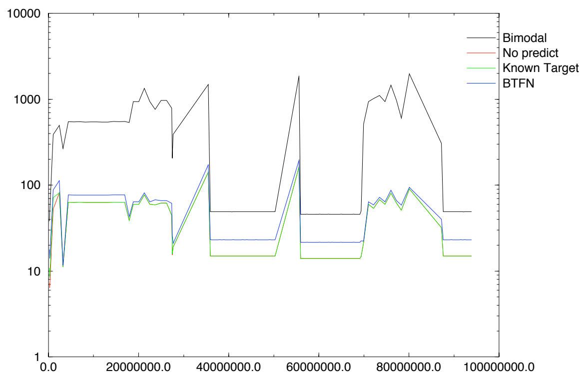

Figure 13: APPLU

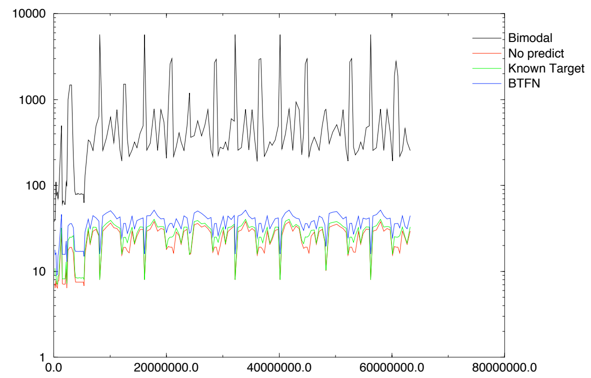

Figure 14: APSI

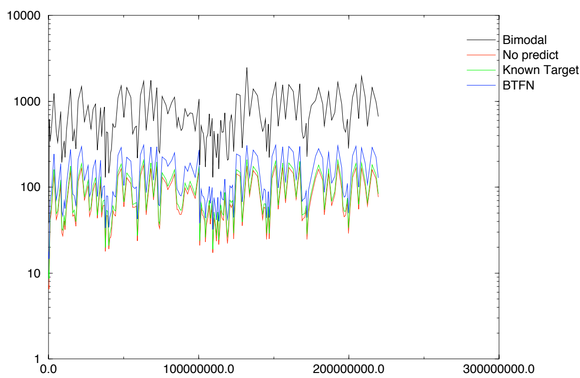

Figure 15: FPPPP

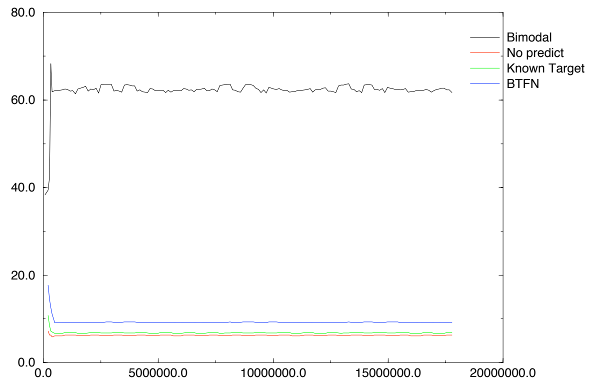

Figure 16: HYDRO2D

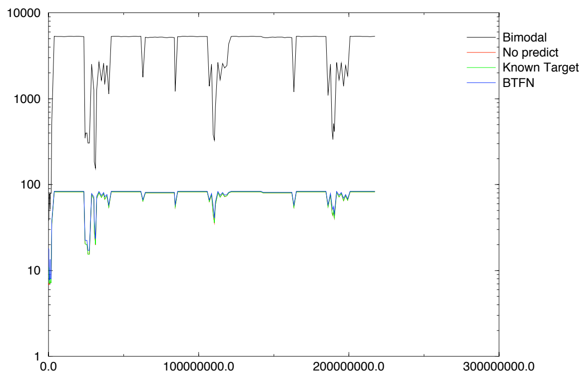

Figure 17: MGRID

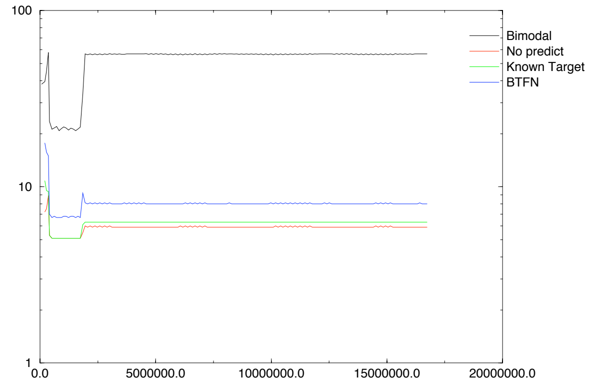

Figure 18: SU2COR

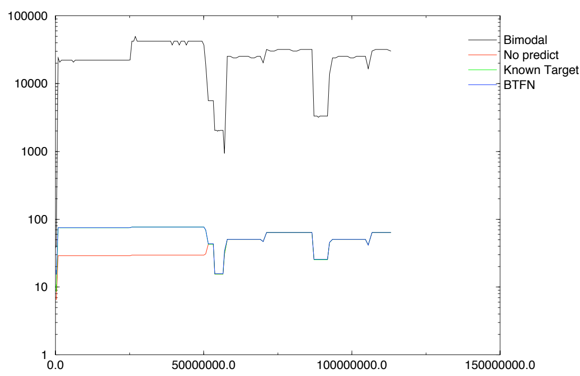

Figure 19: SWIM

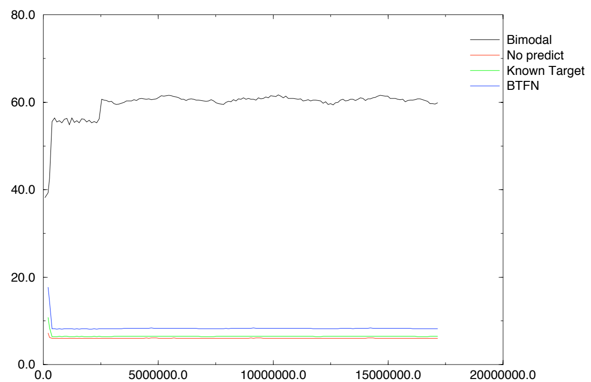

Figure 20: TOMCATV

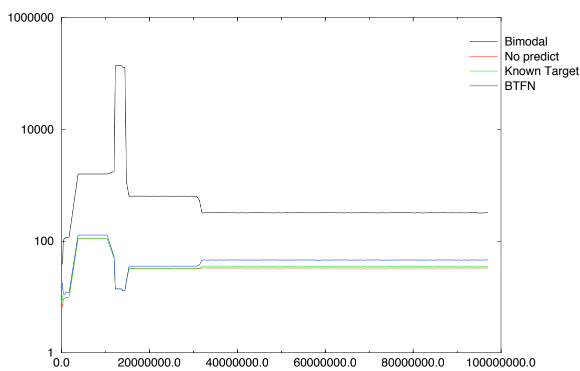

Figure 21: TURB3D

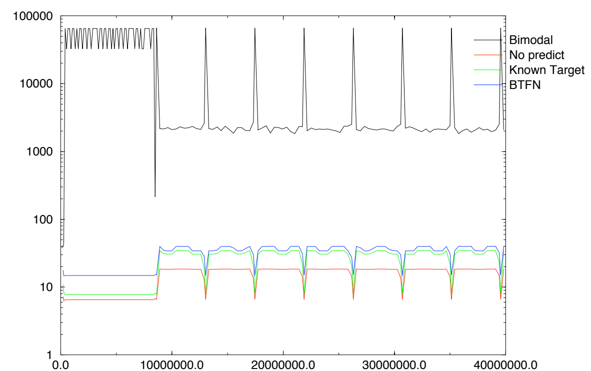

Figure 22: WAVE5

References

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. *ASPLOS*, pages 51–61, October 1992.

[CCM96] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of Branch Prediction via Data Compression. *ASPLOS VII*, pages 128–137, October 1996.

[CHP97] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target Prediction for Indirect Jumps. *ISCA*, pages 274–283, June 1997.

[ECP96] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. *ISCA*, pages 3–11, May 1996.

[ERP95] John H. Edmondson, Paul Rubinfeld, and Ronal Preston. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, pages 33–43, April 1995.

[Fer87] Jeane Ferrante. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[GHPS93] Jeffery D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro*, pages 17–27, 1993.

[Hil88] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, pages 25–40, December 1988.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[HS89] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, pages 1612–1630, December 1989.

[Int94] SPARC International, editor. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., 1994.

[JG97] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. *ISCA*, pages 252–263, 1997.

[Jou90] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ISCA*, pages 364–373, May 1990.

[JR97] Norman Jouppi and Parthasarathy Ranganathan. The Relative Importance of Memory Latency, Bandwidth, and Branch Limits to Performance. *Proceedings of the Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, pages 284–291, June 1997.

[Kir89] D.B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.

[LBCG95] Dennis Lee, Jean-Loup Baer, Brad Calder, and Dirk Grunwald. Instruction Cache Fetch Policies for Speculative Execution. *ISCA*, pages 357–367, 1995.

[MSU97] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *ISCA*, pages 292–303, June 1997.

[Muc97] Steven S. Muchnick, editor. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Mue95] Frank Mueller. Compiler Support for Software-Based Cache Partitioning. *ACM Sigplan Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 125–133, June 1995.

[PM96] Jim Pierce and Trevor Mudge. Wrong-Path Instruction Prefetching. *MICRO-29*, pages 165–175, December 1996.

[RBS94] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. Technical Report CS-TR-96-1310, University of Wisconsin- Madison, April 1994.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. Technical report, Digital Western Research Laboratory, 1994.

[SLM96] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and Aliasing in Dynamic Branch Predictors. *ISCA*, pages 22–33, May 1996.

[Smi82] A. J. Smith. Cache Memories. *Computing Surveys*, pages 473–530, September 1982.

[SPE96] SPEC. SPEC. Technical report, Standard Performance Evaluation Corporation, http://www.specbench.org, 1996.

[XT96] Chun Xia and Josep Torrellas. Instruction Prefetching of Systems Codes With Layout Optimized for Reduced Cache Misses. *ISCA*, pages 271–282, 1996.