

Exploring Prefetching, Pre-Execution and Branch Outcome Streaming for In-Memory Database Lookups

Mustafa Cavus, Mohammed Shatnawi, Resit Sendag and Augustus K. Uht

Abstract—Lookup operations for in-memory databases are heavily memory-bound because they often rely on pointer-chasing linked data structure traversals. They are also branch heavy with branches that are hard-to-predict due to random key lookups. In this study, we show that although cache misses are the primary bottleneck for these applications, without a method for eliminating the branch mispredictions, only a small fraction of the performance benefit is achieved through prefetching. We propose the Node Tracker (NT), a novel programmable prefetcher/pre-execution unit that is highly effective in exploiting inter key-lookup parallelism to improve single-thread performance. We extend the NT with branch outcome streaming (BOS) to reduce branch mispredictions and show that NT with BOS can achieve an extra 3x speedup. Finally, we evaluated the NT as a pre-execution unit and show that we can further improve the performance in both single- and multi-threaded execution modes.

1 INTRODUCTION AND BACKGROUND

In-memory database lookups are highly memory- and branch-bound because they rely on traversals of linked data structures (LDS) (e.g., hash-table walk, binary search tree) whose detailed dependence structure is unknown until runtime. The resultant sequential nature of dependent pointer dereferences cause long memory stalls for each lookup. Executing multiple lookups is an effective way of hiding memory latencies by exploiting memory-level parallelism (MLP). One way to achieve inter-lookup parallelism is multithreading. However, threads stall often due to frequent cache misses, which results in a highly inefficient use of today's aggressive out-of-order cores. Increasing single-thread performance requires the ability to launch multiple lookups, which is not possible due to limited instruction window size.

Helper threads [4] can be used to create separate threads to issue prefetches for increased MLP. However, they eventually tend to stall and struggle to stay ahead of the main thread due to load-miss chains created by LDS traversals. Recent work by Kocberber et al. [1], called AMAC, redesigned the lookup code for exploiting inter-lookup parallelism by maintaining the state of each lookup separately from that of other lookups. AMAC provides significant speedups, however, it requires substantial programming effort, incurs large instruction overhead, and its effectiveness relies on the timeliness of software prefetches.

In this paper, we propose a configurable hardware prefetcher/pre-execution unit called Node Tracker (NT) which implements the AMAC software algorithm. NT has several advantages over its software counterpart. The main lookup code is not altered. A small dose of specialization and the ability to react to prefetched data allows decoupled runahead of future key lookups.

NT resembles the Continuous Runahead (CR) [10] method in that it runs ahead of the demand execution and prefetches future traversals. However, our runahead engine implements the AMAC algorithm, which can stay ahead of the demand execution while CR uses the same traversal loop as the main core and is subject to the same stalls as the main core. Ainsworth and Jones [3] proposed a more general system with programmable RISC cores to implement an event-triggered prefetcher (ETP). While this method performs well for prefetching nodes in lookup traversals, it does this with higher hardware cost and significant fetch and execute cycles per event. In addition, it has a more complex programming interface.

Programmable prefetchers such as ETP and our proposed NT can efficiently reduce cache misses for key/value lookups. However, we observe that single-thread performance is severely bottlenecked by frequent branch mispredictions in the presence of prefetching. Prior work [1-3, 9-11] overlooks this important fact and considers prefetching in isolation and hence can achieve only a fraction of the performance headroom. We extend NT to employ branch outcome streaming (BOS), greatly reducing branch mispredictions. We show that NT can reclaim most of the lost performance opportunity.

Even with perfect prefetching and branch prediction, long dependency chains for a single lookup limits the number of simultaneous lookups in the instruction window. We show that further improvement is possible by running the NT in a pre-execution/accelerator mode that feeds future lookup results to the core.

With typical hardware accelerators, the original application has to be modified and the out-of-order core running the main thread launches the lookup code to be executed on the accelerator. The core is idle until the accelerator completes all of the lookup tasks. Considering adding this type of accelerator per core on a multi-core CPU, many out-of-order cores remain idle wasting resources. Running a multi-threaded key search in this case would further be bottlenecked by extra synchronization traffic in the presence of accelerators, thus programmability becomes an issue.

Unlike prior on-chip accelerators [4, 12], NT is tightly integrated to the processor pipeline to allow the core to handle synchronization and algorithm-specific processing of key-matching nodes. The main core is also not idle with NT. Lookups are still executed by the core but only on the key-matching node provided by the NT. If the core starts catching up with the NT, NT starts dropping future lookups, sharing its load with the core. These lookups are later demand-executed by the main core from start to end.

Summary: NT is neither simply a prefetcher nor an accelerator in the traditional sense. It can support a variety of data layouts with limited programmability. The flexibility provided by NT eases programmability as the original lookup code does not have to be modified. We show that NT provides significant speedups in both prefetcher-only and accelerator modes with and without BOS.

2 NODE TRACKER

Overview: The design of NT is inspired by the software implementation of AMAC [1]. The core idea is to keep the full state of each in-flight traversal separate from that of other in-flight traversals. This allows multiple LDS traversal-misses from different lookups to overlap, increasing MLP. Similar to

• The Authors are with the Department of Electrical, Computer and Biomedical Engineering, University of Rhode Island, Kingston, RI 02881. E-mail: (mcavus, mshatnawi, sendag, gusuht)@uri.edu.

recent prefetchers [3, 10] and on-chip accelerators [4, 12], NT reduces design cost and complexity through tight integration with the core, eliminating the need for dedicated TLB and cache units.

NT is configured using special instructions inserted by the programmer/compiler, (before the outer loop) as it is marked in code snippet 1. These special instructions simply record the traversal details into NT's configuration registers. Figure 1 shows the key components of the proposed NT hardware. There are three main NT components: Key and Node-Address Provider (KNAP), Node Traversal Engine (NTE) and the Result Buffer (RB).

```
[configuration instructions]
for (i = 0; i < NUM_TUPLES; i++) { // outer loop
    int key = keys[i];
    int idx = HASH(key);
    node *n = ht[idx];
    int matched = 0;
    while (n) { // inner loop
        for (j=0; j<n->count; j++)
            if (key == n->keys[j]) {
                matches++; matched = 1;
                //other operations using matching node
                break;
            }
        if (matched) break;
        n = n->next;
    } // end inner loop
} // end outer loop
```

Code Snippet 1: Simplified kernel of probe hash table (PHT). In our experiments, each node in a hash bucket has 2 keys.

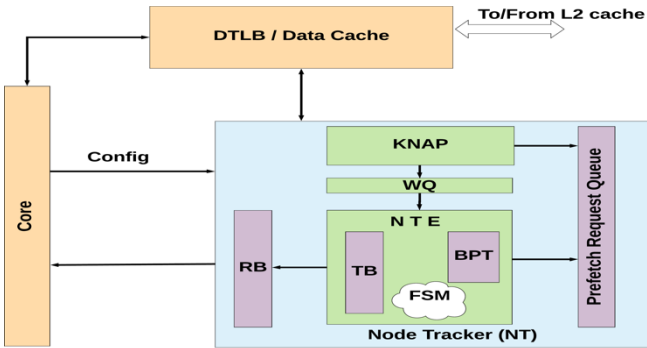


Figure 1. Overview of Node Tracker.

Key and Node Address Provider (KNAP): KNAP is a programmable prefetcher that fetches the target key (i.e., the key to be searched) and computes the address of the node where lookup should start. KNAP is similar to the recent sophisticated programmable prefetchers ATP [9] and ETP [3]. Both ATP and ETP can prefetch irregular patterns – they are initially designed to target indirect access patterns. KNAP's implementation was made similar to ATP because of ATP's simplicity and successful operation.

KNAP runs N iterations, or `lookupIDs`, (32, in our evaluation) ahead of the demand access of the target key array. The operation is triggered when the load instruction reading the target key array **commits**. In Code Snippet 1, KNAP prefetches the elements of the “keys” array and the “ht” array that holds the head node pointers, by using the PC of the load instruction passed from software.

KNAP is decoupled from NTE through a circular-FIFO **Wait Queue (WQ)**. When KNAP calculates the prefetch address of a head node, it assigns a new future `lookupID`, or iteration number, (calculated by adding the prefetch distance (32) and the currently committed `lookupID`) and inserts a lookup task in the WQ. WQ holds the target key values and head node addresses along with the `lookupIDs`. Lookup tasks wait in the WQ until they are fetched by the NTE.

Node Traversal Engine (NTE): The NTE consists of a Task

Buffer (**TB**) to hold lookup tasks received from the KNAP/WQ, a Finite State Machine (**FSM**) to process the tasks in the TB and a Branch Pattern Table (**BPT**) to generate future branch outcomes and facilitate BOS.

TB resembles the software structure that AMAC implements to keep track of in-flight lookups. When a TB entry is available, NTE initiates a new lookup (retrieved from the WQ) by saving the state of the task in the TB entry. TB is a circular buffer whose entries keep the full state of each independent in-flight key lookups. The state contains all the information necessary to continue or terminate the lookup. The key field contains the lookup key and is used for node comparisons throughout the lookup. Upon a match, the `lookupID` field is used for communicating the results. The stage field indicates the stage to process. Finally, the `ptr` points to the node being prefetched but not yet visited. Using the combination of stage and `ptr` fields, the exact status of each in-flight lookup is preserved.

The lookup operations in TB entries are executed by a FSM, shared by all TB entries. The first step in processing a TB entry is to load its state (latest stage in its execution) into the local registers of the NTE. Execution starts from that stage. The lookup key is retrieved and the key is compared against the `ptr`→key to determine the next state. If the lookup is not yet completed, a new memory prefetch is issued to the next linked node and the TB entry's `ptr` field is updated with the address of that node that will be visited in the next stage. Then the next TB entry is read and processed in a round-robin fashion. If the lookup task of a TB entry has been completed, a new lookup is initiated by fetching from the head of the WQ.

Result Buffer (RB) and NT as a Pre-execution Unit: As shown in Code Snippet 1, a lookup operation consists of several node visits until a matching node is found. Since NT already pre-executes these node visits, the CPU pipeline does not need to re-visit the non-matching nodes again in most cases. A Result Buffer (**RB**) stores the matching node addresses and their `lookupIDs` for the completed lookups. The CPU pipeline can read the node addresses from the RB and instead of starting the inner loop from the head node, it can start from the matching node (i.e., the inner loop will be executed for one iteration), thereby improving performance.

We introduce a new instruction called `read_RB` to fetch the matching node address from the RB. If the `lookupID` is found in the RB but the result is not ready yet, the CPU stalls until the result is available. This also means that demand execution starts to catch up with the NT, which causes the oldest lookup entry of the WQ to be dropped. The reason behind this is that that lookup would probably be late in entering the TB. `read_RB` acts as a NOP if the `lookupID` is not found in the RB (which might happen due to drops). In this case, the CPU must perform the traversals for that `lookupID` itself, sharing the lookup tasks with NT.

The choice of inserting `read_RB` instruction into the code determines whether NT acts a pre-execution unit (called *ntpx*) or simply as a prefetcher (called *ntpf*). This is a distinguishing feature of the NT that differentiates it from both prior prefetchers and hardware accelerators.

Branch Outcome Streaming (BOS): Since NT must determine future conditional branch outcomes for correct prefetching / pre-execution, it can be programmed to also generate and store branch results for each key lookup ahead of their demand fetch. Both *ntpf* and *ntpx* can benefit from branch outcome streaming (BOS). We call *ntpf* (or *ntpx*) that also employs BOS as *ntpf+bos* (or *ntpx+bos*).

To implement BOS, the compiler/programmer passes branch patterns for different execution path scenarios to hardware using special instructions. Each pattern is stored in a Branch Pattern Table (**BPT**) and each entry in the BPT is mapped to a unique case (node matched, etc.). Figure 2

demonstrates the branch outcome pattern generation for a lookup in PHT (as shown in Code Snippet 1). In this example, each node holds two keys and the first key of the second node in the bucket has a match. As each node visit is completed by the TB, one of three cases (C1-C3) can occur: the first-key (C1) or the second-key (C2) matches, or no match is found on the node (C3). To visit the next node, as shown in Code Snippet 1, the next node address is calculated and checked if it is not NULL, forming a pattern C0. In our example, C0 is 1-bit, C1 is 2-bits, C2 and C3 are 4-bits each. Maximum and minimum stream lengths for a 4-nodes bucket are 3 (first node, first key matches) and 20 bits (no match in bucket), respectively. All these cases are known at compile time and bit patterns for them are stored in the BPT using NT's configuration instructions. The TB refers to the BPT after a node visit has been completed and the case has been observed. As the lookup finally completes, the TB has generated the branch outcome pattern of C0C3C0C1 for the lookup in this example. At the completion of the lookup, this pattern is written into the RB.

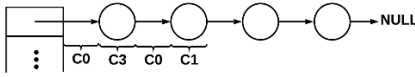


Figure 2: Branch Outcome Stream Generation for a single lookup in PHT. Each node in the bucket holds two keys. Example shows the case when a match has been found in the first key of the second node in the bucket.

Integration of BOS into the core's pipeline: In order to facilitate BOS, the fetch stage of the pipeline must keep track of `lookupIDs`. This can be done with a simple counter that increments when a new key lookup (i.e., a new outer loop iteration) has been detected (via the entry PC of the outer loop in Code Snippet 1). When a branch misprediction is detected, the `lookupID` counter at fetch can be restored using the latest valid instruction's `lookupID`. For every new `lookupID` at fetch, if that `lookupID` exists in RB, all of the branch predictor's (BP) predictions for that key lookup will be overridden with BOS read from the RB. Alternatively, a small 4-entry FIFO BOS buffer can be employed to receive BOS for completed lookups from RB and hence can be referred to at fetch instead of the RB.

It is important to note that NT **does not use nor does it interact with any of the BP hardware structures** and thus it does not change the state of the BP or BP's predictions. Finally, BOS is only activated when NT is programmed for it. Similarly, NT **can be turned off completely when it is not programmed** and therefore it does not have any negative performance impact on applications that do not use it.

3 EXPERIMENT METHODOLOGY

NT was implemented on the gem5 simulator [6]. We modeled an aggressive out-of-order core with a 256-entry instruction window and a 20-stage pipeline, implementing the x86 ISA. We assume a 64KB private L1D with 4-cycle latency and 256KB unified L2 cache with 12-cycle latency, both with 64B blocks and 8-way set-associativity. We used 24 miss-status holding registers (MSHRs) for L1D cache. This is on par with the recent Intel (e.g., Sunny Cove [7]) and AMD (e.g., Family 17h [5]) processors. Finally, our baseline configuration used the state-of-the-art TAGE-SC-L [13] branch predictor.

The multi-core processor system is modeled after AMD Epyc SoC [8]. Multiple CPU complexes are instantiated at the SoC level. Each CPU complex is four cores connected to an L3 cache. The L3 cache is 16-way associative, 4MB, made of four slices and supports 48 in-flight misses. Every core can access every slice with the same average latency of 32-cycles. The processor has four memory controllers with two channels each.

NT is evaluated with a 16-entry WQ, 16-entry TB, 8-entry BPT and 32-entry RB. The KNAP programmable prefetcher is

about 0.5KB in size. The total size of the NT is only 3KB.

In our evaluation, we used three workloads that consist of a probe hash-table algorithm (PHT), the binary search tree (BST) algorithm and the skip list (SL) algorithm. The baseline and AMAC implementations of each workload were inspired by the implementation of Kocher et al. [1]. The number of keys per node is two for PHT and one for BST and SL. The number of lookup keys is 2^{25} for all. The number of hash-table buckets for PHT is 2^{22} with 4-nodes per bucket, the maximum depth observed for the unbalanced tree is 65 for BST, and the number of levels in SL is 8. Finally, each lookup finds exactly one node match with uniformly distributed random unique keys.

4 EVALUATION

We compare our results with the AMAC method [1] and with recent prefetchers ATP [9], BO [14], SPP [15] and ISB [16]. We have tested several different variants of NT, namely *ntpf*, *ntpx*, *ntpf+bos* and *ntpx+bos*, which were discussed in Section 2. Figure 3a shows the single-threaded throughput results. We used million keys per second (MKPS) as the metric to measure the throughput of different methods.

4.1 *ntpf* and *ntpf+bos*

ntpf has large speedups over the no-prefetching baseline, improving the performance by 4.47x, 3.06x and 2.4x for PHT, BST and SL, respectively. *ntpf* significantly outperforms AMAC and other prefetchers tested over all workloads. *ntpf* eliminates almost all (95%, on average) DL1 cache misses therefore although we did not evaluate ETP, its performance is expected to at best match *ntpf*, NT's prefetch-only variant.

ISB and BO provide no speedup for the evaluated workloads. SPP provides marginal 1.03x speedup, on average. ATP (similar to the KNAP component of the NT) benefits from prefetching stride and indirect array accesses but it cannot prefetch for LDS traversals. ATP provides 1.45x speedup for PHT but no speedup for BST and SL.

AMAC allows multiple traversals to be performed concurrently, however it also has significant instruction overhead (2.2x-3.5x) and its performance depends on the timeliness of software prefetches. AMAC's speedup for PHT, BST and SL are 2.26x, 2.54x and 1.27x, respectively.

The branch bottleneck and *ntpf+bos*: *ntpf* and AMAC are both effective in prefetching for lookup traversals. However, we observe that the branch mispredictions per kilo instructions (MPKI) is typically high due to random key data. The baseline MPKI for PHT, BST and SL are 32, 52 and 25, respectively. AMAC has significantly lower MPKI as shown in Figure 3b because it increases the number of instructions dramatically.

We simulated perfect branch prediction for the baseline code and observe 1.62x, 1.86x and 1.93x speedups for PHT, BST and SL, respectively. It is obvious that hiding memory latencies is more important than eliminating branch mispredictions for the baseline. However, adding perfect branch prediction to *ntpf*, shown as *ntpf+perf* in Figure 3a, improves its performance by factors of 3.3x, 3.8x and 2.5x. Only after cache misses were eliminated does the branch bottleneck become more apparent. Prior work misses this because it is overshadowed by the memory bottleneck.

ntpf+bos achieves unprecedented performance by eliminating almost all branch mispredictions achieving speedups close to *ntpf+perf*. The speedup with *ntpf+bos* over the *baseline* is 14x, 11.8x and 5.9x for PHT, BST and SL, respectively. Without BOS, significant performance potential would be missed for the benchmarks evaluated.

4.2 *ntpx* and *ntpx+bos*

Using NT as a pre-execution unit, *ntpx*, improves the performance dramatically for all benchmarks by reducing the number of instructions executed per lookup. Figure 3a shows

that the speedup with *ntpx* over baseline is 7.6x, 20.2x and 17.1x for PHT, BST and SL, respectively.

As discussed above, branch MPKI is a significant bottleneck for *ntpf* and *ntpf+bos* significantly improves performance. In Figure 3b, we observe that *ntpx* also has high MPKI for PHT. This is because although a lookup with *ntpx* starts with the matching node, there are up to two keys per node and which key is a match is not predictable due to random data. Therefore, providing branch outcomes also improves its performance dramatically. *ntpx+bos* improves *ntpx* performance by 2.5x for PHT achieving an 18.93x speedup. However, we do not see any benefit of *ntpx+bos* over *ntpx* for BST and SL because *ntpx* already has significantly lower MPKI. *ntpx+bos* performs better than *ntpf+bos* because it reduces the number of instructions executed per lookup reducing the pressure on CPU buffers.

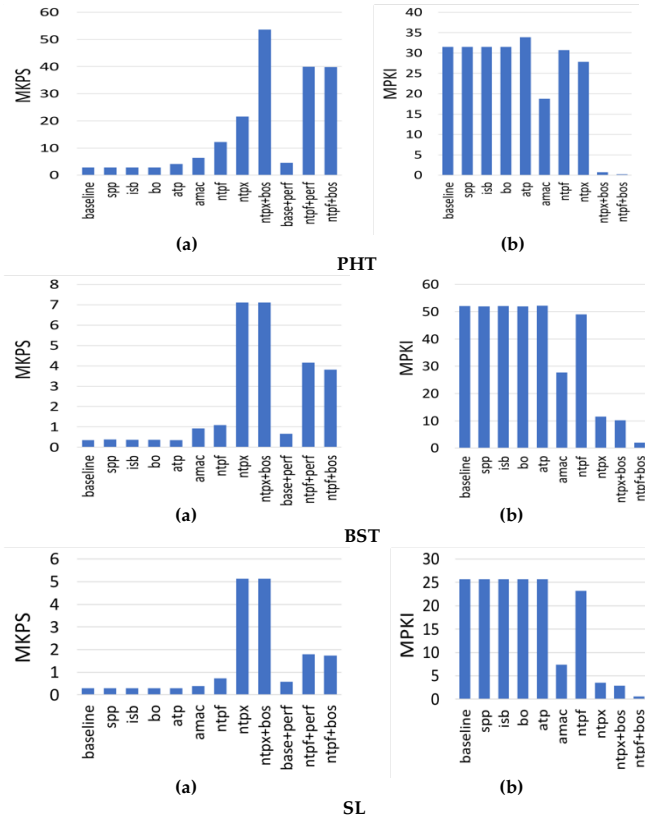


Figure 3. (a) Throughput measured as Millions of Keys Per Second (MKPS), and (b) Branch Mispredictions Per Kilo Instructions (MPKI), for various methods.

4.3 Multi-core Scalability

We evaluated the performance of NT, AMAC and ATP as the number of cores increases. For better visibility, we only show *ntpf+bos* and *ntpx+bos*, the best achieving NT variants. Figure 4 shows the speedup relative to a single-core no-prefetching baseline. NT significantly outperforms ATP and AMAC for all benchmarks. All methods scale well with an increasing number of cores. The scalability of *ntpx+bos* and *ntpf+bos* slightly slows down after 16-cores since they can better utilize the off-chip memory bandwidth. On average, with 24-cores, *ntpx+bos* achieves a speedup of 248.6x across all benchmarks, while the average speedups with *baseline*, *atp* and *amac* are 22.7x, 26.5x and 44.6x, respectively.

5 CONCLUSIONS

Lookup operations for in-memory databases are primarily bottlenecked by frequent cache misses caused by traversing LDS. We introduced the Node Tracker, a configurable

prefetcher/pre-execution unit that implements the AMAC [1] software algorithm to exploit inter-lookup parallelism in hardware. NT is tightly integrated into the core pipeline, accelerating lookup operations and leaving the synchronization and other functions to the main core.

We evaluated NT as a prefetcher and an accelerator and quantified the speedups due to alleviating memory or branch bottlenecks. We discussed four NT variants: prefetcher (*ntpf*), prefetcher with BOS (*ntpf+bos*), pre-execution unit (*ntpx*) and pre-execution unit with BOS (*ntpx+bos*). Our results show that *ntpf*, *ntpf+bos*, *ntpx* and *ntpx+bos* achieve 3.3x, 10.2x, 15x and 18.7x speedups, respectively, over a no-prefetching baseline. Furthermore, our method scales well with increasing numbers of cores.

6 ACKNOWLEDGMENTS

This work was partly supported by NSF under grant SHF-1422516. Authors would like to thank the authors of AMAC [1], especially, Boris Grot and Babak Falsafi, for providing us with the source code for the workloads used in this paper.

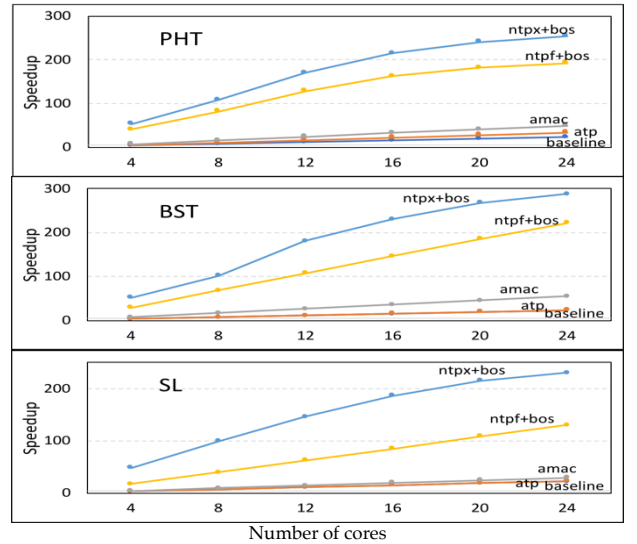


Figure 4. Effect of varying number of cores on performance.

REFERENCES

- [1] O. Kocher, B. Falsafi, and B. Grot, "Asynchronous memory access chaining," *Proc. VLDB Endow.* 9, 4 (2015), 252-263.
- [2] C. Jung, D. Lim, J. Lee and Y. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," *Proc. 20th IEEE International Parallel & Distributed Processing Symposium*, Rhodes Island, 2006.
- [3] S. Ainsworth and T. M. Jones, "An Event-Triggered Programmable Prefetcher for Irregular Workloads," 23rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018, 578-592.
- [4] O. Kocher, B. Grot, J. Picorel, B. Falsafi, K. Lim and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 468-479.
- [5] https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf
- [6] Binkert et al., "The gem5 Simulator," *ACM SIGARCH Comp. Arch. News*, 2011.
- [7] [fuse.wikichip.org/news/2371/intel-sunny-cove-core-to-deliver-a-major-improvement-in-single-thread-performance-bigger-improvements-to-follow/](https://www.wikiwand.com/en/news/2371/intel-sunny-cove-core-to-deliver-a-major-improvement-in-single-thread-performance-bigger-improvements-to-follow/)
- [8] <https://www.amd.com/en/products/epyc-server>.
- [9] M. Cavus, R. Sendag and J. J. Yi, "Array Tracking Prefetcher for Indirect Accesses," *IEEE 36th Intl. Conference on Computer Design (ICCD)*, 2018, pp. 132-139.
- [10] M. Hashemi, O. Mutlu and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," *49th IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, 2016, pp. 1-12.
- [11] X. Yu, C. J. Hughes, N. Satish and S. Devadas, "IMP: Indirect memory prefetcher," *48th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2015, pp. 178-190.
- [12] Mukkara et al., "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," *MICRO-51*, 2018.
- [13] Andre Seznec, TAGE-SC-L Branch Predictors Again. In *Proc. 5th Championship on Branch Prediction*, 2016.
- [14] P. Michaud, "Best-offset hardware prefetching," *HPCA* 2016.
- [15] Kim et al., "Path confidence based lookahead prefetching," *MICRO* 2016.
- [16] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," *MICRO* 2013.