The Temporal Ancestry Prefetcher

Nathan Gober*, Gino Chacon*, Daniel Jiménez†, and Paul V. Gratz*
* Department of Electrical and Computer Engineering
† Department of Computer Science and Engineering
Texas A&M University, College Station, Texas
{ngober, ginochacon, djimenez, pgratz}@tamu.edu

***Abstract*—Typical instruction prefetchers model the control flow graph of a program, looking ahead through the control flow graph to prefetch likely-used blocks. Blocks in the instruction cache, however, tend to have long dead times. Thus, looking ahead deeply into the control flow should be desirable and moving stepwise through the control flow graph may be sub-optimal, since it requires many iterative steps. Iterating only through misses that are descended from a particular instruction block provides strong potential to pass by long chains of instruction hits on the path through the control flow graph.**

This paper introduces the Temporal Ancestry Prefetcher (TAP), a new prefetcher designed for instruction caches. TAP approximates the transitive closure of a program's control flow graph. The solution performs 23% better than no prefetching and outperforms the next-line prefetcher by a wide margin in server workloads. In addition, it is simple to tune and can be adapted to a variety of hardware budgets.

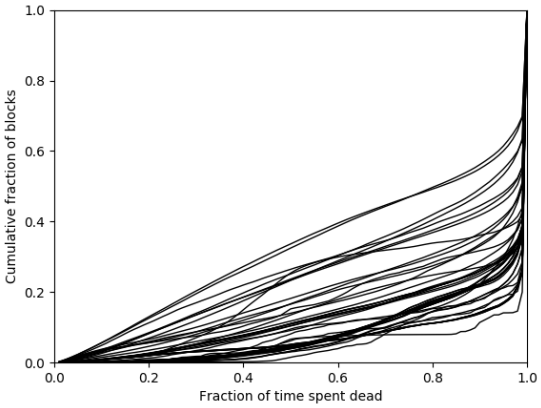

Fig. 1. Fraction of time spent dead by IPC1 traces, plotted cumulatively. In this plot, a higher curve is better, because it indicates blocks in the cache are evicted shortly after becoming dead.

I. Introduction

Graphs, including control flow graphs (CFGs), are often thought of in terms of an adjacency matrix. In an adjacency matrix M, the element M_{ij} equals the edge weight if there is an edge connecting i and j, or the zero weight if there is no such edge.

The transitive closure of a matrix M is

$$A = \sum_{i=1}^{\infty} M^i \tag{1}$$

The transitive closure of an adjacency matrix represents the reachability of two nodes, that is, whether a path exists between a node v and a node u including, if the edges are weighted, the weight of the path between them. For a weighted CFG, the ancestry matrix contains weights that describe the likelihood that v succeeds u in the control flow. In this work, we seek to build a structure that approximates and leverages this likelihood.

The reachability metric in directed graphs is well-studied. The naïve algorithm performs a $O(|V|^3)$ depth-first search for all nodes, [1]. The algorithm can theoretically be improved to $O(w(|V|)(|V|\log|V|)^{\frac{4}{3}})$ in the random case, noting that the typical graph is strongly connected [2].

Here, we describe A as the "ancestry matrix". The ancestry matrix of a random directed graph is known to be strongly connected with high probability, but, in addition, CFGs show a high degree of structure, including dominance of one node over another when a node v is present in each of the many paths descending from u [2], [3]. Under these assumptions, the complexity can achieve $O(|E|)$ time [4].

Temporal prefetching has been explored in the field of data prefetching, both at the L1 level and below [5]–[7]. The primary drawback of temporal prefetching, it has been noted, is that it requires a significant amount of metadata, on the order of megabytes, which often is stored off-chip [8], [9]. Much of the recent work in temporal prefetching has focused on mechanisms to reduce the off-chip state or manage its movement on chip [5], [6].

Instruction prefetching has been studied since early out-of-order designs, where next-line prefetchers were found to improve performance [10]. Other prefetchers have been proposed, many of them variations on next-line prefetchers, such as next-N and stride prefetchers [11], [12]. Many designs depend on the branch

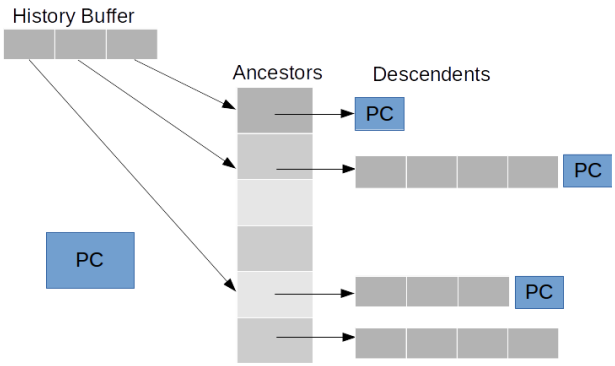

Fig. 2. The function of the ancestry table on a cache miss

predictor or return stack for their accuracy and performance [13]. Some prior works have leveraged temporal prefetching for instruction steams, however these approaches require significant L2 space [7]. In many commercial workloads, the L1I is a bottleneck, causing significant performance loss [14]–[16], thus developing high performance L1I prefetchers is highly desirable.

The remainder of this paper is organized as follows: Section II discusses the potential for deep prefetching in CFGs and its part in the design of the Temporal Ancestry Prefetcher, and Section III evaluates its performance relative to no prefetching and next-line prefetching.

II. Design

A cache block is considered "dead" if it will not be re-referenced before its eviction. Figure 1 plots the liveliness of all instruction blocks in the fifty public traces for the First Instruction Prefetching Competition. It shows the fraction of blocks as a function of how much of their lifetime is spent dead, plotted cumulatively. Traces for which blocks tend to have long dead times will have lower curves in this graph.

We observe that, in an instruction cache, blocks spend much of their lifetime dead, in fact, in most traces, over 80% of blocks are dead for half of their lifetime or longer and the majority of blocks are dead on arrival into the cache.

The length of dead times implies that there is significant room in the cache for prefetches to live. Even if prefetches are not useful, it is highly unlikely that the prefetch will evict a block that is alive.

Next-line prefetching is a strong first step towards this. Next-line prefetching will often fetch blocks that will eventually be useful and that will be long-lived in the cache. However, it is not sensitive to the control flow of the program. To augment next-line prefetching, we introduce the Temporal Ancestry Prefetcher (TAP), which runs alongside next-line prefetching and which looks deeply beyond difficult-to-predict branches for prefetch candidates.

A. Temporal Prefetching

Temporal prefetching attempts to cover future cache misses by replaying old cache misses. In TAP, we approximate the ancestry matrix in the ancestry table by including old misses as descendents of addresses.

TAP maintains a history of the last 14 PC values. This value was empirically chosen to produce the highest performance on our traces. On a miss, TAP visits the rows of the ancestry table that correspond to each of these history values and adds the current miss address as a descendent, inserting it if it is absent or incrementing its weight if it is present. This function is shown in Figure 2. The ancestry table, then, focuses on replaying the misses along arbitrary paths, without concern for the hits that may occur along the path or the particular path that is followed. Because TAP runs alongside next-line prefetching, we never add the next line to the ancestry table, entrusting that prefetch to the next-line prefetcher.

On any cache access, the row of the ancestry table corresponding to the current PC is checked and each of the descendents, as well as the next line of the cache access, are prefetched. The operation is recursively performed on each of the prefetches in order to anticipate future path accesses. Recursion terminates once the confidence, defined as the path product of the item weights (normalized within the row), drops below a threshold. As an optimization, recursion terminates if the row of the ancestry table has already been visited. Because blocks in the instruction cache tend to be long-lived, the confidence threshold can be very small.

To prevent excess prefetching, the threshold is scaled by the current accuracy of the prefetcher. As the prefetcher performs more accurately, the threshold is scaled to promote deeper prefetching. To temper the weights and prevent them from saturating, the weights are decremented whenever a prefetched block is evicted from the cache without becoming useful.

B. The Ancestry Table

The primary mechanism in TAP is the ancestry table, which correlates cache blocks with upcoming misses. The anticipated upcoming misses are present in the rows of the table with their corresponding weight, incremented whenever the miss occurs again.

The table is organized into 2^{12} rows with up to 9 descendents, replaced on a low-confidence basis. If the CFG is sparse, or contains many low-likelihood entries, this will be a strong approximation that can easily be updated when a program's phase changes. The large number of rows also allows the ancestry table to be designed in a way that permits multiple parallel accesses.

TABLE I
METADATA COST OF TAP

Structure	Entry		Total
Ancestry Table	$2^{12} * 9$	Valid (1 bit) PTB Index (11 bits) Page Offset (6 bits) Weight (6 bits)	113 KiB
Visited Flags	2^{12}	1 bit	0.5 KiB
Page Translation Buffer	2^{11}	Valid (1 bit) NRU (1 bit) Page (52 bits)	13.5 KiB
History Buffer	14	56 bits	98 B
Shadow Cache	512	Valid (1 bit) Useful (1 bit) Prefetch (1 bit) Tag (12 bits)	960 B
Accuracy Counters	2	11 bits	22 bits
Total			123.227 KiB

TABLE II
MACHINE PARAMETERS FOR CHAMPSSIM MODEL

Core	1 core, 4 GHz, 6-wide, 352-entry ROB
Branch Predictor	Hashed Perceptron, 16 tables
L1 Instruction Cache	32 KiB, 8-way, LRU, 8 MSHRs, 4-cycle latency
L1 Data Cache	48 KiB, 12-way, LRU, 16 MSHRs, 5-cycle latency Next-Line Prefetcher
Unified L2 Cache	512 KiB, 8-way, LRU, 32 MSHRs, 10-cycle latency Signature Path Prefetcher
Shared L3 Cache	2 MiB, 16-way, LRU, 64 MSHRs, 20-cycle latency No Prefetcher
DRAM Memory	4 GiB, 1 channel, 3200 MT/s

C. Shadow Caching

The TAP mechanism produces many prefetch hits because of the long-livedness of blocks in the L1I. In order to conserve space in the prefetch queue, we implement an approximation of the cache's tag dictionary, which we term the shadow cache. The shadow cache stores 12 bits of tag, enough to examine the tags cheaply and quickly with high accuracy. The partial tags have more than 99% fidelity to the actual tags in the cache. Because the shadow cache is small in size, no more than a kilobyte, the structure can be duplicated easily, allowing many accesses per cycle.

The shadow cache also maintains metadata on whether a block in the cache is a prefetch and if the prefetch is useful. This information is used to train the prefetcher by decrementing weights on a useless prefetch.

D. Page Compression

The instruction footprint of programs tends to be smaller than its data footprint. Figure 3 shows our observation that no more than 2^{11} pages are used in total in the traces used to test our design. We can compress pages by adding them to a page translation buffer and use the index to this buffer as a proxy for the page number. The size of this buffer is 2^{11} entries, which holds all pages used in the traces, so the index is 11 bits, resulting in a high degree of metadata compression in the ancestry table. This approach is similar to the ISB proposed by Jain and Lin [6], but without the intent to order the pages in time. The page translation buffer is associative with NRU replacement. As submitted, the page translation buffer is fully associative, but our tests show that the performance is not significantly sensitive to the associativity of this structure.

E. Implementation

The metadata cost of TAP is listed in Table I. TAP uses 123.227 KiB of metadata, most of which is in the ancestry table. The ancestry table size is highly flexible. The number of descendents and ancestors in the table can be varied to suit many different hardware budgets and can be tuned easily. This flexibility makes TAP a very feasibly constructible system. The page compression scheme used produces a considerable improvement in the metadata size due to the small instruction footprint of the traces, leading to a metadata budget that is considerably less than the budgets of other temporal prefetchers.

III. RESULTS

Our design was tested on the ChampSim simulator in accordance with the configuration for the First Instruction Prefetching Competition. ChampSim is a trace-based cache simulator designed for prefetcher evaluation. It models an out-of-order processor (possibly multicore) with 3 cache levels. The L2 cache is a private, unified instruction and data cache, and the L3 is shared. The machine parameters for the competition are specified in Table II

The simulations were evaluated on 50 public traces, including 8 client workloads, 35 server workloads, and 7 SPEC benchmarks. Simulations were warmed up for 50 million instructions and evaluated for an additional 50 million, repeating the trace if its end was reached. The prefetcher was designed to run entirely in the virtual address space, though the design does not prohibit use in the physical address space.

The prefetcher was evaluated against the next-line prefetcher in terms of speedup over no prefetching Figure 4 shows the performance of both prefetchers. TAP achieves a geometric mean 23% speedup over no prefetching, outperforming or matching the next-line prefetcher except in one case, the first simpoint

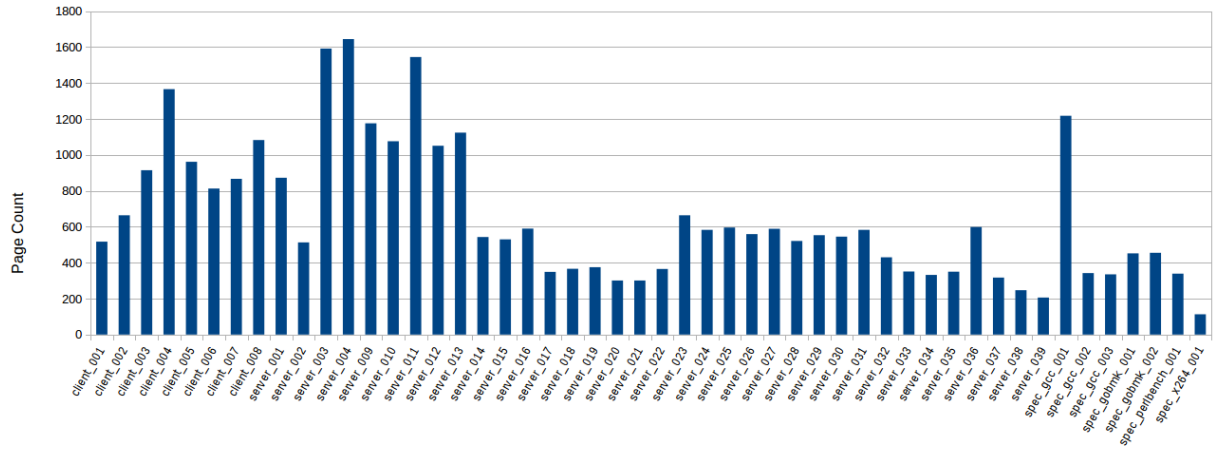

Fig. 3. The page count of the public IPC1 traces. Traces with large instruction footprints are more difficult to prefetch and offer less potential for address compression.

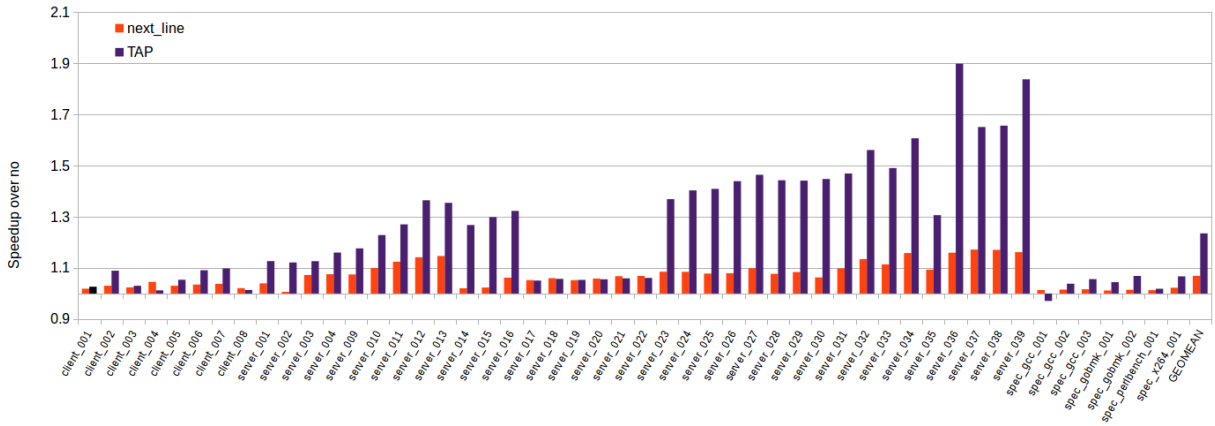

Fig. 4. The speedup of next_line and TAP over no prefetching. TAP achieves a 23% improvement over no prefetching and strongly outperforms next_line on server workloads.

of the SPEC `gcc` benchmark. In particular, TAP performs well on server workloads, but struggles to get significant improvement on client and SPEC workloads. These workloads are well-studied in the domain of data prefetching, and they find their bottlenecks in the data portion of the processor. As a result, because our so improved instruction prefetching shows only mild gains in general.

IV. Conclusion

The Temporal Ancestry Prefetcher models the transitive closure of the control flow graph's adjacency matrix. Rather than attempt to model each step along the path, TAP focuses on blocks that will be eventually reached, without concern for how soon the block will be used. In TAP, it is more significant that a block will be eventually reached than that a particular path is taken to arrive there. This technique is justified by the long dead times of blocks in the L1 instruction cache. Ultimately, it results in a 23% improvement over no prefetching, with particularly strong performance in server workloads.

Acknowledgements

This paper is the result of research sponsored by the National Science Foundation through grants CCF-1912617, I/UCRC-1439722 and CCF-1823403, a contract from the Semiconductor Research Corporation, and generous gifts from Intel Corporation. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

REFERENCES

[1] R. Jin, N. Ruan, Y. Xiang, and H. Wang, "Path-tree: An efficient reachability indexing scheme for large directed graphs," *ACM Transactions on Database Systems (TODS)*, January 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/path-tree-an-efficient-reachability-indexing-scheme-for-large-directed-graphs/

[2] R. M. Karp, "The transitive closure of a random digraph," *Random Structures & Algorithms*, vol. 1, no. 1, pp. 73–93, 1990. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.3240010106

[3] M. Weiss, "The transitive closure of control dependence: The iterated join," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 2, p. 178190, Jun. 1992. [Online]. Available: https://doi.org/10.1145/151333.151337

[4] G. Bilardi and K. Pingali, "A framework for generalized control dependence," *SIGPLAN Not.*, vol. 31, no. 5, p. 291300, May 1996. [Online]. Available: https://doi.org/10.1145/249069.231435

[5] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 131–142.

[6] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 247259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[7] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 1–10.

[8] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA 05. USA: IEEE Computer Society, 2005, p. 222233. [Online]. Available: https://doi.org/10.1109/ISCA.2005.50

[9] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 79–90.

[10] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS 93. New York, NY, USA: Association for Computing Machinery, 1993, p. 6776. [Online]. Available: https://doi.org/10.1145/165939.165956

[11] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.

[12] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 371–382.

[13] A. Kolli, A. Saidi, and T. F. Wenisch, "RDIP: Return-address-stack directed instruction prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 260–271.

[14] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 3–14.

[15] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh, "An analysis of database workload performance on simultaneous multithreaded processors," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 39–50.

[16] L. Spracklen, Yuan Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 225–236.

[17] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel