The Case for Domain-Specialized Branch Predictors for Graph-Processing

Ahmed Samara and James Tuck

Abstract—Branch prediction is believed by many to be a solved problem, with state-of-the-art predictors achieving near-perfect prediction for many programs. In this article, we conduct a detailed simulation of graph-processing workloads in the GAPBS benchmark suite and show that branch mispredictions occur frequently and are still a large limitation on performance in key graph-processing applications. We provide a detailed analysis of which branches are mispredicting and show that a few key branches are the main source of performance degradation across the graph-processing benchmarks we looked at. We also propose a few ideas for future work to improve branch prediction accuracy on graph workloads.

Index Terms—Graph-processing, branch prediction

1 Introduction

Graphs are a common data structure used to represent social networks, roads, knowledge, and a variety of other important data. Graph analytics and graph-processing have emerged as critical workloads due to their growing use in big-data analytics, machine learning, databases, web services, and a wide variety of other application domains. The need for higher performance on these workloads has led many companies to build custom hardware for efficiently running graph-processing algorithms [1], [2], [3], [4], [5].

Previous works have identified the memory-hierarchy as a critical bottleneck, especially because the graphs that represent meaningful real-world data are often bigger than on-chip cache hierarchies and occasionally even bigger than the memory of a single node [6], [7]. It is not surprising that this bottleneck was tackled first. Basak *et al.*[7] point out the poor locality of these workloads and design a graph-aware memory prefetcher that knows the layout of the graph data structure to prefetch nodes and properties before they are needed. Others, such as ReCALL, have attempted to solve the caching issues by changing the labels of the graph to allow for better access locality [8]. However, orthogonal to improving cache performance, there is still a lot more performance to be had by improving branch prediction.

In Fig. 1, we compare the performance of different branch predictors on several graph-processing workloads, with simulations of perfect and realistic caches. The branch predictors shown are a one-bit history predictor (the simplest possible predictor), a Pentium M predictor (which is an approximation of the current state-of-the-art predictor), and a perfect predictor which always predicts correctly.

With a realistic cache, achieving perfect branch prediction gives a 16-41 percent IPC improvement over the pentium M predictor. When the cache is perfect, achieving perfect branch prediction yields a significant 57-132 percent improvement in performance over the next best predictor. Notably, the performance difference between the one-bit predictor (which is the simplest possible predictor), and a state-of-the-art predictor is minimal (at most 6 percent difference in IPC), as shown in Fig. 1.

It's likely that branch prediction has not been previously noted as an area for improvement because the potential benefits are masked by issues in the memory hierarchy. As support for graphs advances in the memory hierarchy, branch mispredictions will become a more significant hindrance to performance. While other works have previously noted that branch predictors perform poorly on many graph-processing algorithms [9], their analysis was limited to two-bit predictors that have well-known limitations. Our analysis extends these findings and suggest that conventional branch prediction techniques are insufficient to solve these problems.

To make our case, we present a detailed architectural simulation of several important graph-processing workloads and provide an analysis of the hard-to-predict branches. We find that the majority of the mispredictions in these workloads come from a small number of derelict branches. To our knowledge, this is the first detailed empirical study of the performance of branch prediction on graph-processing algorithms.

2 Methodology

We use the SNIPER simulator [10] to study the branch behavior of the GAPBS [11] benchmark suite.

We selected SNIPER due to its execution model which is fast and makes modeling an oracle predictor simple. Its branch predictor simulator is reverse-engineered from profiling the Pentium M's branch predictor and is a realistic representation of high-performance branch predictors in real hardware [12]. It consists of a Branch Target Buffer (2,048 entries), an Indirect Branch target Buffer (256 entries), a loop predictor, and a hybrid predictor made up of a bi-modal table (4,096 entries) and a global predictor (2,048 entries).

Although the Pentium M predictor we use is not as advanced as LTAGE [13] and some predictors implemented in recent chips, it is an accurate reflection of the branch predictors found in real processors. To make sure we were making a fair comparison to the state-of-the-art, we verified our analysis on gem5 using an LTAGE predictor with a 4,096-entry BTB. We do not use gem5 for all of the experiments because it lacked perfect branch prediction simulation and was too slow to run multiple inputs over multiple trials.

We chose the GAPBS benchmark suite [11] because it's comprised of basic kernels that are the building blocks of all other graph-processing applications. We marked the region-of-interest (ROI). The version of GAPBS with our modifications to identify ROI can be found in our github repository [14]. In Section 3 when we refer to specific branches, the line numbers are based on our version of the code.

Ideally, we would also use the same input graphs recommended in GAPBS (Facebook, Twitter, Web, Roads) which are meaningful, real-world data sets that are relevant to many real applications. However, all of these data sets are prohibitively large for simulation. Instead, we used LiveJournal and Orkut (social media graphs), and roadsCA (the road network of California), because they had similar topology but are much smaller and easier to simulate.

A brief description of each benchmark is found in Table 1 and a description of the input graphs is found in Table 2. Not displayed, but also run as a base case, is a manually constructed graph with the structure of a binary tree with the same number of nodes as LiveJournal (our largest input). It's notable that for the binary-tree graph all predictors gave near-perfect prediction (MPKI < 0.1). This is likely due to the behavior becoming much more predictable. For example, the list of neighbors is always the same length for

- *Ahmed Samara is with the Department of Computer and Electrical Engineering, North Carolina State University, Raleigh, NC 27695. E-mail: asamara@ncsu.edu.*
- *James Tuck is with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27606. E-mail: jtuck@ncsu.edu.*

Manuscript received 21 May 2020; accepted 19 June 2020. Date of publication 30 June 2020; date of current version 22 July 2020.
(Corresponding author: Ahmed Samara.)
Digital Object Identifier no. 10.1109/LCA.2020.3005895

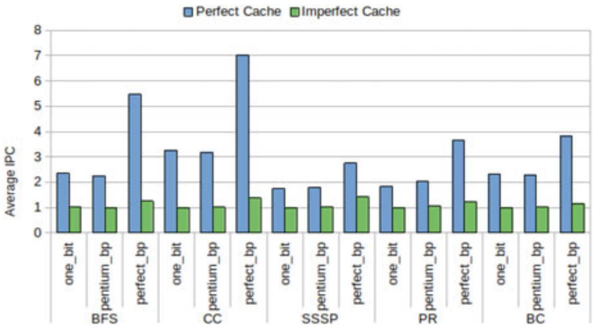

Fig. 1. IPC improvements for different branch predictors with different cache configurations.

TABLE 1
Description of the GAP benchmark suite

Benchmark	Description
Breadth-First Search (BFS)	Starting from the source, visit every node in the graph layer-by-layer.
PageRank (PR)	Give each vertex a rank based on its strength with neighbors. Originally used to rank web searches.
Single-Source Shortest Path (SSSP)	Find the shortest path from the source node to every other node in the graph.
Betweenness Centrality (BC)	Measure the 'centrality' of the source node, defined as the ratio between the shortest path between any other two nodes that contain this node and the total shortest paths between those nodes
Connected Components (CC)	Calculate subgraphs of nodes that have connections to each other from the total graph.

TABLE 2
The graphs used as inputs for the benchmarks

Input Graph	Vertices	Edges	Description
Orkut	3072441	117185083	Social Media
LiveJournal	4847571	68993773	Social Media
roadsCA	1965206	2766607	road network

every node (except for the root and leaf nodes), and the sequence of taken/not-taken for the visited check will always follow the same pattern.

We also ran each benchmark for one trial on gem5 [15] with the LTAGE predictor on the roadsCA input graph, as shown in Table 3.

LTAGE continued to show a high misprediction rate and consistently mispredicted on the same branches that were observed as derelict branches in the Pentium predictor which shows that even state-of-the-art predictors struggle on these branches. This shows that the high rate of mispredictions is not due to the less accurate Pentium branch predictor. We did not use the gem5 simulator for the remaining tests or on the remaining graphs because the simulation time was prohibitively long.

3 Histogram of Mispredictions

We ran each benchmark with every input graph and recorded the PC of every misprediction and the count of total mispredictions in the benchmark to calculate the percentage of mispredictions contributed by each individual branch. We aggregated that information in Table 4. For simplicity, we only include branches with consistently high mispredictions (average roughly 10 percent on all inputs).

The first column of Table 4 shows the workload name and line number of each branch. The line numbers refer to the modified source code in our github repository as noted above.

TABLE 3
The Comparison of MPKI on Each Benchmark Using roadsCA as Input

	Pentium	LTAGE
BC	31.38	24.69
BFS	35.84	19.37
CC	10.95	3.82
PR	15.25	9.91
SSSP	15.37	25.39
Geo.Mean	19.59	13.57

TABLE 4
The Percentage of Mispredictions in Each Benchmark | That Originated With the Respective Branches

Line#	Type	roadsCA	Orkut	LiveJournal
BC		31 MPKI	16 MPKI	21 MPKI
126	Visited-check	25%	32%	29%
75	Value-check	6%	29%	23%
71	Visited-check	4%	11%	16%
70	Neighbors	0%	11%	13%
BFS		36 MPKI	9 MPKI	21 MPKI
61	Neighbors	n/o	54%	36%
60	Visited-check	n/o	12%	28%
62	Visited-check	n/o	17%	24%
CC		11 MPKI	16 MPKI	19 MPKI
63	Var-length loop	13%	16%	31%
110	Neighbors	18%	14%	21%
45	Var-length loop	37%	28%	12%
50	Value-check	28%	27%	11%
PR		15 MPKI	8 MPKI	15 MPKI
52	Neighbors	99%	100%	99%
SSSP		15 MPKI	11 MPKI	17 MPKI
85	Value-check	27%	52%	50%
82	Neighbors	18%	30%	30%
81	Value-check	4%	9%	12%

Branches marked n/o were not observed in this run. Notably, the unlisted contributors to MPKI in BFS-roadsCA are also a value check, and a neighbors-iteration.

Overall, we find that the majority of mispredictions in the program were caused by a small number of branches, 15 to be exact. In each workload, three to five branches were to blame. Furthermore, several common behaviors emerged across the workloads as problematic. The most prevalent ones were the neighbors iteration, ranging from 14 to 100 percent per workload, and the visited check. We describe the specific problematic behaviors in the next section.

4 Analysis of Branches That Re-Occur in Multiple Benchmarks

4.1 The Neighbors Iteration

A common piece of code that existed in all of the GAPBS benchmarks was the iteration over the neighbors of a node. Occurs in BC at line 70, BFS at line 61, CC at line 110, PR at line 52, and SSSP at line 82.

This may be surprising at first because this is a simple for-loop that is often assumed to be easily predictable. However, that is not the case because the for-loop has a variable trip-count that is dependent on the number of neighbors of each node, making it impossible for the loop predictor to learn the correct count. Prior works have noted the difficulty of predicting variable-length for-loops [16]. One might assume that loop predictors could handle this case by learning the trip count and then predicting the exit at the appropriate count, but such strategies rely on the length of for-loops rarely changing [17] or changing only between a small set of values.

This is a particularly extreme example of a variable-length for-loop because it's a loop inside of a loop, and the length of the inner loop can change every iteration.

TABLE 5
The Contribution of Mispredictions on the Neighbors Iteration to the Total Mispredictions in Each Benchmark

Benchmark	mispreds %
PageRank	100%
BFS	35%
CC	21%
SSSP	29%

TABLE 6
The Percentage of Mispredictions in Each Benchmark That are Caused by Some Form of 'Visited Checks'

Benchmark	mispreds %
BFS	49%
BC	45%

These loops are more problematic in terms of IPC and MPKI on low-degree networks like roadsCA that are naturally limited to lower degree nodes. When the iteration count is always in the range from 2 to 4, then the misprediction frequency can be high (frequently failing to predict an exit, or predicting a false exit) and will make up a more significant amount of the execution. On the other hand, social network graphs may have nodes with hundreds or even thousands of neighbors, limiting the frequency of these predictions.

4.2 The Visited Check

Visited checks are common in graph-processing. Before computing the relevant operation, it needs to check whether or not the node has been visited already.

We find this type of branch in BFS at lines 60 and 62 and in BC at lines 71 and 126. In BFS it involves a check to see if a node has a parent or not (or a check if it's in the frontier for a bottom-up search), and in BC it involves checking the successor bit-map for whether or not a depth has been set. It's difficult to predict because whether or not a node has been visited before depends on the topology of the graph data. The prior history of branches may not convey a useful pattern for determining if the next node has been visited. In a graph with a regular shape, like the binary-tree graph, such a pattern does exist leading to a low branch misprediction rate. However, for real-world graphs, our performance data suggests such predictable patterns are uncommon.

Previous works have noted that the check on a visited node is an expensive operation, and sought to minimize the number of checks using an alternative implementation [18], which is the one found in GAPBS [11]. It's likely that a naive implementation of BFS would see even higher mispredictions on this branch.

4.3 Variable Length Loops

As with the neighbors iteration, variable-length loops can be difficult to predict. This occurs at CC line 45 and 63.

Since the while-loop continues executing until a condition is met, the number of iterations it takes for the condition to be met will vary wildly in between instances of the loop and it will be difficult to predict the exit.

4.4 Value Checks

The remaining branches fall into the category of value-checks, simple if statements that check a value that is based on the structure of the graph (for example, checking if the distance of the currently examined edge is less than the previously observed minimum distance). These occur in SSSP at lines 81 and 85, CC at line 50, and BC at line 75. Value checks are similar to the visited check. They all have behavior that is dependent on the properties of the graph, and it is not repetitive enough for the branch predictor to learn a pattern.

TABLE 7
The Percentage of Mispredictions in Each Benchmark That are Caused by Some Form of 'Variable Length Loops'

Benchmark	mispreds %
CC	43%

TABLE 8
The Percentage of Mispredictions in Each Benchmark That are Caused by Some Form of 'Value Checks'

Benchmark	mispreds %
BC	23%
CC	11%
SSSP	60%

5 Future Directions

We propose to systematically specialize branch predictors with new features to handle these problematic branches.

One high-level approach is to specialize for each branch individually. For example, a specialization for the neighbors-iteration may be relatively straightforward. The predictor simply has to know the degree of each node. The graph data-structure can be modified to include the degree of each node, and a pre-fetching scheme or hint instruction can be used to communicate these to the predictor as the neighbors are traversed. Timely pre-fetching of this node degree will be critical for correct predictions but likely feasible.

As another example of customizing for a specific branch, a small table in the branch predictor could be dedicated to caching previously visited nodes to help quickly predict the outcome of the visited check. This cache doesn't need to record all visited nodes, it just needs to allow the visited check branch to be predicted correctly most of the time. Effective ways of populating this cache and informing the predictor of the next node to be visited would need to be studied.

The remaining branches have varying functionality but share common features like being input and computation dependent. For these branches and others like them that may emerge, a more general approach may be needed. One possibility is dynamic precomputation [19], [20]. The concept is to build slices of instructions that the derelict branch instructions depend on and execute them on an idle core or on other dedicated hardware. Like the visited-cache, these slices do not have to execute precisely because they only offer predictions. Effective slice construction has been a big challenge in prior work focused on general-purpose codes, but since we are specializing for graph workloads, these slices can be manually constructed ahead of time with knowledge of specific graph workloads and tuned for high performance. The hardware they run on can similarly be tuned and simplified for graphs.

6 Conclusion

We have presented an analysis that shows that branch predictors fail to achieve low misprediction rates on key workloads in graph-processing. We determined that the majority of mispredictions are due to a small number of branches and we described these branches and their relative importance to the overall misprediction rates. Future work should target these branches to boost the performance of graph-processing workloads on general purpose processors.

References

[1] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[2] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[3] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[4] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "WTF: The who to follow service at Twitter," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 505–514. [Online]. Available: http://doi.acm.org/10.1145/2488388.2488433

[5] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "GraphJet: Real-time content recommendations at Twitter," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, Sep. 2016. [Online]. Available: https://doi.org/10.14778/3007263.3007267

[6] J. Lin, "Scale up or scale out for graph processing?" *IEEE Internet Comput.*, vol. 22, no. 3, pp. 72–78, May 2018.

[7] A. Basak *et al.*, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 373–386. [Online]. Available: https://ieeexplore.ieee.org/document/8675225/

[8] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "ReCALL: Reordered cache aware locality based graph processing," in *Proc. IEEE 24th Int. Conf. High Perform. Comput.*, 2017, pp. 273–282.

[9] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," 2014, *arXiv:1411.1460*. [Online]. Available: http://arxiv.org/abs/1411.1460

[10] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *Proc. 8th Int. Summer School Adv. Comput. Archit. Compilation High-Perform. Embedded Syst. Abstracts*, 2012, pp. 91–94. [Online]. Available: http://hdl.handle.net/1854/LU-2968322

[11] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2017, *arXiv:1508.03619*. [Online]. Available: http://arxiv.org/abs/1508.03619

[12] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 207–217. [Online]. Available: http://ieeexplore.ieee.org/document/4919652/

[13] A. Seznec, "A new case for the TAGE branch predictor," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, Art. no. 117. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2155620.2155635

[14] A. Samara, "AhmedSamara/GABPS-branch-analysis," Feb. 2020, original-date: 2020–02-04T20:04:43Z. [Online]. Available: https://github.com/AhmedSamara/GABPS-branch-analysis

[15] The gem5 simulator | ACM SIGARCH Computer Architecture News, 2011. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2024716.2024718

[16] C. Ozturk and R. Sendag, "An analysis of hard to predict branches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 213–222. [Online]. Available: http://ieeexplore.ieee.org/document/5452016/

[17] T. Sherwood and B. Calder, "Loop termination prediction," in *Proc. Int. Symp. High Perform. Comput.*, 2000, pp. 73–87.

[18] S. Beamer, K. Asanovic, and D. Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500," Tech. Rep. UCB/EECS-2011-117, EECS Department, Univ. California, Berkeley, 2011.

[19] J. Collins, D. Tullsen, H. Wang, and J. Shen, "Dynamic speculative precomputation," in *Proc.. 34th ACM/IEEE Int. Symp. Microarchit.*, 2001, pp. 306–317. [Online]. Available: http://ieeexplore.ieee.org/document/991128/

[20] K. Ibrahim, G. Byrd, and E. Rotenberg, "Slipstream execution mode for CMP-based multiprocessors," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit.*, 2003, pp. 179–190.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.