2021

Fast Key-Value Lookups with Node Tracker

Mustafa Cavus
University of Rhode Island

Mohammed Shatnawi
University of Rhode Island

Resit Sendag
University of Rhode Island, sendag@uri.edu

Augustus K. Uht
University of Rhode Island

Citation/Publisher Attribution

Cavus, M., Shatnawi, M., Sendag, R., & Uht, A. K. (2021). Fast Key-Value Lookups with Node Tracker. *ACM Transactions on Architecture and Code Optimization, 18*(3), 34. https://doi.org/10.1145/3452099
Available at: https://doi.org/10.1145/3452099

Fast Key-Value Lookups with Node Tracker

MUSTAFA CAVUS, MOHAMMED SHATNAWI, RESIT SENDAG, and AUGUSTUS K. UHT, Dept. of Elect., Comp. and Biomed. Eng., Univ. of Rhode Island

34

Lookup operations for in-memory databases are heavily memory bound, because they often rely on pointer-chasing linked data structure traversals. They also have many 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 alone. 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 NT with branch outcome streaming (BOS) to reduce branch mispredictions and show that this achieves an extra 3× speedup. Finally, we evaluate the NT as a pre-execution unit and demonstrate that we can further improve the performance in both single- and multi-threaded execution modes. Our results show that, on average, NT improves single-thread performance by 4.1× when used as a prefetcher; 11.9× as a prefetcher with BOS; 14.9× as a pre-execution unit and 18.8× as a pre-execution unit with BOS. Finally, with 24 cores of the latter version, we achieve a speedup of 203× and 11× over the single-core and 24-core baselines, respectively.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: Hardware and software prefetch, in-memory database applications, pre-execution, branch prediction

ACM Reference format:
Mustafa Cavus, Mohammed Shatnawi, Resit Sendag, and Augustus K. Uht. 2021. Fast Key-Value Lookups with Node Tracker. *ACM Trans. Archit. Code Optim.* 18, 3, Article 34 (June 2021), 26 pages.
https://doi.org/10.1145/3452099

1 INTRODUCTION

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 causes 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

This paper is an extended version of the following CAL paper: M. Cavus, M. Shatnawi, R. Sendag and A. K. Uht, "Exploring Prefetching, Pre-Execution and Branch Outcome Streaming for In-Memory Database Lookups," in IEEE Computer Architecture Letters, vol. 19, no. 1, pp. 5-8, 1 Jan.-June 2020.

Author's addresses: M. Cavus, M. Shatnawi, R. Sendag, and A. K. Uht, Department of Electrical, Computer and Biomedical Engineering, University of Rhode Island, Fascitelli Center for Advanced Engineering, Room 360, 2 East Alumni Avenue, Kingston, RI, 02881, USA; emails: {mcavus, mshatnawi, sendag, gusuht}@uri.edu.

1544-3566/2021/06-ART34
https://doi.org/10.1145/3452099

frequent cache misses, which results in highly inefficient use of today's aggressive out-of-order cores. Increasing single-thread performance requires launching multiple lookups, which is currently not possible due to limited instruction window size.

Helper threads [4] can be added 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 **Asynchronous Memory Chaining (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 a significant advantage over prior software prefetching techniques [2, 3]. However, it requires substantial programming effort, incurs large instruction overhead with the modified algorithm, and its effectiveness relies on the timeliness of software prefetches.

Continuous Runahead (CR) [12] is a pre-execution engine that can run ahead of the demand execution to prefetch future lookup traversals. However, the CR engine uses the same traversal loop as the main core and is subject to the same stalls as the main core. Therefore, for control-dependent LDS traversals with **hard-to-predict (HTP)** branches that resolve late, CR can also not stay ahead of the main core. Recently, Ainsworth and Jones [5] 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 high hardware cost and significant fetch and executes cycles per event. In addition, its programming is complex.

Despite their drawbacks, AMAC, CR, and ETP can perform relatively well for prefetching lookup traversals. However, we observe that single-thread performance, especially true for in-memory key-value lookups, is severely bottlenecked by frequent branch mispredictions in the presence of prefetching. Prior work [1, 4, 5, 11, 12] overlooks this important fact and considers prefetching in isolation, and hence can achieve only a fraction of the performance headroom. In this article, we propose a configurable hardware prefetcher/pre-execution unit called **Node Tracker (NT)** that exploits inter-lookup parallelism by effectively integrating prefetching with branch pre-execution. We show that NT can reclaim most of the lost performance opportunities due to branch mispredictions by employing **branch outcome streaming (BOS)**. With BOS, pre-executed branch outcomes are stored in a buffer and are used to override the branch predictor's predictions. The detailed mechanism and operation of our proposed BOS are described in Section 3.7.

Another important issue for performance is that long dependency chains for a single key-value lookup limit the number of simultaneous lookups in the instruction window. Further improvement is possible by running NT in a pre-execution/accelerator mode that feeds future lookup results to the core. Unlike prior on-chip accelerators [6, 14], our design for NT is tightly integrated to the processor pipeline to allow the core to handle synchronization and algorithm-specific processing of key-matching nodes.

In this article, we make the following contributions:

- We show that although cache misses are the primary bottleneck for in-memory lookup operations, even a perfect prefetcher can only achieve a fraction of the potential speedup, because a large performance opportunity is lost due to branch mispredictions. That is, as one bottleneck is removed, the relative impact of the other bottleneck increases.
- We introduce the NT, which is a configurable prefetcher/pre-execution unit that exploits inter-lookup parallelism in hardware with software-exposed data structure traversal knowledge. NT is tightly integrated into the core pipeline; it accelerates lookup operations and leaves synchronization and other functions to the main core.
- We propose BOS to alleviate the branch misprediction bottleneck for lookup operations. BOS is based on the pre-execution of hard-to-predict data-dependent branches and is tightly

integrated with prefetching for its successful operation. BOS practically eliminates mispredictions in the studied workloads.
- Finally, we evaluate NT as a prefetcher and a pre-execution unit and quantify speedups due to alleviating memory and branch mispredictions. We show that combining accurate prefetching and branch prediction results in synergistic performance improvement.

The four NT variants: prefetcher-only (**ntpf**), prefetcher with BOS (**ntpf+bos**), pre-execution unit (**ntpx**), and pre-execution unit with BOS (**ntpx+bos**) evaluated in this article achieve 4.1×, 11.9×, 14.9×, and 18.8× speedups, respectively, on average, over a no-prefetching baseline. In contrast, AMAC [1] and ATP [11] provided 2.4× and 1.2× speedups, respectively. Other recent prefetchers (BO [16], ISB [18], and SPP [17]) do not offer significant performance gains for the workloads in this article. Although we did not evaluate the recent programmable ETP [5] prefetcher, we expect it to, at best, match the performance of ntpf (4.1×), NT's prefetch-only variant, since ntpf can eliminate almost all cache misses. Finally, our method scales well with the core-count. On average, with 24 cores, ntpx+bos achieves a speedup of 203× and 11× across all benchmarks over the single-core and 24-core baselines, respectively.

2 MOTIVATION

A lookup in a pointer-intensive data structure (e.g., a hash table or binary search tree, etc.) requires chasing pointers, resulting in low MLP as the next pointer cannot be discovered until the current access completes. Algorithm 1 shows a **probe hash table (PHT)** routine. Each key in a table of keys probes the hash table. Each hash table entry is a linked list of nodes searched for a key match. The lookup performance strictly depends on an arbitrary number of dependent memory accesses (i.e., number of pointers chased) required to locate an item. This also makes its branches HTP due to irregularities in the traversal path (e.g., early exit).

ALGORITHM 1: Pseudocode of PHT Algorithm

```
for key ← KEYS do                                  ▷ outer loop
    Node ← BUCKETS[hash(key)]
    while Node do                                  ▷ inner loop
        for key_node ← Node.Keys do
                                                   ▷ key-search loop
            Compare key with key_node
        end for
        if node_not_match then
            Node ← Node.next
        end if
    end while
end for
```

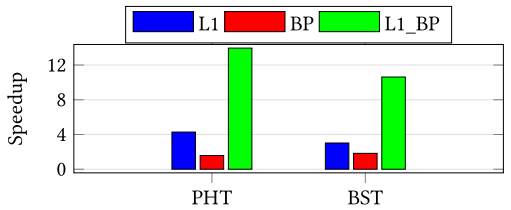

Fig. 1. The speedup of PHT and BST over the baseline. (L1: perfect L1D, BP: perfect BP, L1_BP: perfect L1D/BP).

Similarly, the performance of a lookup in a search tree (**binary search tree (BST)**) is directly related to the number of nodes traversed before finding a match. A single tree lookup is an inherently serial operation as the next tree node (i.e., child) to be traversed cannot be determined before the comparison in the current (parent) node is resolved. The data-dependent control flow also makes it hard to predict branches.

In-memory databases employ pointer-intensive data structures such as those in PHT and BST, and hence their performance is limited due to frequent cache misses and branch mispredictions. Fortunately, database operations such as indexing have abundant inter-lookup parallelism that can be exploited to increase the MLP extracted by each core.

In our experiments, we observed that both DL1 cache miss and branch misprediction rates were very high for PHT and BST. For a 64-KB L1 data cache, miss rates for PHT and BST were 68% and 62%, respectively. Also, with the state-of-the-art TAGE-SC-L [15] predictor, the branch **mispredictions per kilo instructions (MPKI)** for PHT and BST are 32 and 52, respectively. To understand the impact of cache and branch mispredictions, we simulated a perfect L1 cache (L1), perfect **branch prediction (BP)**, and both a perfect L1 cache and branch prediction (L1_BP). Figure 1 shows the results. A perfect cache improved the performance by 4.5× for PHT and 3.2× for BST. The perfect branch prediction impact was lower: A 1.6× performance boost for PHT and 1.8× for BST. Interestingly, when both BP and cache were perfect, the speedups for PHT and BST improved dramatically, reaching 14× and 11×, respectively.

This experiment shows that although eliminating cache misses is clearly more important at first sight, once the cache bottleneck has been removed, the relative negative impact of the branch mispredictions increases dramatically. For example, initially, perfect branch prediction improved baseline BST's performance by 1.8×. However, after cache misses were eliminated, BP boosted performance by 3.8×. This is an important observation, because, for an application with both high cache miss rate and high branch MPKI, the real performance potential of either bottleneck cannot be accurately measured when they are studied in isolation.

Since memory bottlenecks are more significant for in-memory database workloads, we first motivate our effort by comparing the most recent techniques that can potentially be used for hiding memory latencies for the workloads that we used in this study. Many recently proposed prefetchers, such as BO [16], SPP [17], and ISB [18], exploit dynamic memory access histories and are not effective for control-flow dependent LDS traversals, providing almost no speedup (up to only 3%) for the workloads in this article. Therefore, they are not included in the discussion.

Figure 2 illustrates how the most effective recent techniques impact key-value lookup executions (shown as lookups) on the CPU core and compares them with our proposed mechanism, NT. The key-value lookups on the CPU core are shown using numbers 0,1,2,3,. . . ,*n*. 0 denotes lookup number 0, 1 denotes lookup number 1, and so on. Execution time for each lookup on the CPU core is broken into three parts: stalls for data (shown in green), stalls for branch mispredictions (shown in red), and execution time with no stalls (shown in blue). All lookups are shown to execute in equal time for illustrative purposes. In the following, we explain each of the techniques presented in Figure 2 in the order they appear.

Continuous Runahead [12] can execute future lookups providing prefetches by running a stripped-down version of the traversal loop on a separate RISC core, but it runs the same algorithm as the main core and is subject to the same stalls. In Figure 2, Runahead is shown to start execution 4 lookups ahead of the CPU core. That is, when the core is executing lookup 0, Runahead engine is executing Lookup 4. When the core reaches Lookup 4, it can execute it faster, since the data have already been prefetched by the Runahead engine. Since the core is now faster in completing lookups, it eventually catches up with the runahead engine because of the serial nature of the traversals (linked data structures).

The recently proposed ETP by Ainsworth and Jones [5] is a sophisticated programmable prefetcher that employs many mini RISC cores to execute short program snippets to calculate prefetches on cache fill events (demand or prefetch requests). This method runs multiple lookups simultaneously (4 in the example) as shown in Figure 2, providing prefetches for the core. It can be more effective than the Runahead, because it can stay ahead of the core, since it handles multiple lookups simultaneously. However, it does not help with the stalls due to mispredictions. In addition, for more accurate prefetching, the original ETP in Reference [5] has to be extended to support the fetching of node keys and to make prefetching decisions on key comparisons. Otherwise, it

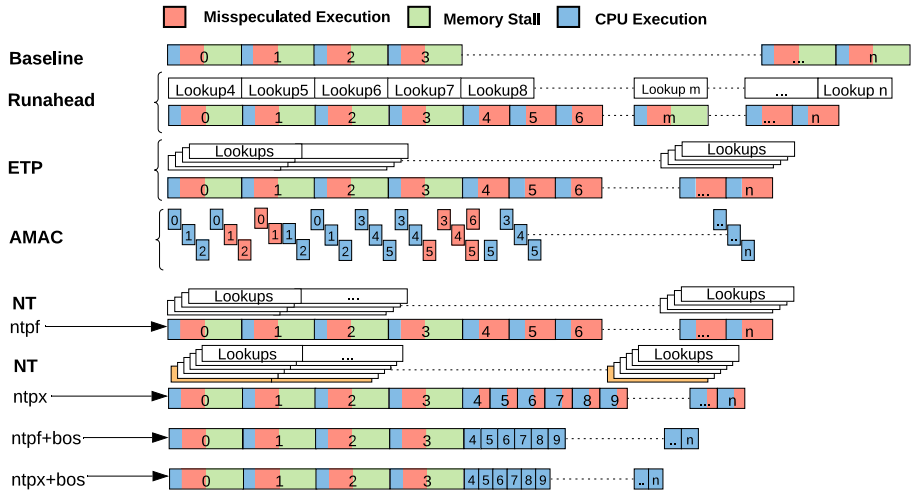

Fig. 2. Comparison of NT with runahead, ETP and AMAC. The CPU stalls due to cache misses and mispredictions are shown as red/blue boxes, respectively. Each hardware method runs four tasks ahead of the core. AMAC switches between multiple lookups and issues prefetches for future tasks at the cost of adding extra instructions per lookup. Runahead identifies future load misses but cannot effectively provide prefetching for load miss chains. ETP overlaps multiple tasks but with expensive operations per event. NT achieves ETP performance with ntpf but can execute more lookups with the addition of ntpx and BOS. (Note: figure data is not to scale.)

cannot be very effective for the workloads in this article because of potentially prefetching a large number of incorrectly predicted addresses.

AMAC [1] is a software method that is specifically designed to accelerate key-value lookups for in-memory databases. It works by employing a buffer to keep track of the full state of each in-flight traversal separately. A lookup executes by visiting each buffer entry multiple times, in a round-robin fashion, each time executing part of the lookup (and/or issues software prefetches for the next visit). In Figure 2, we show how lookups are executed with a 3-entry buffer. The figure shows how the CPU core can start lookup 1 before lookup 0 has been completed, overlapping multiple lookups by moving from one to the other after issuing a prefetch. AMAC overlaps multiple lookups effectively but requires substantial modifications to the source code and incurs significant instruction overhead due to state tracking. Furthermore, its effectiveness depends on the timeliness of software prefetches. Also, as Figure 2 shows the misspeculated execution would cause frequent squashes and re-execution of lookups.

Although these three recent methods are valid candidates for effective prefetching for key-value lookup traversals, each has the drawbacks described above. More importantly, as Figure 1 suggests, branch mispredictions are significant bottlenecks in the presence of prefetching alone, missing substantial performance potential. In fact, coordination of prefetching and branch pre-execution is necessary for the best performance gains. Prefetching can improve branch prediction by supplying data for pre-execution of future HTP branches—a mechanism that can be used as a helper for the branch predictor. Similarly, control-flow-dependent memory accesses cannot be predicted well due to HTP branches, but pre-execution of HTP branches can, in turn, improve prefetching accuracy.

Our proposed NT exploits inter-lookup parallelism and effectively integrates prefetching with branch pre-execution. Even without BOS, ntpf uses prefetched data to pre-execute branches, which

in turn guides prefetching. ntpf can effectively eliminate all cache misses and is expected to perform at least the same or better than Runahead, ETP, and AMAC, because it can eliminate or reduce their drawbacks. Figure 2 shows ntpf as capable of executing four lookups simultaneously and achieving performance similar to ETP. ntpf+bos can effectively pre-execute branches, overriding branch predictions in the core. By eliminating both memory stalls and branch mispredictions it can achieve unprecedented performance gains. Figure 2 shows that lookups in the core run much faster with ntpf+bos, since both branch mispredictions and cache misses are dramatically reduced or eliminated. In ntpx mode, the NT provides the key-matching node to the core and hence reduces the number of instructions to run for each lookup considerably (represented using smaller boxes in Figure 2), providing further speedups. We explain the details of NT and its operation modes in Section 3.

3 NODE TRACKER

We develop a novel prefetching/pre-execution scheme called the NT to accelerate key-value lookups for in-memory databases. To achieve this, NT employs mechanisms for accurate prefetching as well as mechanisms for BOS.

3.1 NT Architecture Overview

NT employs a mechanism that keeps 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. Similarly to recent prefetchers [3, 11] 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, where indicated in Algorithm 1. These special instructions simply record the traversal details into NT's configuration registers. The configuration information includes the PC of the load instruction that reads the target keys; encoded hash operations if any; base addresses and sizes of the traversed data structure (including the number of keys per node, key and pointer offsets, etc.); and the data structure type (e.g., hash table buckets, binary search tree).

To configure NT for the benchmarks tested, we have used at most 11 NT instructions. As mentioned, these instructions are executed only once, before the outer loop in Algorithm 1. Therefore, the impact of configuration on execution time is insignificant, considering millions to billions of key lookups.

Figure 3 shows the overall architecture of the NT. There are four main components: the **Key and Node-Address Provider (KNAP)**; which prefetches the target keys and the head nodes where the lookup should start (Section 3.3), the **Wait Queue (WQ)** that holds pending lookups to be executed (Section 3.4); the **Node Traversal Engine (NTE)**, which processes multiple in-flight lookups by maintaining their individual states in a **Task Buffer (TB)** (Section 3.5); and the **Result Buffer (RB)**, which holds the final lookup results (Section 3.6) to be consumed by the CPU.

3.2 Core Pipeline Modifications

The core pipeline is minimally changed to accommodate NT's operation. We added two new instructions: *ntcfg* and *read_RB*. *ntcfg* is the configuration instruction for the NT. Multiple *ntcfg* instructions are used to program the NT. A separate ID field in the *ntcfg* instruction is used by **NT's Configuration Unit (NTCU)**. As the configuration instructions are fetched by the core, they are forwarded to the NTCU at the decode stage of the core pipeline (not executed to configure NT before they are committed).

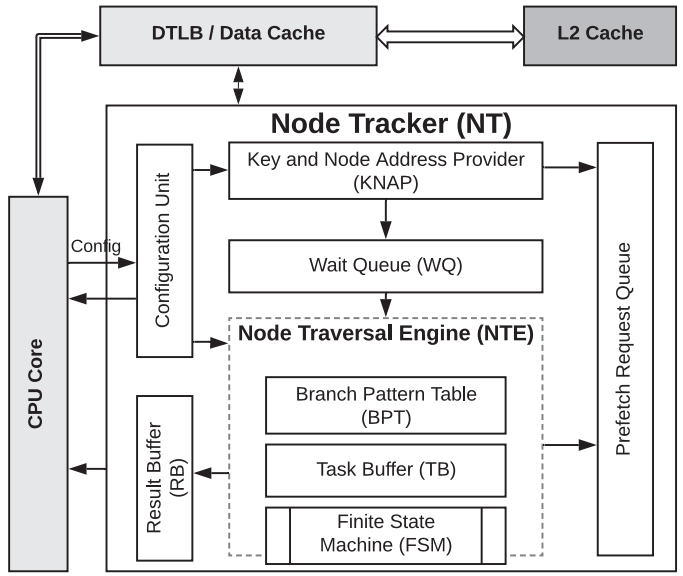

Fig. 3. Overview of the Node Tracker.

The NTCU processes *ntcfg* instructions with a specialized **Finite State Machine (FSM)**, which simply updates NT registers. However, NT's operation is initiated after the *ntcfg* instructions have been committed. This ensures that NT has not been configured speculatively after a branch misprediction. Once initialized, NTCU notifies the core to start detection of the instruction address that marks a new lookup (next outer loop iteration in Algorithm 1). This is initiated by setting three specialized registers at the fetch. A 1-bit *ntStart* register is set for activating NT's operation in the core. A 32-bit *itrPtr* register holds the least significant 32-bits of the PC of the instruction to detect the start of new lookups. A 6-bit counter, *itrCnt*, assigns a lookupID for each key lookup. Once *itrPtr* matches the least significant 32 bits of the fetch PC, *itrCnt* is incremented, and all following instructions are tagged with that lookupID until the next iteration has been detected. A lookupID field in the ROB is used to track iterations as tagged instructions are placed in the ROB.

On a branch misprediction, *itrCnt* must be restored to its value prior to the fetching of the mispredicted branch instruction by simply copying the latest valid lookupID from the ROB. There is nothing special to be done at the NT, since new lookups are only triggered by committed instructions.

NT's KNAP (Section 3.3) monitors committed load PCs (accessing the target-key array) and their corresponding addresses and lookupIDs to initiate prefetch address calculations. *read_RB* is the only new instruction that is executed in the core, as explained in Section 3.6. The most significant changes to the core are due to BOS, which is explained in detail in Section 3.7.

3.3 Key and Node-Address Provider

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 the lookup should start. KNAP is similar to the recent sophisticated programmable prefetchers ATP [11] and ETP [5]. Both ATP and ETP can prefetch irregular patterns—they were initially designed to target indirect access patterns. KNAP's implementation was made similar to ATP because of ATP's simplicity and good performance.

KNAP employs configurable registers and tables (see ATP [11]) that are configured by special instructions provided by compiler/programmer. It effectively creates a dependency graph between arrays using an array table (one entry for each array) and a relation table (keeps an entry for each relationship between two arrays). The prefetch is triggered whenever there is an access to the root array (which is the access to array B in A[B[i]]), and it creates the corresponding prefetch accesses (for both array B and array A in A[B[i]]). KNAP runs `N` iterations, or lookupIDs (32, in our evaluation) ahead of the demand access of the target key array. The lookup operation is triggered when the load instruction reading the target key array commits. Referring to Algorithm 1, KNAP prefetches the elements of the "Keys" array and the "Buckets" array that holds the head node pointers, by using the PC of the load instruction passed from the software.

KNAP is programmed using three instructions (same as in ATP [11, 64]): *ATCL*, *ATAR*, and *ATRL* (for more complex hashing behavior, a fourth instruction *ATOP* is utilized as described in References [11, 64]). *ATCL* clears all KNAP tables. *ATAR* insert entries in the array table. *ATRL* inserts relationship information between two arrays. For example, for the A[B[i]] structure, *ATRL* creates an entry in the relation table between target array A and index array B. *ATRL* has three operands: PC of target array, PC of index array and a 1-bit operand that specifies whether the array access is in the form of A[B[i]] or A[B[i][j]]. In Section 3.3, we show how to configure KNAP for the PHT algorithm. More details about ATP can be found in References [11, 64].

3.4 Wait Queue

KNAP is decoupled from NTE through a circular-FIFO 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 of 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 of the target keys. Lookup tasks wait in the WQ until they are fetched by the NTE.

3.5 Node Traversal Engine

The NTE consists of a TB to hold lookup tasks received from the KNAP/WQ, a FSM to process the tasks in the TB, and a **Branch Pattern Table (BPT)** to generate future branch outcomes and facilitate BOS.

The 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 status of the task in the TB entry. TB is a circular buffer whose entries keep the full status of each independent in-flight key lookup. The status of every task 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 `state` field indicates the current state to process. Finally, the `ptr` field points to the node being prefetched but not yet visited. Using the combination of `state` and `ptr` fields, the exact status of each in-flight lookup is preserved.

The lookup operations in TB entries are executed by an FSM, shared by all TB entries. The first step in processing a TB entry is to load its state (the latest `state` in its execution) into the local registers of the NTE. Execution starts from that `state`. 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, then a new memory prefetch is issued to the next linked node, and the TB entry's `ptr` field is updated with the address of the node that will be visited in the next `state`. 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, then a new lookup is initiated by fetching from the head of the WQ.

Table 1. Simplified FSM of NT (CS: Current State, NS: Next State)

CS	PHT, BST		NS	CS	Skip-List	NS
S0	Load new lookup from WQ		S1	S0	Load new lookup from WQ	S1
	Load num-keys:			S1	Get num-keys	S2
S1	PHT? Get num-keys in node		S2		Load node-key address, Update BOS	
	BST?		S2		key == node-key?	S4
	Load node-key address, Update BOS			S2	key != node-key?	
	key == node-key?				level >0?	S3
	Match?		S4		level == 0?	S4
S2	No match?				Load next-node address	
	PHT: More keys?	T →	S2		next-node is null?	
		F →	S3		Level >0?	S3
	BST:		S3		Level == 0?	S4
	Load next-node address, Update BOS			S3	More nodes?	S2
S3	next-node is null?		S4			
	more nodes to traverse? PHT:		S1			
		BST:	S2			
S4	Update RB with lookup result		S0	S4	Update RB with lookup result	S0

Table 1 shows the FSM of task execution for the benchmarks tested. There are five different states: (1) S0 is the starting state for all lookups in which a search key is loaded. (2) S1 loads the number of keys in each lookup. (3) S2 compares the key throughout the node traversal process. (4) S3 traverses to the next nodes. Finally, (5) S4 writes the result into the RB.

NT performs several memory accesses while transitioning between the FSM states. If a data load instruction (key, num-keys, node-key, next-node, etc.) leads to a cache miss, then NT has to issue a prefetch to bring the data into the L1 cache. However, to ensure reliable NT performance, each TB entry is associated with a (`pfState`) bit that is initially set to 1 when an access to the corresponding task entry is created, and it is set to 0 when the data are filled into the cache. If `pfState` is 0, then NT performs an access to the cache to read the data. If the data are in the cache, then they continue with the execution of the lookup. However, if it is a cache miss, then NT creates a memory access and switches to another task entry in the TB.

3.6 Result Buffer and NT as a Pre-execution Unit

As shown in Algorithm 1, a lookup operation consists of several node visits until a matching node is found. Since NT has already pre-executed these node visits, the CPU pipeline does not need to revisit the non-matching nodes again. A 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, then 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 a ***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.

3.7 Branch Outcome Streaming

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 BOS. We call `ntpf` (or `ntpx`) that also employs BOS as `ntpf+bos` (or `ntpx+bos`).

To configure BOS for NT, the compiler/programmer passes branch patterns for different execution path scenarios to hardware using special instructions. Each pattern is stored in a BPT, and each entry in the BPT is mapped to a unique case. As TB visits nodes in buckets, it refers to the BPT for observed cases and creates the BOS by appending bit patterns read from the BPT. Once the lookup has been completed, BOS for the corresponding lookup is stored in the RB. We now explain how BOS is generated for each lookup in PHT.

Code Snippet 1 shows the C code, and Code Snippet 2 the x86 assembly code for the PHT algorithm. Important branches (B1–B7) are marked in both the C code and the x86 assembly code. Table 2 shows unique cases for the PHT in Code Snippet 1, regardless of the number of nodes in a bucket. For example, the C1 and C1′ cases represent whether or not the node's first key is a match. Similarly, C2 and C2′ show whether or not the second key (or the next key) is a match or not. C3 determines if the next node exists, and C0 checks if the bucket exists (head node not NULL).

```
1  for (i = 0; i < rel->num_tuples; i++)                    B7
2  {
3    int matched = 0;
4
5    . . .
6
7    idx = HASH(rel->tuples[i].key, hashmask, skipbits);
8    bucket_t *b = ht->buckets + idx;
9    while (b){                                             B1,B4
10     for (j = 0; j < b->count; j++){                      B2,B5
11       if (rel->tuples[i].key == b->tuples[j].key){       B3,B6
12         matches++;
13         matched = 1;
14         break;
15     }
16   }
17     if (matched)
18     break;
19     b = b->next;
20 }
```

Code Snippet 1. C code of PHT algorithm.

Figure 4 shows the possible paths of execution for a node in a hash table bucket, using the cases in Table 2. As the TB completes each node visit, one of three execution paths can occur: 1) the first-key matches (C1), 2) the first-key does not match, and the second-key matches (C1′C2), or no match is found on the node (C1′C2′). To visit the next node, as shown in Code Snippet 1, the next node address is calculated and checked to see if it is not NULL, forming a pattern C3.

In this work, we have generated branch outcomes for each BOS case presented in Table 2 by employing a simple script that runs these cases using predetermined inputs. The configuration instructions are then hand inserted. However, both BOS and configuration instructions can be generated and inserted by a compiler automatically (e.g., can be implemented as LLVM [60] passes) similar to the work by Ainsworth and Jones [5].

Table 2. Branch Patterns for PHT in Code Snippet 1

Case	Sequence of branches	Pattern	Meaning
C0	B1	1	Initial head node check: bucket exists
C0′	B1	0	Head node NULL: lookup terminates
C1	B5, B6	00	First key in the node matches
C1′	B5, B6	01	First key in the node does not match
C2	B2, B3	01	Next key in the node matches
C2′	B2, B3	00	Next key in the node does not match
C3	B4	0	Next node exists
C3′	B4	1	Next node NULL: lookup terminates

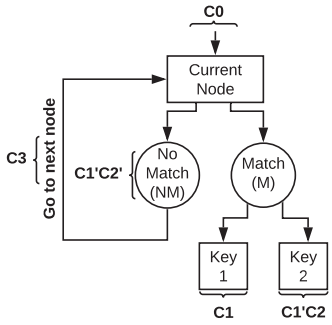

Fig. 4. Possible PHT execution cases.

```
1  [NT_Configuration_Instructions]
2  . . .
3       401a80: mov(%rx), %rdi
4  B1:  401a93: jne 401ab4
5       401a95: jmp 401ac5
6                ...
7       401aa0: cmp $0x1,%esi
8  B2:  401aa3: jbe  401aab
9       401aa5: cmp 0x18(%rdx),%rdi
10 B3:  401aa9: je  401ac1
11      401aab: mov 0x28(%rdx), %rdx
12      401aaf: test %rdx,%rdx
13 B4:  401ab2: je  401ac5
14      401ab4: mov 0x4(%rdx),%esi
15      401ab7: test %esi,%esi
16 B5:  401ab9: je  401aab
17      401abb: cmp 0x8(%rdx),%rdi
18 B6:  401abf: jne  401aa0
19      401ac1: add 0x1, %rax
20      401ac5: add 0x10, %r8
21      401ac9: cmp %r8,%r10
22 B7:  401acc: jne  401a80
```

Code Snippet 2. x86 assembly code of PHT algorithm.

Figure 5 shows how the BOS for each lookup can be generated by appending these cases. Two examples are shown. In the top bucket, a match is found on the second node's first key. In the bottom bucket, the third node's second key has a match. For the top bucket in Figure 5, the BOS generated is C0C1′C2′C3C1: C0 for the initial check for the head node, C1′C2′ for no key match in the first node, C3 for getting to the second node, and finally, C1 for the match in the first key of the second node. The branch pattern cases are known at compile-time, and bit patterns are stored in the BPT using the NT's configuration instructions. The TB refers to the BPT after a node visit has been completed and the case has been observed.

Although we only described how BOS worked for PHT, the same mechanism can be used for BST and Skip-List with minor modifications.

Whether NT is configured as a prefetcher (`ntpf`) or as a pre-execution unit (`ntpx`), BOS can be combined and integrated into NT to speedup key lookups. When NT acts as a pre-execution unit (`ntpx`), BOS generates a short sequence of patterns for each lookup, since the matching node is determined ahead of demand execution. This applies for PHT and BST workloads where the CPU starts executing from the matching node skipping all unnecessary comparisons, which results in a sequence of bits that consist of the starting branch outcomes combined with the corresponding matching pattern. However, for Skip-List, BOS is expected to produce long patterns even with the matching node being identified. The reason is that the algorithm will still traverse down between the levels until it reaches the node that contains the matching key.

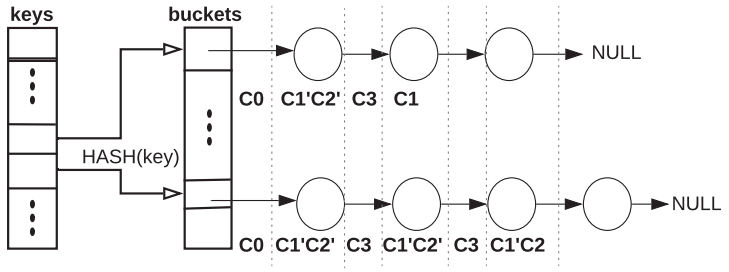

Fig. 5. Branch Outcome Stream Generation for two lookups in PHT. For this example, we assume a maximum of 4 nodes for each bucket and exactly two keys per node. For the bucket on top, there is a key match in the second node's first key. In the bottom bucket, a match has been found in the third node's second key. The branch outcome streams generated for each lookup are shown. For example, for the lookup in the top bucket, C0C1′C2′C3C1 is the branch outcome stream generated where C0–C3 are different branch pattern cases listed in Table 2.

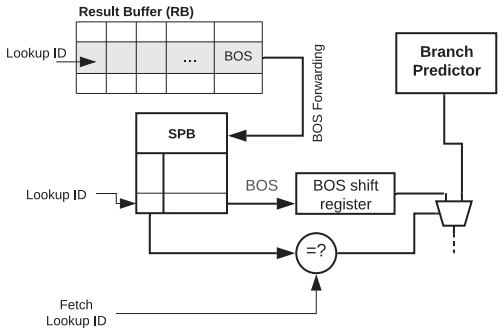

Fig. 6. BOS integration with the branch predictor.

The integration of BOS into the Branch Predictor: 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 (0x401a80) of the outer loop in Code Snippet 2). When a branch misprediction is detected, the lookupID counter at fetch can be restored using the latest valid instruction's lookupID.

Once the TB has completed a task, it updates the RB with the matching node's address and sets the ready flag for the RB entry, at which point, the BOS for that lookup is also ready to be used. The RB forwards the BOS with its lookupID (in program order) to a 4-entry FIFO buffer called the **Stream Predictions Buffer (SPB)**. The SPB is physically located near the I-cache and the branch predictor for fast access. For every new lookupID at fetch, if that lookupID matches the lookupID of the oldest entry of the SPB, then the BOS in this entry is placed in a shift register, and the branch outcomes from it override all branch predictor's predictions for that key lookup until a new lookup has been reached at fetch. Figure 6 shows how BOS is integrated with the branch predictor.

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. Furthermore, 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.

Table 3. List of NT Internal Registers

NT Register	Description
NTRM	Holds the type of data structure to be traversed. (Linked-List, Binary Tree, Skip-List)
NTR0	Holds the PC of the keys array's load instruction
NTR1	Holds the PC of the bucket array's load instruction, which reads the head node address
NTR2	Holds the distance between elements in key array
NTR3	Holds num-keys offset (offset from the node's address to the value where the number of keys is stored).
NTR4	Holds the key array offset value at which the key array within each node is located from the node address
NTR5	Holds the offset needed to calculate the address of the next-pointer within each node.
NTR6	Single-bit that defines whether the number of keys per node is dynamic or fixed.
NTR7	Holds the key element size.

For simplicity, the descriptions correspond to PHT Algorithm.

Table 4. Configuration Instructions: Configuring NT for PHT Algorithm

```
ntcfg   NTRM, 0x0        ; configure NT mode to run a linked-list
ntcfg   NTR0, 0x401a80   ; pass the PC of the keys array's load instruction which reads the key value
ntcfg   NTR1, 0x401a90   ; pass the PC of the bucket array's load instruction which reads the head
                           node address
ntcfg   NTR2, 0x10       ; pass the distance between elements in keys array
ntcfg   NTR3, 0x4        ; number of keys stored inside each node is located at 0x4 bytes from the
                           node address
ntcfg   NTR4, 0x8        ; the offset value to be used to calculate the address of the node's keys array
ntcfg   NTR5, 0x28       ; node address + this offset (0x28) is used to get the address of next pointer
ntcfg   NTR6, 0x0        ; inform NT that the number of keys within each node is dynamic.
ntcfg   NTR7, 0x10       ; pass the key element size in bytes
```

3.8 Example: Configuring NT for the PHT Workload

NT has two programmable units: KNAP and NTE. Detailed description of how to program KNAP can be found in Cavus et al. [64]. NTE employs several configuration registers (NTRM, NTR0-7), which are set by a newly introduced *ntcfg* instruction. *ntcfg* instructions are inserted in the code before the main key-value lookup loop, as shown in Code Snippet 2. These instructions are fetched by the core and flow through the pipeline. They are, however, forwarded to NT's configuration unit only after they are known to be non speculative (i.e., oldest instructions in the instruction window). *ntcfg* instruction has two arguments. The first argument is the NT register id (e.g., NTR0, the destination), and the second argument is the source value. The details of NTE's configuration registers are shown in Table 3. *NTRM* register holds the type of data structure that NT is configured for (BST (0), PHT (1) or Skip-List (2)). *NTR0-7* are general-purpose and their use depends on the data structure. The detailed explanations given in Table 3 are for the PHT algorithm. Table 4 shows the sequence of *ntcfg* instructions used to configure the NTE for the PHT workload.

Table 5 shows the sequence of instructions that are used to program the KNAP (see Section 3.3) for the PHT workload. To program the KNAP, the load instruction PCs for the keys and the buckets

Table 5. Configuring KNAP to Issue Indirect Prefetches

atcl	; clear AT tables
atar 0x401a80	; pass PC of key array (tuples[].key)
atar 0x401ab4	; pass PC of hash array
atrl 0x401a80, 0x401ab4, 0	; pass the relationship between tuples and hash arrays

arrays as well as the relation information between these arrays are needed. Although not shown in the example, *atop* instructions [64] can be used for implementing complex hashing operations when needed.

4 DISCUSSION

Exceptions and Faults: Most common faults encountered by NT are TLB misses, and they are handled by the host core's MMU in its usual fashion. Since the memory request that causes the fault is reissued when the corresponding NT task buffer entry is revisited, there is no need for the core MMU to signal NT to retry after the missing translation is available. This greatly simplifies the process. Other types of faults and exceptions trigger handler execution on the host core, and the problem causing lookup is dropped from the task buffer entry, and lookup is executed in the core. An alternative implementation may prevent NT from triggering any exceptions by simply dropping lookup tasks from NT. This would slightly reduce NT's performance potential.

Support for KVS writes: In single-threaded mode, key insertion and deletion operations can also be supported with minor modifications to `ntpx`. NT's multi-threaded mode with the BOS versions would require a complicated mechanism to support KVS writes. In the existing implementation only prefetching is allowed in those cases. This is done by the compiler/programmer simply not inserting the ***read_RB*** instruction into the code. In this article, we only consider key-value lookups without updates, allowing the employment of both `ntpx` and `ntpf`, with and without BOS.

Simultaneous Multi-Threading (SMT): NT assumes that each core runs a single thread. In an SMT core, if only one thread uses NT, then we just need to add the thread id to NT. However, if two or more threads on the same core need to use NT, then additional hardware and mechanisms would be needed. SMT support is beyond the scope of this article.

Why is NT not an attached accelerator? Our focus in this article was not to design an accelerator as is usually done in computer architecture research, but to show that prefetching and branch outcome streaming together could result in performance as good as if not better than a plain accelerator, especially in the case of in-memory database lookups. Also, with NT, the core is not idle and can run any functions on the payload before all the lookups have been completed – that is, operations on a record can run in the core while lookups are simultaneously carried out in the NT, further improving performance.

NT as a prefetcher for other applications: Although the NTE component is workload-specific, the KNAP component of NT can also be used for other applications. KNAP can currently prefetch for stride and indirect access patterns and be used without activating the NTE component (that targets multiple lookup traversals). However, KNAP would not be effective alone for the workloads in this article. Finally, NT can co-exist with other prefetchers or accelerators without significant complexities. On the whole, NT is more general than a typical accelerator.

5 METHODOLOGY

Processor parameters: NT is implemented on the gem5 simulator [8]. We modeled an aggressive out-of-order core with a 256-entry instruction window size and a 20-stage pipeline, implementing the x86 ISA. We assume a 64-KB private L1D with 4-cycle latency and 256-KB unified L2 cache

Table 6. Processor Parameters

Parameter	Value
ISA	x86
Number of cores	1-24
Architecture	2GHz OoO: 4-wide, ROB:256, LQ/SQ:96/64
Branch Predictor	TAGE-SC-L BP
L1D cache	64KB, 8-way, 4-cycles latency, 64Byte block, MSHR:24
L2 cache	256KB, 8-way, 12-cycles latency, MSHR: 24
L3 cache	Multi-core : 16-way, 32-cycles latency, 1 shared L3/4-cores, 1MB/1core, MSHR:48 Single-core : 1 L3 (4MB)
Memory	8GB DDR3, 800MHz, 4 MCs (2 channels / MC)

Table 7. Hardware Cost of NT

Component	Size/Count	Bytes
KNAP	1	512
RB	32-entries	1284
TB	16-entries	388
WQ	16-entries	128
Misc. Reg	1	16
NTE	1	334
BPT	8-entries	11
SPB	4-entries	144
ROB Inst. Counters	6-bit/Ins	192
Total Size = 2.94KB		

with 12-cycle latency, both with 64-B blocks and 8-way set-associativity. We used 24 **miss-status holding registers (MSHRs)** for the L1D cache. This is on par with the recent Intel (e.g., Sunny Cove [9]) and AMD (e.g., Family 17h [7]) processors. Finally, our baseline configuration used the state-of-the-art TAGE-SC-L [15] branch predictor.

The multicore system is modeled after the AMD Epyc SoC [10]. Multiple CPU complexes are initiated at the SoC level. Each CPU complex is four cores connected to an L3 cache. The L3 cache is 16-way associative, 4 MB, 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. The architectural parameters of the evaluation environment are summarized in Table 6.

Workloads: In this study, we use three commonly used data structures for database indexing, PHT, BST and Skip List. We focus on indexing, because it is the most significant part of the execution time for the end-to-end query operation. Kocberberger et al [42] showed that TPC-H and TPC-DS queries on MonetDB spend, on average, 35% and 45% of their execution times, respectively, on indexing.

PHT: Hash tables are commonly used in modern databases for accelerating indexing operations. The hash table lookup throughput is the main bottleneck, and its performance depends on the number of pointer chasing memory accesses required to locate an item. When build relation keys follow a skewed value distribution, hash collisions are unavoidable. Probing such hash table buckets requires as many memory accesses as the number of hash table nodes present in that bucket. One cannot guarantee a constant number of memory accesses for each probe. For the PHT workload, we use the highly optimized chained hash table implementation of Balkesen et al. [55, 56]. Each hash table bucket contains two 16-byte tuples and an 8-byte pointer to the next hash table node to be used in the case of collisions.

BST: Tree index search is a fundamental operation in database systems to handle large datasets. The performance of a lookup in a search tree is directly related to the number of nodes traversed before finding a match. A single tree lookup is an inherently serial operation as the next tree node to be traversed cannot be determined before the comparison in the current node is resolved. Due to the control flow and large datasets, traversing BST cause frequent memory stalls. We use a canonical implementation of a binary search tree by Balkesen et al. [55, 56]. The tree was built by using an input relation with uniformly distributed random keys and payloads. Each binary tree node contains an 8-byte key, an 8-byte payload and two 8-byte child pointers (i.e., left and right). The probe relation contains uniformly distributed random unique keys.

Skip List: Skip list is another commonly used data structure for indexing in database systems. Fast search is made possible by maintaining a linked hierarchy of sub-sequences, with each

successive sub-sequence skipping over fewer elements than the previous one. For the skip list workload, we adopt the implementation from ASYCLIB [19]. In our implementation, both the build and probe relation contain uniformly distributed random unique keys and payloads.

We simulated three variants of PHT (PHT-B2, PHT-B4, and PHT-B8) with two, four, and eight nodes per bucket, respectively. The number of keys per node is two for PHT, one key per node for BST and Skip-List. The number of buckets for PHT is 2^{22} while the depth of the unbalanced tree is 65 for BST. In Skip-List, the number of levels is 9. The number of lookups is 2^{25} for all workloads. Each lookup finds exactly one node match with a uniformly distributed random unique key.

For all the workloads, we start a simulation from the beginning of the region of interest, which is the beginning of the outer loop traversing the target keys. We simulated 100M instructions, with 10M instructions used for warm-up. For multi-core simulations, each core is simulated for 100M instructions.

Hardware budget: NT is evaluated with a 16-entry WQ, 16-entry TB, 8-entry BPT, and 32-entry RB. KNAP is about 0.5 KB in size while the total size of the NT is only about 3 KB. The hardware budget of NT is summarized in Table 7.

6 PERFORMANCE EVALUATION

In this section, we evaluate the performance of NT in terms of the overall throughput, **million keys per second (MKPS)**, and MPKI. NT is evaluated both as a prefetcher `ntpf` and as a pre-execution unit `ntpx`, both with and without `BOS`. It is compared to both a baseline (no prefetching) and the recent prefetchers ATP [11], BO [16], SPP [17], and ISB [18], and the AMAC method [1]. Although we have not implemented the recent programmable ETP [5], we expect it to, at best, match the performance of `ntpf`, since `ntpf` can eliminate almost all cache misses, and because NT's accuracy and timing are expected to be better than ETP's. The latter is true, since ETP does not pre-execute branches to guide prefetching decisions.

All the simulations are evaluated on single and multi-core systems. We demonstrate the substantial performance gains of NT over the state-of-the-art, as well as BOS's ability in many cases to practically eliminate branch mispredictions.

6.1 Single-Core Analysis

NT is first evaluated assuming it is implemented in a single-core processor. NT works well both as a prefetcher and as a pre-execution unit.

6.1.1 NT as a Prefetcher. The plots on the left side of Figure 7 show the throughput speedup in MKPS normalized to baseline of the different prefetchers on a single-core processor. The plots on the right side indicate the branch mispredictions in MPKI. `ntpf` has a large speedup over the baseline, improving the performance by 2.6×, 4.3×, 6.5×, 3×, and 2.4× for PHT-B2, PHT-B4, PHT-B8, BST, and Skip-List, respectively. `ntpf` significantly outperforms AMAC and other prefetchers tested for all workloads.

ISB and BO provide no speedup for the evaluated workloads. SPP provides a marginal 1.03× speedup, on average. These state-of-the-art prefetchers were tested using their original configurations. Because they provide insignificant speedups, we have not analyzed them any further for the remainder of the article.

ATP benefits from prefetching stride and indirect array accesses but it cannot prefetch for LDS traversals. ATP provides 1.45× speedup for PHT-B4 but no speedup for BST and Skip-List. AMAC allows multiple traversals to be performed concurrently; however, it also has significant instruction overhead (2.2×–3.5×) and its performance depends on the timeliness of software prefetches. AMAC's speedup for PHT-B4, BST, and Skip-List are 2.26×, 2.54×, and 1.27×, respectively.

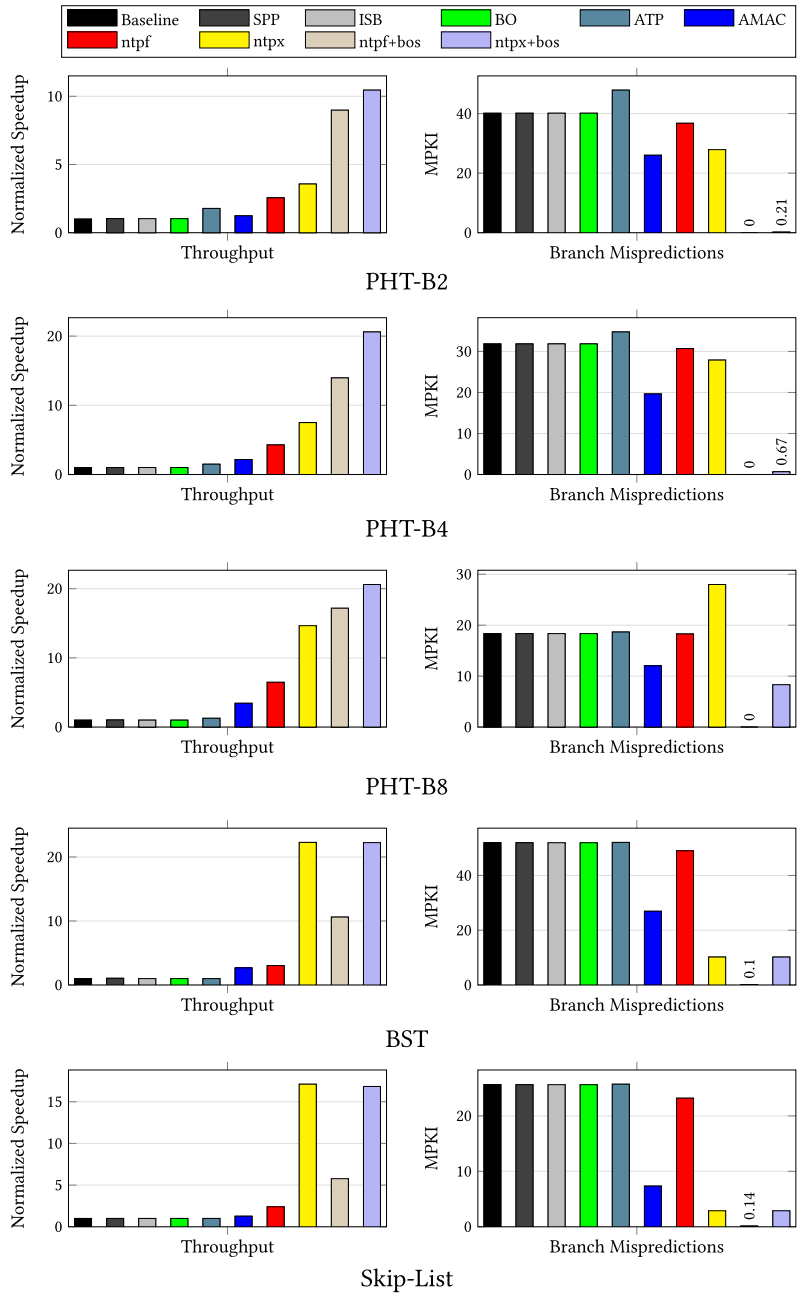

Fig. 7. Normalized throughput (MKPS: million keys/sec) compared to baseline, and branch MPKI.

The branch bottleneck and ntpf+bos: In the PHT workload, the MPKI increases as the number of nodes per bucket decreases due to accessing more linked lists when the number of nodes is small, resulting in mispredicting the last iteration within a linked list traversal more often. For example, the baseline's MPKI decreases from 40 in PHT-B2 to 31 in PHT-B4 down to 18 when 8 nodes are used in each linked list (i.e., PHT-B8). In Skip-List, AMAC's MPKI is almost one third

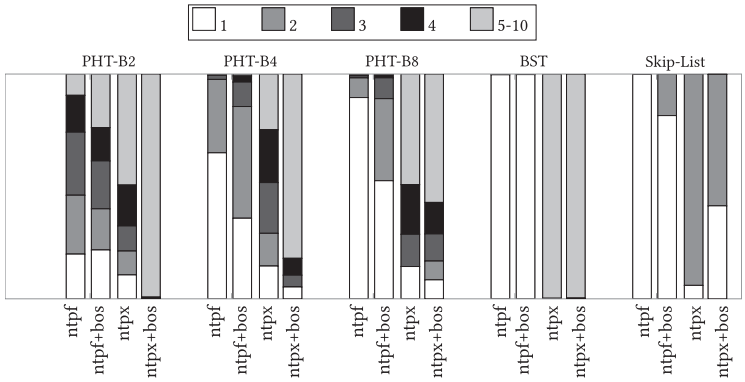

Fig. 8. The number of in-flight lookups in the CPU instruction window at a snapshot in time. We observe the same distribution in many snapshots and present only one. However, the numbers should be very close to the average number of in-flight lookups over many snapshots.

less than baseline, ntpf and ATP. This is due to the extra instructions (2.8× extra) added to the original algorithm while the number of conditional branches is fixed. In contrast, ntpf does not alter the algorithm and executes more lookups than AMAC.

Since MPKI is very high for baseline and ntpf in all workloads (MPKI of BST is 52), eliminating these mispredictions has a noticeable impact on the throughput. To demonstrate this we simulated a perfect branch predictor on ntpf. ntpf+perf improves the throughput by 9.04×, 14.09×, 17.5×, 11.52×, and 6.05× for PHT-B2, PHT-B4, PHT-B8, BST, and Skip-List, respectively. The baseline also benefits from eliminating mispredictions achieving up to 2× for PHT-B2, and marginally more than 1.4× to 1.96× for the other workloads.

Adding BOS to ntpf can improve the throughput remarkably. ntpf+bos can effectively generate branch results for each lookup ahead of their demand fetch, thus eliminating almost all branch mispredictions. ntpf+bos consequently increases MKPS by 3.6×, 3.3×, 2.6×, 3.5×, and 2.4× over ntpf for PHT-B2, PHT-B4, PHT-B8, BST, and Skip-List, respectively. As shown in Figure 7, branch MPKI is reduced to less than 0.8 for all workloads and nearly to zero for Skip-List. For BST, ntpf provides only slightly better speedup than AMAC (3.1× versus 2.5×) (see Figure 7, on the left). However, both AMAC and ntpf are significantly bottlenecked by branch mispredictions. ntpf+bos provides 10.5× speedup over the baseline and 3.5× and 4.2× speedups over ntpf and AMAC, respectively.

6.1.2 NT as a Pre-execution Unit. Using NT as a pre-execution unit, ntpx, improves the performance dramatically for all workloads by reducing the number of instructions executed per lookup.

Figure 7 shows that ntpx increases the throughput by 3.6×, 7.5×, 14.7×, 22.3×, and 17.2× over the baseline, and by 1.4×, 1.8×, 2.3×, 7.4×, and 7.2× over ntpf for PHT-B2, PHT-B4, PHT-B8, BST, and Skip-List, respectively.

Similarly to ntpf, ntpx is extended with BOS (ntpx+bos). Figure 7 shows that ntpx also has high MPKI for PHT (especially because ntpx reduces the number of instructions per lookup), therefore, providing branch outcomes will likely enhance the performance. ntpx+bos improves ntpx by 2.9×, 2.8×, and 1.4× for PHT-B2, PHT-B4, and PHT-B8, respectively. For BST and Skip-List workloads, BOS does not increase the throughput further, because ntpx has already significantly lowered MPKI compared to ntpf.

ntpx+bos performs better than ntpf+bos, because it reduces the number of instructions executed per lookup reducing the pressure on CPU buffers. Figure 8 shows that the number of in-flight lookups in the instruction window is significantly higher for ntpx over ntpf for BST and Skip-List.

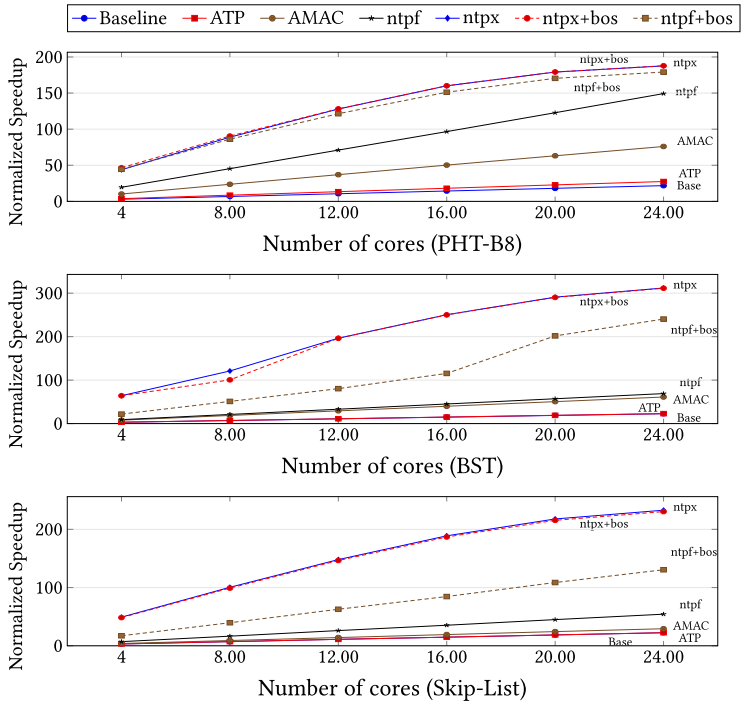

Fig. 9. Effect of varying number of cores on performance normalized to single-core baseline. PHT-B2 and PHT-B4 results are not shown, because the trend is similar to PHT-B8.

6.2 Multi-Core Evaluations

We evaluate the performance of NT on 4–24 cores for all variants of NT, AMAC, and ATP. The results are depicted in Figure 9. As the number of cores increases, `ntpf` and `ntpx` scale very well achieving up to 55×–150× throughput speedup with `ntpf`, and 82×–312× with `ntpx` over the single-core baseline. Exploiting BOS in a multi-core processor also affects the performance positively. BST benefits from branch outcomes strongly on a multi-core with a throughput speedup of 312× compared with the baseline on a single-core. On average, `ntpx+bos` achieves 203× throughput speedups (over single-core) for all workloads on 24 cores.

6.3 The Impact of Prefetching Distance

The prefetch distance in NT defines how far ahead of the demand execution to trigger a lookup prefetch. Increasing the distance may result in early prefetches causing high cache misses, while short distances impact the timeliness and conflicts with demand executions. Since all workloads show the same behavior on 1–24 cores, we only show how the distance affects NT for the BST benchmark on a single-core. Figure 10 shows the sensitivity of performance to prefetch distance on `ntpf`, `ntpx`, `ntpx+bos`. As the distance increases from 8 to 32, NT eliminates cache misses proportionally.

6.4 The Effect of Number of L3 MSHRs

Figure 11 plots the throughput of all methods for different L3 MSHR sizes. `ntpx` exhausts more than 16–32 MSHR entries on a single core, which makes the cache stop accepting new memory requests.

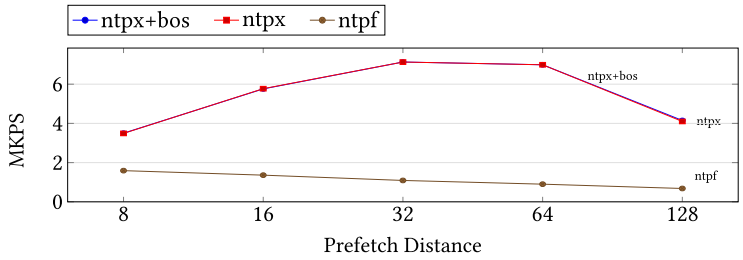

Fig. 10. Effect of prefetching distance for BST on single and 16 cores. Other workloads show the same behavior on 1–24 cores.

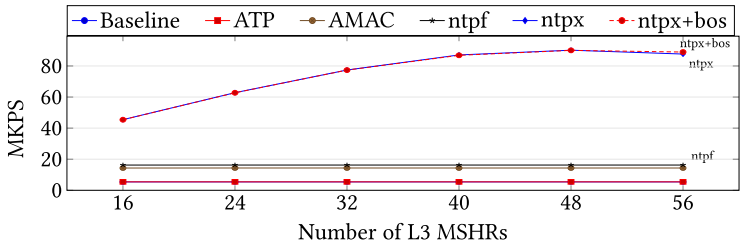

Fig. 11. Impact of the number of L3 MSHRs for BST. PHT-B2, PHT-B4, PHT-B8, and Skip-List are similar. Lower unlabeled overlapping lines correspond to baseline, AMAC, and ATP.

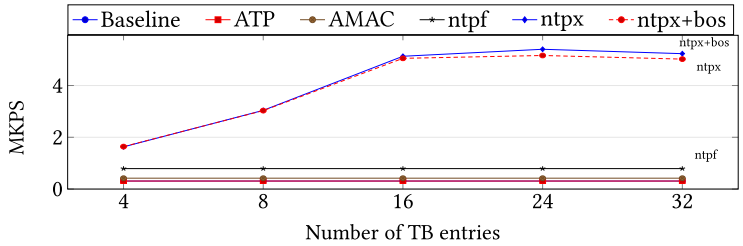

Fig. 12. Impact of task buffer size for Skip-List on single and multi-core processors. Lower unlabeled overlapping lines correspond to baseline, AMAC, and ATP.

In our design, the typical number of L3 MSHRs is 48 after which NT behaves steadily (i.e., the MSHR size is large enough to service all of the memory requests issued by NT). However, NT as a prefetcher (`ntpf`) requires fewer memory requests than `ntpx`, showing a constant performance with 16–56 MSHR entries. The reason `ntpx` exhausts more than 16–32 MSHRs is that, with `ntpx`, the core will execute iterations very efficiently, since key-matching nodes are provided by NT. This will push KNAP to issue prefetches more frequently (for future lookup keys and head nodes), because KNAP keeps a fixed 32-iterations distance from the core. As TB receives lookup tasks more frequently, TB entries fill quickly. Once TB is full and all entries are stalled due to cache misses (because of each TB entry's prefetch requests), NT exhausts more than 32 MSHRs as TB and WQ both fill their 16 entries.

6.5 The Effect of Task Buffer Size

Each entry in the TB corresponds to a separate lookup to be performed by NT. Figure 12 shows the Skip-List throughput of all methods for varied TB sizes on single/multi-core. Increasing TB size more than 16 has no impact on the performance as NT has to keep up with demand execution,

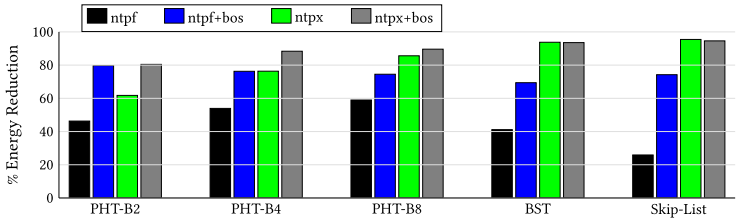

Fig. 13. Dynamic energy reduction normalized to baseline.

and service these lookups before dropping their tasks from TB entries. Other workloads share the same behavior and use 16 TB entries for close to the best results.

6.6 Energy Efficiency

We use McPAT [20], assuming a 45-nm process technology, to measure the overall dynamic energy reduction for all NT variants. Figure 13 shows the energy estimates. Energy usage is significantly reduced with all NT variants. Most improvement comes with `ntpx+bos` as it provides best performance and reduces the number of instructions per lookup significantly. `ntpx+bos` achieves 80% to 95% energy reductions over all workloads.

7 RELATED WORK

Recent hardware prefetchers targeting regular/repetitive patterns: Recent proposals in data prefetching include AAMPM [22], BO [16], SPP [17], ISB [18], VLDP [23], Bingo [24], and most recently, IPCP [25]. AMPM, BOP, and IPCP are the first-, second-and third-place winners of the 2019 data prefetching competition [26], respectively. All of these prefetchers use temporal and spatial behavior they observe at runtime to make predictions. They are successful in regular repetitive patterns; however, they are not effective for complex data structure traversal behavior that we target in this article. Finally, BFetch [27] leverages control flow prediction to generate an expected future path of the executing application. It then speculatively computes the effective address of the load instructions along that path. However, BFetch is unsuccessful for workloads where branch MPKI is high.

Software Prefetching: Software prefetching [28–33] provides a way for programmers to insert prefetching instructions into a program targeting various simple and complex patterns. In Ainsworth [34], while the insertion of software prefetches for indirect memory accesses is automated and eliminates the requirement for programmer effort, it cannot guarantee insertion of the instructions in an optimized way for a specific architecture. Furthermore, significant instruction overhead may offset its benefits. Finally, Lee et al. [28] studied the interaction between software and hardware prefetching and found that inserting software prefetching instructions in the presence of hardware prefetchers may hurt the over-all prefetching performance due to the incorrect training of hardware prefetchers. NT does not have this problem, because prefetching is only initiated by hardware, not by software prefetch instructions; coarse-grain metadata instructions are used to guide the hardware prefetcher.

Helper Threads: Helper threads are separate threads that are used to prefetch future data accesses, and they are useful for applications with irregular memory access patterns. Kim and Yeung [35] developed a compiler-based approach for creating helper threads that can capture irregular memory accesses. Lau et al. [36] proposed a small helper core, and Ham et al. [37] provide a different scheme to create separate access and execute threads. Ganusov and Burtscher [38] proposed a lightweight architectural framework for efficient event-driven software emulation of complex

hardware accelerators, which can be applied to implement a variety of prefetching techniques. In general, helper threads are unable to deal with stalling on intermediate loads created by the accesses of indirect or LDS memory access chains, making it difficult for them to keep ahead of the main thread.

Accelerators: Most of the recent work proposes specialized hardware accelerators to efficiently execute targeted irregular workloads. Ho et al. [39] proposed to encode memory accesses as a set of rules to allow them to be mapped to a dataflow architecture. Kumar et al. proposed sqrl [40] and dasx [41] to accelerate iterative accesses of B-tree and hash table structures. Lloyd et al. [54] proposed a near-memory accelerator emulated in an FPGA that combines hardware and software to service multiple lookups in a hash table-based key/value store. Kocberber et al. [42] designed an accelerator for database inner joins exploiting parallel hash table walks.

Although accelerators often provide large performance improvements, the original application has to be modified, making it incompatible with devices not having the accelerator.

Prefetchers targeting Irregular Patterns: Prefetching techniques targeting pointer-based applications have been studied in References [43–48]. Guided Region Prefetching [48] is a hardware/software scheme that uses compiler hints encoded in load instructions to regulate an aggressive hardware prefetching engine. Indirect memory prefetching has been studied in References [34, 49]. ATP [11] and ETP [5] have significantly improved indirect prefetching with software-provided course-grain metadata.

CR execution [12] proposed a complex mechanism to dynamically identify the dependence chain of a load that is likely to create a cache miss. It can accurately prefetch data needed in the near future. However, CR cannot provide effective prefetching for load miss chains, which prevents CR from running sufficiently ahead.

Ainsworth and Jones' [5] ETP uses event-triggered programmable RISC cores, each of which executes a compiler-generated subprogram to calculate a prefetch address for a prefetching trigger event. This requires relatively expensive operations per event. NT is a more specialized yet programmable prefetcher/pre-execution unit that integrates branch streaming with prefetching, important for in-memory database lookups.

Hashemi et al. [63] propose to pass the dependent operations between a source miss and a dependent miss to the **(enhanced) memory controller (EMC)**, where it is executed immediately after the source data arrives from DRAM. EMC can help improve the single lookup performance by cutting round-trip travel of data between the core and the memory controller. Our NT provides the ability to perform multiple lookups simultaneously. In that sense, the methods are orthogonal.

Branch Pre-execution: Today's best-known branch predictors, such as TAGE-SC-L [15] and Multiperspective Perceptron [50], push the envelope of what is possible using dynamic branch histories. Yet, for some programs, branch history alone cannot provide very high accuracy. Prior work in References [51, 52] recognized the need to sometimes correlate on program values. Gao et al. [51] developed the ABC (address-branch correlation) predictor specifically for HTP branches that depend on loads that miss in the L2 cache. Al-Otoom et al. [52] proposed the EXACT predictor targeting all address branch correlations by actively communicating the value changes in these addresses to the branch predictor. More recently, Farooq et al. [53] proposed the SLB predictor targeting store-load-branch correlations by providing compiler hints to the branch predictor.

Prior work also explored pre-execution for HTP branches. Dundas et al. [59] employs runahead execution using a stripped-down version of the traversal loop to stay ahead of the main thread, providing future branch outcomes for the latter. However, since it runs the same algorithm as the main thread, it is subject to the same stalls and, therefore, cannot stay ahead. Slipstream [57] proposes to run a shorter version of the program (called A-stream) ahead of the original one (called R-Stream) by removing ineffectual computations. A-Stream supplies R-stream with control and

dataflow outcomes. For LDS traversals whose control-flow is highly unpredictable, A-stream fails to stay ahead of R-stream. NT with BOS, however, is able to exploit MLP and hence stays ahead of the main thread.

Control-Flow Decoupling [58] is a pre-execution method for so-called separable branches, whose backward slices do not depend on their control-dependent instructions. This is not applicable to our study, where the targeted branches are very serial in nature: A dynamic branch guards instructions that are predicates of future dynamic branches.

BFetch [27] leverages control flow prediction to speculatively compute the addresses of the load instructions along that path. However, BFetch is only successful when branch prediction is highly accurate, which is not the case for our workloads.

Chen et al. [61] propose a mechanism to speculatively execute the instruction stream using the feedback from the branch predictor. When a branch is encountered, the prefetching unit predicts the most likely path to be executed and starts fetching these instructions ahead of the regular program counter. All branches and correct paths are recorded on a log, and this log is fed to the CPU in demand execution. The PC (halt execution) is reset if misprediction happens as they use the predicates produced by branch prediction (which is not 100% accurate). Finally, Ferdman et al. [62] propose a method to use control flow information to do accurate instruction prefetches. In both cases, the methods are only effective when there is no hard-to-predict branch in the loop, which is the case for the workloads that we studied.

8 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 exploits 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 rest of the main core.

We evaluated NT as a prefetcher and an accelerator and quantified the speedups due to alleviating memory or branch mispredictions. 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 4.1×, 11.9×, 14.9×, and 18.8× speedups, respectively, over a no-prefetching baseline. Furthermore, our method scales well with an increasing number of cores. With 24-cores of `ntpx+bos`, we achieve a speedup of 203× and 11× over the single-core and 24-core baselines, respectively.

REFERENCES

[1] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263.

[2] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (Aug. 2007), Article 17. DOI: https://doi.org/10.1145/1272743.1272747

[3] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'10)*.

[4] Changhee Jung, Daeseob Lim, Jaejin Lee, and Y. Solihin. 2006. Helper thread prefetching for loosely-coupled multiprocessor systems. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*.

[5] S. Ainsworth and T. M. Jones. An event-triggered programmable prefetcher for irregular workloads. *ACM SIGPLAN Not.* 53, 2 (Mar. 2018), 578–592. DOI: https://doi.org/10.1145/3296957.3173189

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, 105–117.

[7] AMD. 2017. Software Optimization Guide for AMD Family 17h Processors, 2.8.1.6. https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[9] WikiChip Fuse. 2019. Intel Sunny Cove Core to Deliver a Major Improvement in Single-Thread Performance, Bigger Improvements to Follow. Retrieved from https://fuse.wikichip.org/news/2371/intel-sunny-cove-core-to-deliver-a-major-improvement-in-single-thread-performance-bigger-improvements-to-follow/.

[10] AMD. 2019. Retrieved from https://www.amd.com/en/products/epyc-server.

[11] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. 2018. Array tracking prefetcher for indirect accesses. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'18)*. 132–139. DOI: 10.1109/ICCD.2018.00028

[12] M. Hashemi, O. Mutlu, and Y. N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. DOI: 10.1109/MICRO.2016.7783764.

[13] Mingxing Tan, Xianhua Liu, Tong Tong, and Xu Cheng. 2012. CVP: An energy-efficient indirect branch prediction with compiler-guided value pattern. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 111–120.

[14] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.

[15] Andre Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *Proceedings of the 5th Championship on Branch Prediction*.

[16] P. Michaud. 2016. Best-offset hardware prefetching. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 469–480.

[17] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path confidence based lookahead prefetching. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1–12.

[18] A. Jain and C. Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 247–259.

[19] William Pugh. 1990. Concurrent Maintenance of Skip Lists. Technical Report. University of Maryland at College Park, College Park, MD, 1990.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 469–480.

[21] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.* 49, 2, Article 35 (Aug. 2016), 35 pages. DOI: https://doi.org/10.1145/2907071

[22] Y. Ishii, M. Inaba, and K. Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the International Conference on Supercomputing (ICS'09)*. 499–500.

[23] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 141–152.

[24] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*, 399–411.

[25] S. Bakalapati and B. Panda. 2019. Bouquet of instruction pointers: Instruction pointer classifier based hardware prefetching. In*Third Data Prefetching Competition.* ISCA 2019. IEEE Press, 118–131. DOI: https://doi.org/10.1109/ISCA45697.2020.00021

[26] Retrieved from https://dpc3.compas.cs.stonybrook.edu.

[27] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. 2014. B-Fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Los Alamitos, CA, 623–634.

[28] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.* 9, 1 (Mar. 2012), Article 2.

[29] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. 1995. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, 232–236.

[30] Luk and Mowry. 1996. Compiler-based prefetching for recursive data structures. *ACM SIGOPS Rev.* 30, 5 (1996), 222–233.

[31] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, 40–52.

[32] Mowry. 1994. Tolerating Latency through Software-controlled Data Prefetching. Ph.D. Dissertation, Stanford University, 1994.

[33] Youfeng Wu, Mauricio Serrano, Rakesh Krishnaiyer, Wei Li, and Jesse Fang1. 2002. Value profile guided stride prefetching for irregular code. In *Compiler Construction*. Springer, 307–324.

[34] Ainsworth and Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'17)*. IEEE Press, 305–317.

[35] Kim and D. Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 159–170. DOI: https://doi.org/10.1145/635508.605415

[36] E. Lau, J. E. Miller, I. Choi, D. Yeung, S. Amarasinghe, and A. Agarwal. 2011. Multicore performance optimization using partner cores. *Proceedings of the USENIX Conference on Hot Topics in Parallelism (HotPar'11)*.

[37] T. J. Ham, J. L. Aragón, and M. Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 191–203. DOI: 10.1145/2830772.2830800

[38] I. Ganusov and M. Burtscher. 2006. Efficient emulation of hardware prefetchers via event-driven helper threading. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*.

[39] C.-H. Ho, S. J. Kim, and K. Sankaralingam. 2015. Efficient execution of memory access phases using datafow specialization. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 118–130. DOI: 10.1145/2749469.2750390

[40] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips. 2014. Sqrl: Hardware accelerator for collecting software data structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 475–476. DOI: 10.1145/2628071.2628118

[41] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan. 2015. Dasx: Hardware accelerator for software data structures. In *Proceedings of the International Conference on Supercomputing (ICS'15)*. 361–372. DOI: https://doi.org/10.1145/2751205.2751231

[42] O. Kocberber, B. Grot, J. Picorel, B. Falsaf., K. Lim, and P. Ranganathan. 2013. Meet the walkers: Accelerating index traversals for inmemory databases. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*.

[43] E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th International Symposium on High Performance Computing Architecture (HPCA'09)*, 7–17.

[44] A. Roth, A. Moshovos, and G. S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS'98)*, 115–126.

[45] C.-L. Yang and A. R. Lebeck. 2000. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing (ICS'00)*. 176–186.

[46] J. Collins, S. Sair, B. Calder, and D. Tullsen. 2002. Pointer cache assisted prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, 62–73.

[47] Amir Roth and Gurindar S. Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE Computer Society, Los Alamitos, CA, 111–121.

[48] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, 388–398.

[49] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, 178—190.

[50] Daniel Jimenez. 2016. Multiperspective perceptron predictor. *Championship Branch Prediction Competition* (2016).

[51] H. Gao et al. 2008. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-14)*. 74–85. DOI: 10.1109/HPCA.2008.4658629

[52] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: Explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF'10)*. ACM, New York, NY, 165–176.

[53] M. U. Farooq, Khubaib, and L. K. John. 2013. Store-load-branch (SLB) predictor: A compiler assisted branch prediction for data dependent branches. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*, 59–70.

[54] Lloyd, G. Scott and Maya Gokhale. 2017. Near memory key/value lookup acceleration. *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*. 26–33. DOI: https://doi.org/10.1145/3132402.3132434

[55] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2014. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.* 27 (2014), 1–1. 10.1109/TKDE.2014.2313874.

[56] C. Balkesen, J. Teubner, G. Alonso and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. 362–373. DOI: 10.1109/ICDE.2013.6544839

[57] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. *Sigplan Not.* 35 (2000), 257–268. 10.1145/356989.357013.

[58] R. Sheikh, J. Tuck, and E. Rotenberg. 2012. Control-flow decoupling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 329–340. DOI: 10.1109/MICRO.2012.38

[59] James David Dundas and Trevor Mudge. 1998. *Improving Processor Performance by Dynamically Pre-processing the Instruction Stream*. Ph.D. Dissertation. University of Michigan, USA.

[60] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization* 2004 (2004), 75–86. DOI: 10.1109/CGO.2004.1281665

[61] Cheng K. Chen, Chih-Chieh Lee, and T. N. Mudge. 1997. Instruction prefetching using branch prediction information. In *Proceedings of the International Conference on Computer Design VLSI in Computers and Processors*. 593–601. DOI: 10.1109/ICCD.1997.628926.

[62] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. Association for Computing Machinery, New York, NY, 152–162. DOI: https://doi.org/10.1145/2155620.2155638.

[63] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating dependent cache misses with an enhanced memory controller. *SIGARCH Comput. Archit. News* 44, 3 (Jun. 2016), 444–455. DOI: https://doi.org/10.1145/3007787.3001184.

[64] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. 2020. Informed prefetching for indirect memory accesses. *ACM Trans. Archit. Code Optim.* 17, 1, Article 4 (Mar. 2020), 29 pages. DOI: https://doi.org/10.1145/3374216

Received August 2020; revised February 2021; accepted February 2021