BARÇA: Branch Agnostic Region Searching Algorithm

Daniel A. Jiménez Paul V. Gratz Gino Chacon Nathan Gober
Texas A&M University

Abstract

We introduce BARÇA[1]*, a branch agnostic region searching algorithm for instruction prefetching. The technique is based on searching a control-flow graph. The nodes in the graph are fixed-sized* regions *of a number of cache blocks. Each edge is labeled with the number of times that edge was traversed in the control-flow of the program, allowing the probability of an edge being traversed to be calculated from among all the edges leaving a given node. To find candidate blocks to prefetch, the graph is searched to a given maximum depth, stopping when the product of probabilities from the source node to a given target reaches a minimum value defined for that depth. The algorithm is largely agnostic of branch instructions, inferring the flow of control from demand fetches to regions. A few optimizations are applied to improve space efficiency and performance, including using a spatial pattern within regions, handling returns specially, and compression of region addresses.* BARÇA *achieves a geometric mean speedup of 28.3% over a baseline of no prefetching, comparing reasonably well to an optimistic 35.0% speedup for an infinite-sized first-level instruction cache. This paper describes the algorithm, the optimizations, and the overhead.*

1 The Algorithm

The prefetcher divides memory into multi-block groups called *regions*. The number of blocks per region is a parameter to the algorithm. Each region reached by the program becomes a node in the control-flow graph. When control leaves a source region for a target region, an edge is inserted or updated in the control-flow graph. On certain demand accesses, the prefetcher begins a limited-depth search at the region that includes that demand access. The nodes reached by that search become candidates for prefetching. Each node includes a spatial pattern, a BPR-bit array that indicates whether the corresponding cache block has ever been accessed; blocks that have never been accessed are not considered for prefetching.

The intuition behind the BARÇA algorithm is that, for large working set programs, we would like to store relatively few entries in the control-flow graph compared to a branch target buffer (BTB) based prefetcher. This is the same intuition behind Shotgun [4], but rather than using a hierarchical approach, we choose a simpler coarse-grained node. The algorithm is mostly agnostic of branch instructions within cache blocks or regions; a discontinuous fetch obviously implies a branch somewhere in the region but is represented simply as another edge in the control-flow graph. The one exception is returns, as returns have many targets but only one that will be fruitful on a given search, so they are handled as a special case.

[1]FC Barcelona, the Barcelona football club, is colloquially known as Barça.

1.1 Edges

Edges in the control-flow graph are weighted with a count of the number of times that edge was traversed. They are also augmented with a flag that indicates whether this edge can be traversed by a return instruction. The counts of every edge originating at a given node can be used to determine the probability that one of those edges will be traversed, which guides the search. The counts are 18 bits to conserve space. If incrementing a count would exceed 18 bits, all the counts in the control-flow graph are halved before the increment so the counts never overflow and continue to retain their proportional values.

1.2 The Control-Flow Graph

The control-flow graph is represented as a 256×64 set-associative memory of edges. The tags are the source region addresses. On an access to a control-flow graph edge, multiple edges may match the tag, indicating multiple targets for the source. Many sources may share a single set, allowing an efficient adjacency-lists graph representation. Conflict misses in control-flow graph sets are handled by replacing edges using a least-frequently-used policy based on the edge count.

1.3 The Search

A search may be initiated from a demand fetch, an instruction cache fill or return instruction. When a new region is entered that has not recently been searched, a new search is initiated. The search is a depth-limited depth-first search. We find a maximum depth of 5 results in the best performance. Along each path from the source, the algorithm keeps a running product of the probabilities of each edge along the path. If the product fails to exceed a pre-determined per-depth threshold, the search along that path is terminated. When the search algorithm considers traversing an edge labeled as 'is-return,' meaning that it had been traversed once before as a return of a return instruction, the edge is only traversed if the target region can be found on the return address stack.

1.4 Shadow Cache

BARÇA keeps a *shadow cache* that mirrors the tags in the L1 i-cache. It is updated on every fetch as well as on every i-cache fill. The shadow cache is used to filter prefetch candidates as well as to identify useful and useless prefetches. For each block, it contains a tag, a valid bit, replacement state for the LRU policy, a bit indicating whether the block was prefetched but not yet used, and a pointer to the control-flow graph node responsible for issuing the prefetch if that block had been prefetched.

1.5 Selecting and Issuing Prefetch Candidates

For each region encountered along the search, BARÇA considers all the blocks in that region. If the block is not found in the shadow cache, it is added to a list of prefetch candidates. Then, the list is traversed in order of the probability of the node from which the candidate originated. Up to a fixed number of candidates are added to a prefetch queue (not ChampSim's queue) on each search. On every cycle, a number of prefetch candidates will be dequeued and issued from the prefetch queue.

2 Optimizations

We employ a number of optimizations to improve performance and space efficiency of BARÇA:

2.1 Compression of Region Addresses

With 64-byte blocks and 2 blocks per region, distinctly identifying each region requires 57 bits. Each control-flow graph node has two region addresses: a source and a target. A naive representation would consume 114 bits per node just for region addresses. Thus, we use a compressed representation. We divide region numbers into 45 upper "area bits" and 12 lower "offset bits." We keep a table called the "area table" of 128 distinct 45-bit areas. A compressed region number consists of the 7-bit index into the area table where the area bits can be found and the 12 offset bits, for a total of 19 bits per region address. The control-flow graph is indexed by taking the source address modulo the number of sets in the structure, *i.e.* 256, so the lower 8 bits of the source address need not be stored in a control-flow graph node. Thus, source addresses consume 11 bits and target addresses consume 19 bits.

The area table is initialized to contain an invalid value that has not been encountered in the traces. An area table entry is filled the first time a region from that area is encountered. If the table is full of valid entries, random replacement is used. The area table allows covering 64MB of program text simultaneously. In practice we find that area table never requires replacement. Similar address compression schemes were used in previous work on indirect branch prediction [8, 1].

2.2 Special Treatment of Returns

We tried a number of ways to improve the accuracy of the prefetcher. We found that, generally, letting it be as aggressive as possible led to better performance than trying to reduce useless prefetches. One exception was that following all possible targets of returns led to too many harmful useless prefetches. So we made an optimization to attempt to follow only the correct return target. We implemented a return address stack, pushing return addresses of calls and popping the stack on returns. We labeled edges originating from returns, discovered through the operation of branches, with an "is-return" flag. When searching the graph, we allow an "is-return" edge to be followed only if the target of that edge is on the return address stack.

We would have liked to have done the same sort of optimization for indirect branches, only following the correct target, but to do so would require implementing a highly accurate indirect branch predictor that could speculate through several levels of the search. We determined that doing that was not worth the extra hardware required.

2.3 "Would Be Nice" Queue

The results of the search are ordered by the probabilities of the associated edges. The algorithm only allows a certain number of these prefetches into the prefetch queue so as not to overwhelm the prefetcher and possibly delay subsequent prefetches. However, we find that during a large portion of most programs' execution, the prefetch queue is empty. Thus, we maintain another queue of lower probability prefetches that "would be nice" to issue if there is available bandwidth. If the main prefetch queue filled by the search becomes empty on some cycle, this alternate prefetch queue is used to issue prefetches.

2.4 Recently Searched List

We keep a list of regions from which searches were recently initiated, maintained in first-in-first-out order. If a region is on this list, it is will not be searched. Thus, we avoid issuing some superfluous prefetches.

2.5 Tweaking Probability Counts

Each control-flow graph edge keeps a count of the number of times that edge was traversed. The count is used to compute the probability that a given edge will produce a fruitful prefetch. We bias these probabilities to produce more useful, fewer useless prefetches, and late prefetches.

Each time the shadow cache evicts a block that was prefetched but never accessed, the count for the edge responsible for that useless prefetch is decremented by 2. Each time the shadow cache accesses a prefetched block indicating the prefetch was useful, the count for the corresponding edge is incremented by 3. Each time a demand fetch misses but the fetch block is recorded as prefetched in the shadow cache, indicating a late prefetch that was requested but has not yet found its way into the cache, the corresponding edge is incremented by 5. This way, the next time the same block is put onto the prefetch queue, it will have a higher probability and thus a more favorable position in the schedule of prefetches. Now the values computed no longer reflect the true probability of encountering an edge, but take into account the usefulness and timeliness of the edge together with its frequency.

2.6 Tuning of Parameters

We empirically explored the design space of parameters to the algorithm. Table 1 shows the values that resulted in the best performance. With the large control-flow graph afforded by the 128KB championship budget, we find that BPR=2 is a good trade-off between region granularity and prefetch accuracy. With a smaller budget we would likely use a coarser granularity.

Parameter	Value
Blocks per region	2 blocks
# of prefetches to dequeue	4 per cycle
Prefetch queue size	14 entries
Max. # prefetches to queue per search	5
“Would be nice” queue size	10 entries
Edge counter width	18 bits
Recently searched list size	5 entries
Max. depth of search	4
“Real” maximum depth of search	5
Min. probability to search depth 1	0.046
Min. probability to search depth 2	0.001
Min. probability to search depth 3	0.0275
Min. probability to search depth 4	0.007
Counter increment on useful prefetch	3
Counter increment on late prefetch	5
Counter decrement on useless prefetch	2

Table 1: Best values of parameters to the algorithm

3 Overhead

3.1 Space Overhead

There are several structures in the prefetcher:

The area map is a direct-mapped memory of 128 58-bit entries mapping areas of memory to regions to support compression of node numbers in the control-flow graph. The area map is 7,424 bits.

The control-flow graph is a 256×64 set-associative memory. Nodes in the graph use a compressed representation as a pair of 7-bit area number and 12-bit offset within the area, giving a region number. Each control-flow graph entry has the following fields: an 18-bit counter, a source node, a target node, a 2-bit spatial pattern, and an “is-return” bit that is true if this edge is traversed by a return instruction. Recall that compressed region addresses consume 19 bits, and that since the structure is indexed by source node using modulo indexing as in a set-associative cache, the lower 8 bits of the source node’s area offset need not be represented. Thus, total number of bits in the control-flow graph is $256 \times 64 \times (1 + 2 + 18 + 19 + 11 + 1) = 851,968$ bits.

The shadow cache is a 64×8 set-associative memory. Each entry has the following fields: a 1-bit valid bit, a 1-bit prefetched bit indicating whether an entry has been prefetched but not yet demand-accessed, a 24-bit partial tag, a 3-bit LRU position, and a pointer to a control-flow graph edge that would consume an 8-bit row and 6-bit column in a real implementation. Thus, the total number of bits in the shadow cache is 22,016 bits.

The recently-searched list keeps recently searched regions that should not be searched again. It has 5 57-bit entries, consuming 285 bits.

The prefetch candidates list keeps prefetch candidates identified by the depth first search. It has 56 entries. Each entry has a 58-bit block address, a 64-bit probability, a 3-bit depth, and an 8+6 bit pointer to a control-flow graph edge. Thus the list consumes 7,784 bits.

The prefetch queue is a queue apart from ChampSim’s internal prefetch queue that stores candidate prefetches to be dequeued as most one per cycle. It has 14 entries, each the same size as the entries in the prefetch candidates list above. Thus it consumes 1,946 bits.

The “would be nice” queue has 10 entries the same size as the prefetch queue entries, so it consumes 1,390 bits.

The search results map of regions is filled by the depth-first search. The map contains up to 56 entries. The key is an 8+6 bit pointer to a control-flow graph edge, and the value is a pair of a 64-bit probability and a 3-bit depth, for a total of 4,536 bits. **The depth-first search frontier list** contains the contents of the search results map sorted by probability. It has 56 entries of the same type as the prefetch queue entries, so it consumes 7,784 bits.

The distinct candidates map is an associative map that aids in filling the prefetch candidates list by making sure there are no duplicate entries. It has 56 entries of 58-bit block addresses, consuming 3,248 bits.

The return address stack is a 64-entry memory of 64-bit return addresses maintained with stack discipline. It consumes 4,096 bits.

Adding the bits for all structures, we get 912,477 bits, or 111.4KB. There are various bits of state in the code the evaluators might wish to consider, *e.g.* scalar variables for counting etc. We are confident that these variables would form a negligible fraction of the 16.6KB we leave on the table.

4 Scalability

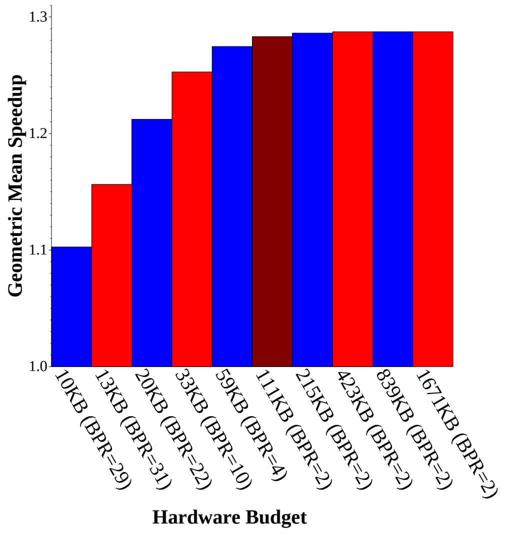

Figure 1: BARÇA scales to a wide range of hardware budgets. BPR is blocks per region. (Red and blue are the colors of the Barça football club, and maroon represents Texas A&M.)

The idea of BARÇA is to give a compact metadata representation by aggregating multiple blocks into a single CFG node. It is hard to demonstrate the value of the idea with the contest's large 128KB hardware budget. The optimal number of blocks per region (BPR) is only 2. Figure 1 shows the geometric mean speedup over no prefetching given at budgets from 10KB to 1671KB. The number of CFG sets ranges from 8 to 4,096, doubling the number of sets for each point on the x-axis. For each budget, we find the optimal number of blocks per region (BPR). At 10KB budget, the best BPR is 29, giving a region size of 1856 bytes. At this budget, BARÇA delivers a geometric speedup of 10.2%. At 33KB budget, about the same capacity as the L1 i-cache, BARÇA yields a speedup of 25.2% with a BPR of 10. Our IPC1 entry uses a hardware budget of 111KB, giving a 28.3% speedup. At larger budgets, there is not much improvement. At a 4KB budget, the speedup is 28.7%, only 1.4% better than the 111KB version.

4.1 Logical Complexity

The prefetcher state easily fits into the hardware budget of 128KB. The hardware structures used are well-understood in front-end microarchitecture: set-associative SRAM arrays and small fully-associative arrays. The most challenging task is the depth-first search. An efficient parallel depth-first search has been demonstrated in the network-on-chip literature [7]. That work showed a practical low-latency implementation of the Bellman-Ford algorithm that could be adapted for our work.

Although we focus on the algorithm in this paper, we believe a reasonable implementation is possible by pipelining the search and taking advantage of the slack time between prefetches. For example, we measured that, on average, ChampSim's prefetch queue is empty for more than 50% of simulated cycles even with our "would be nice" list of potential prefetches. We limit the number of prefetches that will be inserted into our prefetch queue from the search to 5. However, on average far fewer prefetch candidates are generated per search. On average, each search results in 0.84 prefetch candidates over the 50 traces, with a maximum of 1.7 average prefetches candidates per search for one benchmark. We also limit the number of searches by only searching when entering a region that has not been recently searched, so most demand fetches do not result in invoking the search algorithm at all. We are aware of timing issues when *e.g.* accessing a large structure such as the control-flow graph and making several accesses to the shadow cache. Having worked with an industrial RTL team on very complex front-end designs, we believe that with clever pipelining and aggressive SRAM macros these issues can be resolved with minimal impact on performance.

5 Related Work

Our scheme bears strong resemblance to BTB-directed prefetching approaches. It is also similar to Markov-based prefetchers [2] as it uses a graph labeled with weights that represent probabilities that the next step in the graph will be reached. The idea of prefetching through branch targets began with Smith and Hsu [9]. Recently, BTB-directed prefetching has been enhanced by using a stream-based prefetcher to do both instruction and BTB prefetching [3]. A more efficient instruction prefetcher, Boomerang [5], reduces the metadata required for branch-predictor-directed BTB and instruction prefetching, followed up by Shotgun [4] which uses a hierarchical approach to further relieve the metadata problem. Our idea can be seen as a simplified take of Shotgun; rather than explicitly acknowledging program structure through procedures and branches, we divide the program into equal-sized coarse-grained regions. Our spatial patterns that indicate whether a given block in a region has been touched is inspired by spatial footprints [6], an idea that has found its way into several modern prefetchers including spatio-temporal streaming [10].

6 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 Computing Research.

References

[1] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. Bit-level perceptron prediction for indirect branches. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 27–38, New York, NY, USA, 2019. Association for Computing Machinery.

[2] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[3] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 166–177, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 30–42, New York, NY, USA, 2018. ACM.

[5] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 493–504. IEEE, 2017.

[6] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 357–368, 1998.

[7] Mukund Ramakrishna, Vamsi Krishna Kodati, Paul V. Gratz, and Alexander Sprintson. Gca: Global congestion awareness for load balance in networks-on-chip. *IEEE Trans. Parallel Distrib. Syst.*, 27(7):2022–2035, July 2016.

[8] André Seznec. A 64-kbytes ittage indirect branch predictor. In *Proceedings of the JWAC-2: Championship Branch Prediction*, June 2011.

[9] J. E. Smith and W. . Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 588–597, 1992.

[10] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News*, 37(3):69–80, June 2009.