Asynchronous Memory Access Chaining

Asynchronous Memory Access Chaining

Onur Kocberber
EcoCloud,EPFL
onur.kocberber@epfl.ch

Babak Falsafi
EcoCloud,EPFL
babak.falsafi@epfl.ch

Boris Grot
University of Edinburgh
boris.grot@ed.ac.uk

ABSTRACT

In-memory databases rely on pointer-intensive data structures to quickly locate data in memory. A single lookup operation in such data structures often exhibits long-latency memory stalls due to dependent pointer dereferences. Hiding the memory latency by launching additional memory accesses for other lookups is an effective way of improving performance of pointer-chasing codes (e.g., hash table probes, tree traversals). The ability to exploit such inter-lookup parallelism is beyond the reach of modern out-of-order cores due to the limited size of their instruction window. Instead, recent work has proposed software prefetching techniques that exploit inter-lookup parallelism by arranging a set of independent lookups into a group or a pipeline, and navigate their respective pointer chains in a synchronized fashion. While these techniques work well for highly regular access patterns, they break down in the face of irregularity across lookups. Such irregularity includes variable-length pointer chains, early exit, and read/write dependencies.

This work introduces Asynchronous Memory Access Chaining (AMAC), a new approach for exploiting inter-lookup parallelism to hide the memory access latency. AMAC achieves high dynamism in dealing with irregularity across lookups by maintaining the state of each lookup separately from that of other lookups. This feature enables AMAC to initiate a new lookup as soon as any of the in-flight lookups complete. In contrast, the static arrangement of lookups into a group or pipeline in existing techniques precludes such adaptivity. Our results show that AMAC matches or outperforms state-of-the-art prefetching techniques on regular access patterns, while delivering up to 2.3x higher performance under irregular data structure lookups. AMAC fully utilizes the available micro-architectural resources, generating the maximum number of memory accesses allowed by hardware in both single- and multi-threaded execution modes.

Proceedings of the VLDB Endowment, Vol. 9, No. 4

1. INTRODUCTION

In recent years, server memory capacity has increased dramatically, reaching the point where database tables and supporting auxiliary structures can reside completely in memory. Pointer-intensive data structures (e.g., hash tables, trees) are essential for enabling sub-linear access time to in-memory data. In this context, vast datasets overwhelm on-chip caches and unpredictable access patterns due to dependent pointer dereferences frequently leave the CPU waiting on a long-latency memory access. Consequently, the performance bottleneck for many database operations is accessing main memory [2, 21].

A modern CPU employs multiple out-of-order cores, which are designed to hide the memory access latency by identifying and issuing multiple independent memory accesses. The number of in-flight memory accesses at a given point in time is called memory-level parallelism (MLP). The amount of MLP is dictated by the number of independent memory operations within the instruction window of the processor core. A lookup in a pointer-intensive data structure (e.g., a hash table) may require chasing pointers, resulting in low MLP as the next pointer cannot be discovered until the current access completes.

Fortunately, many database operations that leverage pointer-intensive data structures (e.g., hash join, index join, and group-by) have abundant inter-lookup parallelism that can be exploited to increase the MLP extracted by each core. Exploiting such inter-lookup parallelism within a core is difficult due to the limits on instruction window size imposed by technology [1]. As a result, recent proposals have examined software prefetching techniques that exploit inter-lookup parallelism through loop transformations. State-of-the-art software prefetching approaches for database systems work by arranging a set of independent lookups into a *group* (Group Prefetching [8]) or *pipeline* (Software-Pipelined Prefetching [8, 16]), in effect synchronizing their memory accesses into highly structured sequences.

These approaches work well whenever the number of pointer dereferences is known ahead of time and is constant across lookups, in which case the group size or the number of pipeline stages can be provisioned to perfectly accommodate the memory access pattern. Whenever such perfect knowledge or regularity is not present, some lookups may exhibit *irregularity* with respect to the expected or average case. Examples of irregularity include variable number of nodes per bucket in a hash table, early exit (e.g., on a match of a unique key), and read/write dependencies that require se-

rialization of subsequent accesses via a latch. When such irregularities occur, existing software prefetching techniques must execute expensive and complex cleanup or bailout code sequences that greatly diminish their effectiveness.

We observe that many real-world execution scenarios involving pointer chasing entail irregularity across lookups. Achieving high MLP in these circumstances requires a degree of dynamism that is beyond the capability of today's software prefetching techniques. To overcome the existing capability gap, this work introduces Asynchronous Memory Access Chaining (AMAC), a new software prefetching scheme that avoids the need for rigidly arranging independent lookups into a group or a pipeline. By preserving and exploiting the lack of inter-dependencies across lookups, AMAC is able to attain high MLP even for highly irregular access patterns.

AMAC achieves its dynamism by maintaining the state of each in-flight lookup separately from that of other lookups. State maintenance operations are explicit, meaning that once a prefetch is launched, the state associated with that lookup is saved into a dedicated slot in a software-managed buffer, at which point a different lookup can be handled by loading its respective state. By decoupling the state of all in-flight lookups from each other, AMAC enables unprecedented flexibility in initiating, completing, and waiting on lookups. Such flexibility directly translates into high MLP, as potential memory access opportunities are not wasted due to common issues such as variable-length pointer chains.

Our contributions are as follows:

- We corroborate prior work showing that Group Prefetching and Software-Pipelined Prefetching attain considerable single-thread speedups of 2.8x-3.8x over a baseline of a highly optimized no-prefetching hash join of 2^{27} uniformly distributed unique relation keys ($2GB \bowtie 2GB$). However, in the presence of irregularity across lookups, these techniques lose much of their performance advantage. On the same no-prefetching hash join, when the relation keys follow a skewed key distribution, the performance difference between the best-performing technique and a no-prefetch baseline is just 39%.
- AMAC is highly robust, achieving a competitive 4.3x speedup over the no-prefetching baseline for uniform lookups and maintaining its performance advantage in the presence of irregular accesses. In the skewed scenario, AMAC improves the performance by up to 2.8x over the no-prefetch baseline and by up to 2.3x over the existing techniques.
- AMAC is able to fully utilize the available hardware MLP resources. For the single-thread case, the achieved MLP is constrained by the number of L1 data cache misses that can be in-flight at once; for the multi-thread/multi-core case, the achieved MLP is bottlenecked by the number of outstanding last-level cache misses.

2. MOTIVATION

2.1 Pointer-Intensive Data Structures

This section explains the use of common pointer-intensive data structures – namely, hash tables and trees – in database operations.

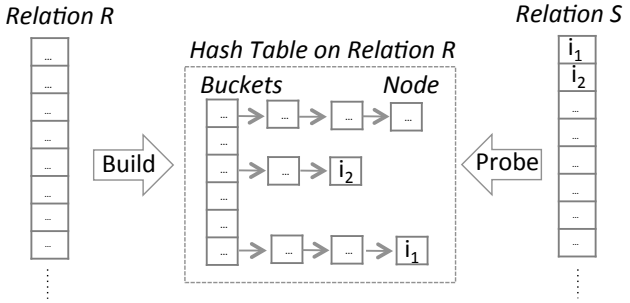


Figure 1: Hash join.

2.1.1 Hash Tables

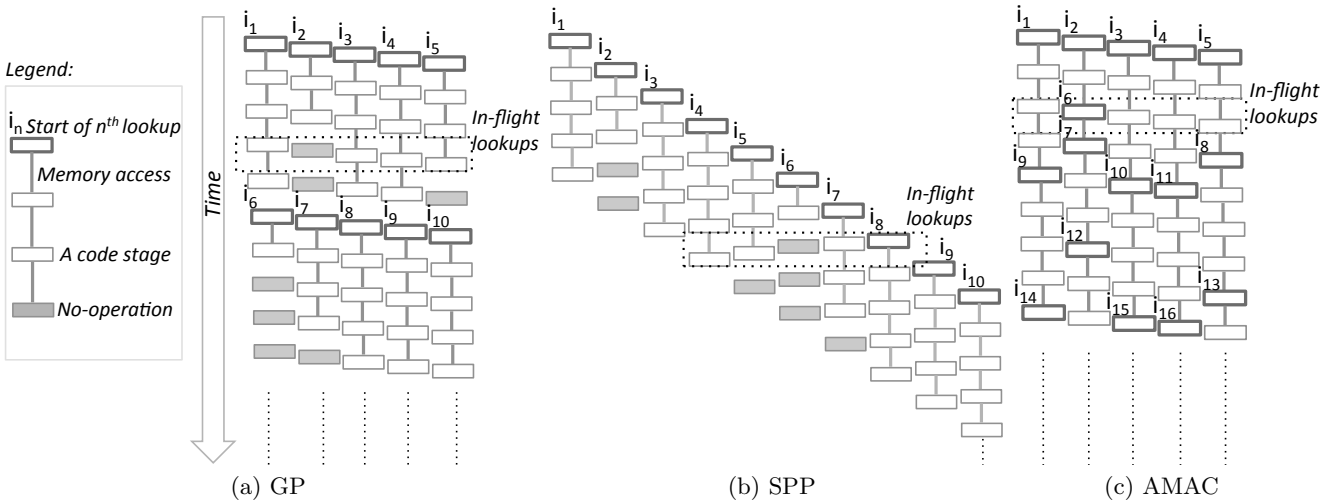
Hash tables are prevalent in modern databases for accelerating data-finding and grouping operations. We consider the use of hash tables for two frequent database operators: hash join and group-by.

Figure 1 depicts the use of hash tables in a hash join operator, which locates the matching entries in a pair of relations. Hash table lookup throughput is the main bottleneck of the join operation, and its performance strictly depends on the number of dependent memory accesses (i.e., number of pointers chased) required to locate an item. A lookup in the hash table can result in an arbitrary number of memory accesses as state-of-the-art hash tables offer a tradeoff between performance (i.e., number of chained memory accesses) and space efficiency [4, 6, 7]. Moreover, when the build relation keys follow a skewed value distribution, hash collisions are unavoidable as some build keys are identical but carry different payloads. Probing such hash table buckets requires as many memory accesses as the number of hash table nodes present in that bucket. The bottom line is that it is not possible to generalize a single type of hash table layout or guarantee a constant number of memory accesses for each probe.

Another use of hash tables in database systems is the group-by operator, which collects payloads of an input relation and groups them according to relation keys. Similar to the hash-join build phase, each payload in the input relation is added into a hash table. However, the difference is that in the case of non-unique keys, the common case for group-by, the matching hash table node is located first and then either the payloads are added to a separate list pointed to by the hash table node (i.e., late aggregation) or the necessary aggregation function is applied immediately on the payload of the matching node. As a result, depending on the group-by scenario, hash table layout, and relation cardinality, the number of memory accesses per tuple might differ significantly.

2.1.2 Tree Search

Tree index search is a fundamental operation in database systems to handle large datasets with low latency and high throughput. 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 (i.e., child) to be traversed cannot be determined before the comparison in the current (parent) node is resolved. The combination of branching control flow and large datasets leads to frequent memory stalls due to low cache and TLB locality. There are numer-



(a) GP (b) SPP (c) AMAC

Figure 2: Execution patterns of Group Prefetching (GP), Software-Pipelined Prefetching (SPP) and Asynchronous Memory Access Chaining (AMAC). Gray boxes indicate no-operations due to traversal divergences.

ous proposals to optimize the layout of the index trees to improve their locality characteristics [10, 16, 23]. However, crossing the cache block and VM page boundary, which is unavoidable, still incurs significant memory access latency penalties.

2.2 Hiding Memory Access Latency

2.2.1 Software Prefetching Techniques for Pointer-Chasing Database Operations

The state-of-the-art pointer-chasing prefetching techniques – namely, Group Prefetching (GP) and Software-Pipelined Prefetching (SPP) [8] – exploit inter-lookup parallelism to improve the performance of hash table operations. SPP has also been applied to balanced search trees [16]. Both GP and SPP are loop transformations that break down a loop with N dependent memory accesses into a loop that contains $N+1$ code stages, where each stage consumes the data from the previous stage and prefetches the data for the next stage. To hide the memory access latency by doing useful work, Group Prefetching executes each code stage for a group of M lookups[1], thereby performing a maximum of M independent memory accesses at a time. Similarly, Software-Pipelined Prefetching forms a pipeline of $N+1$ stages; each code stage in the pipeline belongs to a different lookup and a single lookup completes after going through $N+1$ pipeline stages. In the cases where N is too small to hide the memory access latency, the pipeline is initiated with a prefetch distance so that M independent memory accesses ($M \geq N$) are performed in-flight.[2]

Obviously, neither of the techniques is parameter-free as they require the number of stages, N, and the number of in-flight lookups, M, to be determined ahead of time. Setting M is relatively easy, as it is a function of the underlying hardware's MLP capabilities; specifically, the maximum number of outstanding L1-D cache misses that can be in flight at once.[3] In contrast, N is a data structure- and algorithm-specific parameter, which makes both GP and SPP vulnerable to irregularities in the execution because N explicitly structures the execution pattern of all the M lookups, which leads to three issues:

1. Lookups might require less than N stages, due to the irregular data structure layout (e.g., probes i_1 vs i_2 in Figure 1). A similar situation could also occur on a regular structure when certain lookups terminate earlier than others (i.e., early exit after finding a match). Regardless of the reason, when the actual number of stages is less than N, the remaining code stages must be skipped (i.e., no-operation) for that lookup.

2. Lookups might require more than N stages as the data structure is irregular (e.g., unbalanced trees or due to bucket collisions in a hash table). These cases require a bailout mechanism to complete the lookup sequentially.

3. Lookups might have a read/write dependency on each other (e.g., hash table build or update). When this occurs, the actual code stage and the subsequent ones should be executed later, when the dependency is resolved.

Whenever any of the above cases occurs, the maximum M will not be reached, necessarily lowering MLP. Over-provisioning M does not help, as it increases the number of no-operations (as explained in #1 and #2 above) or the likelihood of an inter-dependency (as in #3).

Figure 2a illustrates the GP execution of ten independent lookups (i_{1-10}), where the number of code stages required and the maximum number of in-flight lookups are five ($N = 4, M = 5$). Each white box indicates a code stage and lines connecting the boxes indicate a memory access (prefetch) produced in the earlier code stage and consumed in the later

[1]This parameter is referred to as G in the original work [8].
[2]This parameter is referred to as D in the original work [8]. Hence, $M = N * D$.
[3]In microarchitectural terms, MLP is constrained by the number of L1 Miss Status Handling Registers (MSHRs).

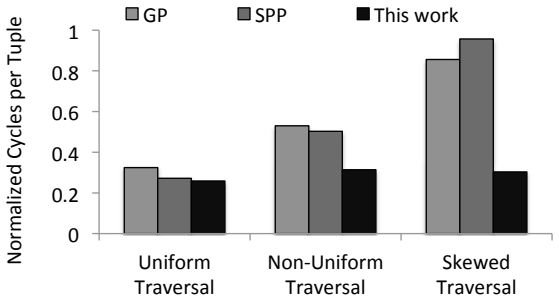


Figure 3: Normalized cycles per lookup tuple on Xeon x5670.

code stage. The dashed box depicts all the lookups within a group ($i_1 - i_5$) going through the same code stage. Once all the stages complete for all the lookups in the group, a new group of lookups is initiated ($i_6 - i_{10}$). Figure 2b depicts the same example for SP; the main difference is that one iteration (shown in a dashed box) contains different code stages for different lookups.

The execution patterns of $i_{1,3,4,8,9,10}$ for both GP and SPP in Figure 2 perfectly match the parameter N as there are four memory accesses (five code stages). In contrast, the lookups $i_{2,5,6,7}$ turn out to be irregular as they terminate at different stages of the execution. To handle such complexities, GP and SPP maintain status information per lookup so that code stages can be skipped (necessary for correctness), resulting in a loss of extracted MLP and a waste of CPU cycles checking and propagating the status of completed lookups. The other irregularities, such as lookups that turn out to require more than N accesses or have a read/write dependency (not shown in the example), are more difficult to handle and require special "clean-up" passes and/or bailout mechanisms.

2.2.2 Performance Analysis of Software Prefetching

In order to understand the performance impact of the irregularities, we implemented GP and SPP for hash table probes. Our baseline implementation leverages a chained hashed table with linked lists used in recent hash join studies [4, 5]. The first hash table node is clustered with the bucket header as shown in Figure 1. We run the experiments on a Xeon x5670 with uniformly distributed random 2^{27} lookup tuples (8B key and 8B payload) corresponding to 2GB in total. We report cycles spent per tuple key lookup normalized to baseline code with uniform lookups. In all experiments, we pick the best performing configuration, which is $M = 15$ for GP and $M = 12$ for SPP.

We perform three experiments, namely *uniform, non-uniform* and *skewed* traversals by populating the hash table with 2^{27} build relation tuples. Throughout the experiments, the average number of node traversals per lookup is almost four, but the hash table bucket occupancy and the traversal algorithm vary. In the uniform case, each hash table bucket contains exactly four nodes and each lookup traverses all the nodes in a bucket. In the non-uniform case, we relax the assumption that each hash table should contain four nodes and we terminate the key search upon a match by assuming the build keys are unique. Finally, in the skewed traversals build keys follow a Zipf-skewed distribution (Zipf factor = .75). Therefore, some hash table buckets contain more nodes than the others, and 1% of the hash table buckets, which are the most populous, contain 19% of the total build tuples. This situation may arise in hash joins when the join key is a non-unique attribute in the build relation.

Figure 3 shows the results of the performance experiment. On uniform traversals, GP and SPP achieve impressive speedups of 3.1x and 3.7x, respectively, over the no-prefetch baseline thanks to their ability to fully reach their MLP potential. On non-uniform traversals, GP and SPP are 1.6x and 1.8x worse in terms of cycles per lookup compared to the uniform lookups, due to the wasted code stages on lookups that terminated early. Finally, on skewed traversals, we observe that GP and SPP perform 2.6x and 3.5x worse compared to the uniform lookup case, delivering virtually no improvement over the uniform baseline.

Throughout the experiments, we also observe that the performance of SPP compared to GP fluctuates by $\pm$ 20%. The inconsistent performance of existing techniques thus underscores the need for a robust software prefetching solution.

3. ASYNCHRONOUS MEMORY ACCESS CHAINING

The main drawback of Group Prefetching and Software-Pipelined Prefetching is the static staging of all in-flight memory accesses. In effect, the set of lookups comprising these accesses are coupled within a group or pipeline, resulting in artificial inter-dependencies across the otherwise independent lookups. This coupling is the reason for the lack of robustness in existing prefetching techniques in the face of irregularity in the data structure (e.g., unbalanced tree) or in the traversal path (e.g., early exit).

This work introduces Asynchronous Memory Access Chaining (AMAC), a new prefetching scheme whose distinguishing feature is the ability to deal with irregular and divergent memory access patterns. AMAC accomplishes this by preserving the independence across lookups, thus avoiding the coupling behavior that plagues existing techniques. Figure 2 shows a cartoon comparison of AMAC to the existing prefetching techniques.

3.1 Design Overview

The core idea of AMAC is to keep the full state of each in-flight memory access separate from that of other in-flight accesses. Whenever an access completes for a lookup, a new access for the same lookup can be initiated without any knowledge of the state of other accesses. If the completed memory access was the last one for a given lookup (e.g., in the case of a key match), a new lookup sequence can be started with similar ease and, again, without any regard to the state of other lookups.

Figure 4 shows the key components of the proposed scheme in the context of a hash table probe in a hash-join operation. All in-flight requests are kept in a software-managed circular buffer, whose total number of entries is sufficient to cover the memory access latency. Once a lookup has been initiated, its state is saved in one entry of the circular buffer. This state, comprised of the five fields shown in Figure 4, contains all the information necessary to continue or terminate the lookup. The *key* field contains the lookup key and used for node comparisons throughout the lookup. Upon a key match, the *rid*(*idx*) and *payload* fields are used for output materialization. The *stage* field indicates the appropriate code stage to execute. Finally, *ptr* points to the node

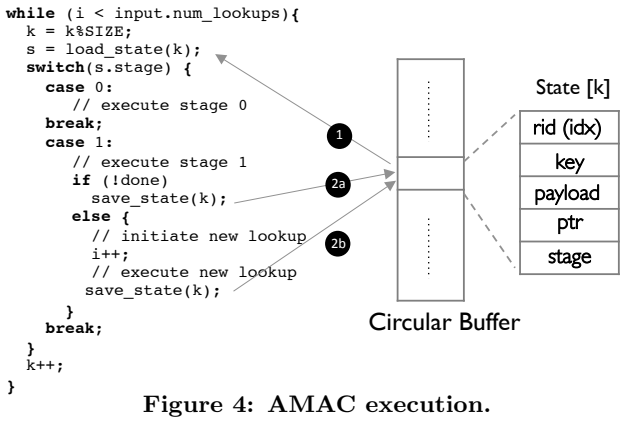


Figure 4: AMAC execution.

```
1  struct state_t {
2  int64_t idx;
3  int64_t key;
4  int64_t pload;
5  node_t  * ptr;
6  int32_t stage;
7  };
8  /* Hash table probe loop  */
9  void probe
10   (table_t *input, hashtable_t *ht, table_t *out) {
11 state_t s[SIZE];
12 node_t  * n;
13 int32_t   k, i;
14 /* Prologue */
15 //...
16 /* Main loop */
17 while (i < input->num_keys){
18   k = (k == (SIZE-1)) ? 0 : k;
19   if (s[k].stage == 1){
20     n   = s[k].ptr;
21     /* Code 1: Output matches or visit next node */
22     if (n->key == s[k].key){
23       out[s[k].idx] = n->pload;
24       s[k].stage = 0;
25     } else if (n->next){
26       prefetch(n->next);
27       s[k].ptr = n->next;
28     } else {
29       /* initiate new lookup (Code 0) */
30     }
31   } else if (s[k].stage == 0){
32     /* Code 0: Hash input key, calc. bucket addr. */
33     int64_t hashed = HASH(input->tuple[i].key);
34     bucket_t * ptr = ht->buckets + hashed;
35     /* Prefetch for next stage */
36     prefetch(ptr);
37     /* Update the state */
38     s[k].idx   = ++i;
39     s[k].key   = input->tuple[i].key;
40     s[k].ptr   = ptr;
41     s[k].stage = 1;
42     /* Optionally fetch payload to emit results */
43     s[k].pload = input->tuple[i].pload;
44   }
45   k++;
46 }
47 /* Epilogue */
48 //...
49 }
```

Listing 1: AMAC hash table probe pseudo-code.

being prefetched but not yet visited. Using the combination of *stage* and *ptr* fields, the exact status of each in-flight lookup is preserved.

To execute a code stage of a lookup, the first step is to load a single in-flight request from the circular buffer. As a single in-flight request is read from the circular buffer, the state of the lookup is loaded into the local variables of the software thread (step 1 in Figure 4). Once the state is loaded, the execution starts by jumping to the necessary code stage directed by the *stage* information in the state entry. The execution stage retrieves the lookup key and the key is compared against the $ptr \rightarrow key$ to determine the outcome of the stage. If the lookup is not completed (step 2a in Figure 4), a new memory prefetch is issued to the next data structure node and the state is updated with the address of the node that will be visited in the next stage. If the lookup is completed (step 2b in Figure 4), a new lookup is initiated by incrementing the index of the input array and the state is saved into the circular buffer entry (of the completed lookup) after executing the necessary code to initiate a lookup. Then, the next buffer entry is read unless all the entries in the input array are consumed.

The pseudo-code of AMAC , shown in Listing 1, depicts a more realistic implementation of the scheme described above. The differences between our example in Figure 4 and Listing 1 are minor. One difference is that the modulo operation to access the circular buffer is costly when the number of in-flight accesses is not a power of two, as a division instruction is required. Therefore, we implement a rolling counter that is reset to zero when it reaches the size of the buffer, which allows us to pick arbitrary number of in-flight operations. We also use additional state entries, wherever necessary, for boundary checking, code simplification, and in certain cases for avoiding re-execution of the same functionality.

Code Stages. The simplified code stages for hash join probe, hash join build, group-by, binary search tree (BST) search, and skip list insert are depicted in Table 1. To identify the code stages, we analyze the baseline implementations and create the stages based on the pointer accesses. The entries in the table define the state transitions throughout the execution of the algorithm. The *NS (Next Stage)* field indicates which of the stages should be executed next based on the outcome(s) of the present stage. For example, the hash join probe stages depicted in Table 1 show that *stage 0* initiates a new lookup, while the access to the data structure and key comparison happens in the next stage, which is *stage 1*. For simplicity, we depict the hash join probe stages for unique keys, therefore upon a match in *stage 1*, a new lookup is initiated by transitioning to *stage 0*. Other algorithms have similar stages and actions; one difference is the *latch?* action, which returns a *true* value when the latch is acquired by another lookup.

While accurate, the states we show in the table are simplified. For instance, two optimizations, not captured in the table, are the following: (1) in order not to lose an opportunity to initiate a new memory access, we merge the terminating stages of each lookup with the initial stage (for the next lookup) wherever it is applicable. Therefore, when one lookup completes, a new lookup starts immediately (similar to our example in Figure 4), thus guaranteeing a constant number of memory accesses in flight at all times, (2) in the data structures with latches, we might employ extra intermediate stages to avoid deadlocks during the lookups. An example of such an implementation is the *stage 1* of group-by, which is implemented as two different stages depending on whether the latch was already acquired for a given lookup or not (not shown in Table 1).

Output order. Even though the lookup sequence does not follow the sequential order of row ids, the original order

Table 1: AMAC simplified codes stages (S: Stage, NS: Next stage).

S	Hash Join Probe	NS	Hash Join Build	NS	Group-by	NS	BST Search	NS	Skip List Insert	NS
0	Get new tuple Compute bucket address	1	Get new tuple Compute bucket address	1	Get new tuple Compute bucket address	1	Get new tuple Access root node	1	Get new tuple Access highest head node's successor	1
1	Compare (==) keys? T: Output result	0	Latch? T: Retry	1	Latch? T: Retry	1	Compare (==) keys? T: Output result	0	Compare (<) keys? T: Move to next	1
	F: Next node? T: Move to next	1	F: Node empty? T: Insert tuple	0	F: Compare (==) keys? T: Update node	3	F: Move to next	1	F: Compare (==) keys? T: Key exists	0
	F: No match	0	F: Next node? T: Move to next	2	F: Next node? T: Move to next	1			F: Lowest lvl? T: Insert key	2
			F: Get new node	3	F: Get new node	2			F: Collect pred. node Move one lvl down	1
2			Node empty? T: Insert tuple	0	Insert tuple	3			Generate rand. lvl Get new node	3
			F: Get new node	3						
3			Insert tuple	0	Update the aggr. field	0			Initialize new node Splice w/ collected nodes	0

is preserved through the $rid(idx)$ field of the state. This ensures that results are materialized in the input order.

3.2 Handling Read/Write Dependencies

The baseline implementation of the hash join build and group-by contains a latch per node for updating the contents of the data structure. When a latch cannot be acquired (i.e., the latch is acquired by another thread), the thread spins on the latch until it becomes available. Obviously, in AMAC, if one lookup cannot acquire the latch, there are still in-flight lookups pending. When an AMAC thread executes a code stage with latch acquire (e.g., *stage 1* of group-by and hash join build), we try to acquire the latch but if the attempt fails, we move on to the next lookup in the circular buffer and retry when the same lookup is performed later. As a result, we still spin on the latch but at a coarser granularity. In the cases where there is a probability of acquiring a latch but failing to complete the stage (i.e., group-by $stage1$), we employ an extra intermediate stage to avoid any deadlocks (not shown in Table 1) as described in the previous subsection.

In summary, if one wants to run AMAC in a multi-threaded fashion, the latch acquire should be implemented with an atomic swap instruction only and if the attempt fails, the thread moves on to the next lookup (i.e., no spinning on a single lookup). For single-threaded runs, the same ideas apply but there is no need for an atomic instruction to acquire the latch.

4. METHODOLOGY

Workloads. For all the workloads evaluated in this work, we use 16-byte tuples containing an 8-byte integer key and an 8-byte integer payload, representative of an in-memory columnar database storage representation. In all the cases, the data structure nodes are aligned to 64-byte cache block boundary with the *aligned* attribute.

For the hash join workload, we adopt the highly optimized chained hash table implementation of Balkesen et al. [4, 5], including the execution profiling functionality based on hardware performance counters. Each hash table bucket contains a 1-byte latch for synchronization, two 16-byte tuples and an 8-byte pointer to the next hash table node to be used in the case of collisions. For the uniform hash join workload, we again use the no-partitioning hash join workload of Balkesen et al. [4, 5] with uniformly distributed random R and S relation keys following a foreign key relationship. In the uniform workload, the key value ranges are dense and when the sizes of R and S are equal both relations contain unique values given the foreign key relationship. In the case where the relation sizes are not equal, the S relation key range is restricted to the keys in the R relation. As our study stresses the robustness of algorithms, we also relax the foreign key relationship and evaluate the case where R and S keys follow various Zipfian distributions, similar to prior studies [3, 17].

For the group-by workload, we extend the hash table used in hash join with an additional aggregation field. The input relation contains uniformly distributed random keys, where each key appears three times. We also evaluate the Zipf-skewed key distributions of 0.5 and 1 [27]. The values (payloads) in the input relations are uniformly distributed random unique values. We aggregate the values with six aggregation functions (*avg, count, min, max, sum and sum squared*), which are applied upon a match in the hash table.

We use a canonical implementation of a binary search tree. We build the tree 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. The probe relation size is always equal to the number of tree nodes, each of which finds a single match in the tree, resembling a join scenario with using an index.

For the skip list workload, we adopt the concurrent *pugh* skip list implementation from ASYCLIB [11]. We create an insert workload by building a skip list from scratch. For the search workload we perform lookups to the built skip list and each lookup finds exactly one match in the skip list. Both the build and probe relation contain uniformly distributed random unique keys and payloads. Because the skip list elements occupy larger memory space than the other evaluated data structures, we limit the number of input relation keys to 2^{25} (as opposed to 2^{27}) to avoid the memory footprint of the workload exceeding the physical memory capacity of our server.

Experimental Setup. The server machines used in our experiments are listed in Table 2. The Intel Xeon x5670 server features a two-socket CPU with 6 cores per socket. We use just one socket in our experiments except for the bottleneck analysis, which uses both. The server runs Red Hat Linux (kernel version 2.6.32). On x86, we compile our

Table 2: Architectural parameters.

Processor	Xeon x5670	SPARC T4
Technology	32nm @ 2.93GHz	40nm @ 3GHz
ISA	x86	SPARC v9
CMP Cores/Threads	6/12	8/64
Core Types	4-wide OoO	2-wide OoO
L1 I/D Cache (per core)	32KB	16KB
L2 Cache (per core)	256KB	128KB
L3 Cache	12MB	4MB
TLB entries (L1/L2)	64/512	128/-
Main Memory	24GB, DDR3	1TB, DDR3

code with `gcc 4.7.2` using the `-O3` flag. On Oracle SPARC T4, we use a single 8-core CMP. The server runs Sun OS 5.11 and we compile our code with the `C compiler 5.13` found in `Oracle Solaris Studio 12.4`. In addition to the Oracle Solaris Studio compiler flags recommended by prior work [5], we use the `-xprefetch=no%auto` option to disable the automatic generation of prefetch instructions by the compiler, which leads to overall performance improvements for all of the evaluated techniques including the baseline code.

In all measurements, we use large VM pages, 2 MB on x86 and 4 MB on SPARC. For all the evaluated techniques, we parameterize the code and perform a sensitivity analysis to pick the best performing parameters for each experiment. For prefetching data blocks, on x86 we use the `PREFETCHNTA` instruction via the built-in gcc functions. On SPARC, we use the strong prefetch variant [24]. In both platforms, prefetch instructions complete as long as a TLB miss does not cause a fault, which is rare with in-memory execution.

5. EVALUATION

5.1 Hash Join

In order to analyze the performance of various techniques on hash join with various dataset sizes, we keep the size of the probe relation constant at 2GB ($|S| = 2^{27}$). We evaluate two different build relation sizes, 2MB ($|R| = 2^{17}$), referred to as *small*, and 2GB ($|R| = 2^{27}$), referred to as *large*. In addition, we evaluate skewed datasets, where the keys of R and S follow a Zipfian data distribution. The Zipf factor of each relation is denoted by $[Z_R, Z_S]$. We pick the best tuning parameter for all the techniques for each experiment.

Figure 5 depicts the cycles per output tuple for build and probe in hash join. We observe that for the join of the differently sized columns ($2MB \bowtie 2GB$), shown in Figure 5a, the build time is negligible and all the cycles are spent on probing the hash table, which fits in the last-level cache (LLC) of the Xeon processor. For the same reason, the skew in the build and probe relation keys does not have a significant impact on the execution cycles. In contrast, in the hash join with equally sized relations ($2GB \bowtie 2GB$), shown in Figure 5b, the build cycles constitute half of the join cycles as the size of the build relation is beyond the LLC capacity. At the same time, increasing the Zipf parameter in the relation R (from $Z_R = .5$ to $Z_R = 1$) increases the number of probe cycles for GP and SPP by 1.8x and 2.4x, respectively. In contrast, the probe cycles for AMAC only increases by 5% on average underscoring the robustness of AMAC under irregular data structure accesses. The build phase overall is not sensitive to skew because the link list insertions are uniform operations regardless of the data distribution.

Table 3: Execution profile of uniform join with unequal table sizes ($2MB \bowtie 2GB$) on Xeon x5670.

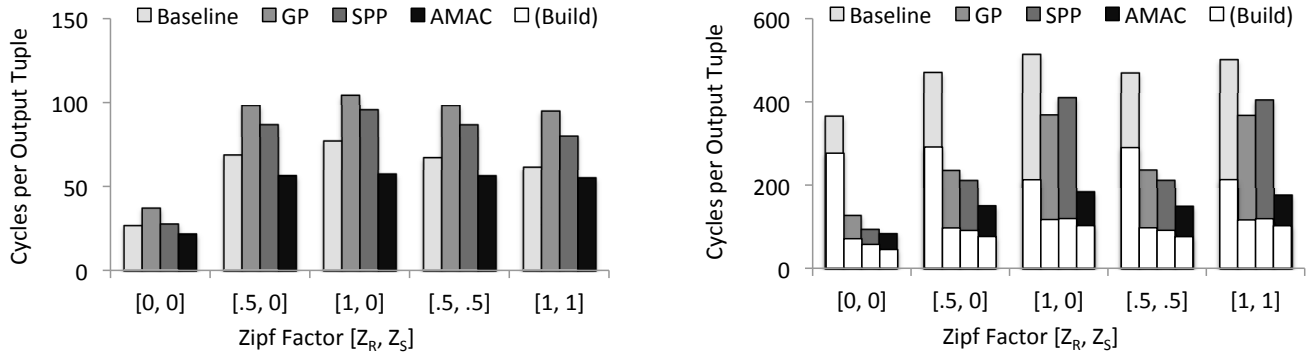
	Baseline	GP	SPP	AMAC
Instructions per Tuple	36	90	67	55
Cycles per Tuple	27	37	28	22

Figure 5a shows the performance of the small relation join ($2MB \bowtie 2GB$). We observe that the baseline is faster than both GP and SPP by 32% on average, while AMAC outperforms the baseline by 21%. The hash table built with the small relation fits in the LLC of Xeon, therefore the core partially hides the LLC latency in the baseline case. To investigate the cause of the GP and SPP slowdown, we perform a profiling analysis. Table 3 shows the number of instructions executed per tuple and the performance obtained for the uniform join ($[0, 0]$). We find that GP and SPP have a 2.5x and 1.9x overhead in the number of instructions per tuple over the baseline code, therefore offset the prefetching benefits. In contrast, AMAC has only a 1.5x instruction overhead, which explains the relatively better performance.

Figure 5b shows that, with equally sized relations ($2GB \bowtie 2GB$), all three techniques (GP, SPP, AMAC) achieve significant speedups (2.8x, 3.8x, 4.3x, respectively) under uniform input ($[0, 0]$) as they all effectively hide the memory latency. It is important to note that the 27% performance gap between AMAC and SPP in the previous unequally sized join is bridged as both techniques simply hit the limit of the MLP provided by the hardware. However, for the skewed R ($[.5, 0]$, $[1, 0]$), GP and SPP lose their effectiveness at generating memory-level parallelism due to the irregular traversal paths in the hash table and deliver average speedups ranging from 1.4x to 2x and 1.2x to 2.2x, respectively. As expected, AMAC gracefully handles the divergence in the hash table walk and achieves a robust performance of 3x on average for all the skewed cases. Moreover, adding skew to the relation S ($[.5, .5]$, $[1, 1]$), which can help with the locality of the hash table buckets, has a minor impact on the performance as the workloads' working set is too large to be captured in the LLC even when both relations are skewed.

Figure 6 depicts the execution cycle sensitivity to the tuning parameters of GP, SPP, and AMAC. For all the techniques, we vary the parameters that increase the number of parallel lookups performed within a thread. For the uniform probes ($[0, 0]$), we observe that increasing the number of parallel lookups lowers the cycles-per-tuple due to the increase in memory-level parallelism and, in general, ten in-flight lookups deliver the best performance except for GP. Further increasing the number of in-flight requests does not improve the performance of SPP and AMAC as the limit of L1-D outstanding misses (i.e., 10 L1-D MSHRs) is reached on the Xeon core [14]. For GP, (Figure 6a), the best performance is achieved with a group size of 15 due to the fact that GP is limited by the instruction-count overhead, instead of the number of MSHRs, and larger group sizes yield fewer outer-loop iterations slightly improving core's performance.

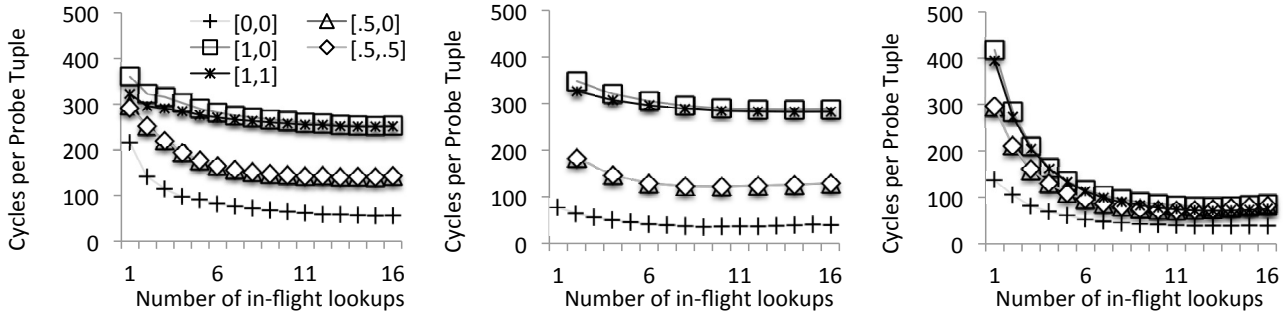
For the skewed data distributions, we observe that GP (Figure 6a) and SPP (Figure 6b) have limited benefits from multiple parallel lookups as the skewed key distribution leads to buckets with long pointer chains, which cannot be handled by GP and SPP. Especially in the cases where



(a) Small build relation ($2MB \bowtie 2GB$)

(b) Large build relation ($2GB \bowtie 2GB$)

Figure 5: Hash join cycles breakdown under different data distributions on Xeon x5670.



(a) GP

(b) SPP

(c) AMAC

Figure 6: Probe performance sensitivity to GP, SPP, AMAC tuning parameters on Xeon x5670 ($2GB \bowtie 2GB$).

$Z_R = 1$, the performance difference between a single in-flight lookup and the best case is only 16%. In contrast, AMAC (Figure 6c) is robust to various data distributions and handles the non-uniform cases without incurring any additional performance overhead.

Scalability analysis. We next study the scalability of AMAC and focus on the read-only hash join probe phase to mitigate any algorithm-related scalability issues. We report the probe throughput, which is calculated as $|S|/probeExecutionTime$ on the Xeon and T4 machines. On both platforms, we perform the experiment by assigning software threads first to physical cores (six on Xeon and eight on T4) and we start using SMT threads upon running out of physical cores.

The results on Xeon with uniform data, shown in Figure 7a, indicate that GP, SPP, and AMAC throughputs scale well up to four threads. However, the throughputs start leveling off after four cores and increasing the number of contexts does not result in any significant improvement. Meanwhile, the baseline algorithm achieves better scalability and brings the initial 2.5x throughput gap down to 80% by taking advantage of all the hardware contexts on Xeon. In contrast, the same experiment on T4, shown in Figure 8a, shows that GP, SPP, and AMAC scale well with the available physical cores (eight) and even benefit from SMT threads moderately.

5.1.1 Bottleneck Analysis

The results presented in Figure 8 clearly indicate that the AMAC approach does not affect the inherently scalable nature of the hash table probe algorithm. While the T4 machine offers almost linear scalability with the number of physical cores (i.e., eight), increasing the number of the SMT threads has diminishing returns as the threads start competing for physical core cycles. However, the results on Xeon (Figure 7) signal a more significant problem as the algorithms do not even scale with the number of physical cores (i.e., six cores), indicating that there is a hardware bottleneck on Xeon. Moreover, our results corroborate recent work by Balkesen et al. [5], which also reports relatively low benefits for prefetch-based hash joins on a fully loaded Xeon (Nehalem) core.

We further investigate the source of the bottleneck on Xeon by using hardware performance counters. Table 4 depicts the performance counter measurements while increasing the number of threads for the probe phase of the large join. We measure the *instructions per cycle (IPC)* and *L1-D MSHR hits* per kilo-instruction, which are the memory references that miss in the L1-D but hit in the L1-D MSHRs, meaning that the memory access (e.g., prefetch) was already issued by the core but the data has not arrived yet (i.e., outstanding miss). We observe that the average IPC of six threads is 2x worse than the single-threaded execution, which verifies the drop in the speedups explained above. Similarly, L1-D MSHR hits show an almost 4x increase in the six-thread experiment vs. single-thread experiment as the prefetches do not arrive in a timely manner in the six-thread experiment.

While the cause of this problem can be off-chip accesses, our additional performance counter measurements indicate that increasing the thread count has a marginal impact on the number of off-chip accesses. Therefore, our last hypoth-

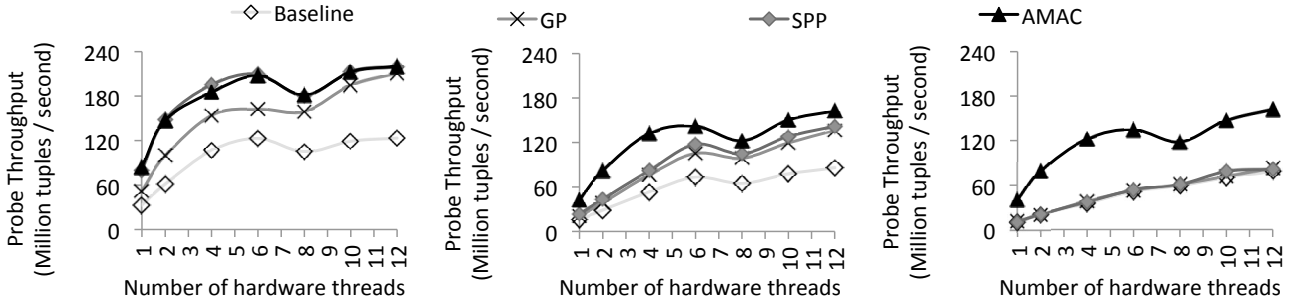


Figure 7: Hash table probe scalability with uniform and Zipf-skewed keys ($2GB \bowtie 2GB$) on Xeon x5670.

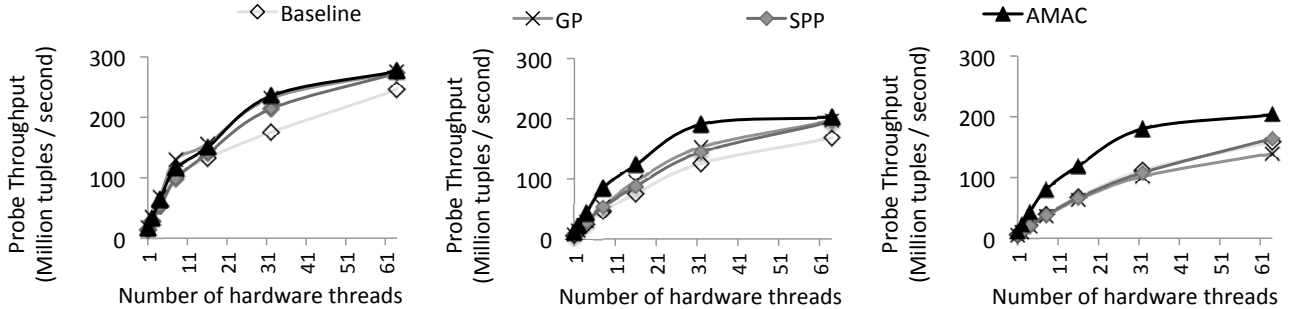


Figure 8: Hash join probe scalability with uniform and Zipf-skewed keys ($2GB \bowtie 2GB$) on SPARC T4.

Table 4: Hash join probe scalability profiling on Xeon x5670.

Threads	1	2	4	6	2+2
IPC	1.4	1.4	1.0	0.7	1.3
L1-D MSHR Hits (per k-inst.)	1.8	2.5	5.5	6.9	3.7

esis is that there is a resource contention in the LLC. This assumption is sensible given that the number of LLC MSHRs provisioned for off-chip load requests (referred to as Global Queue) in the Xeon (Nehalem) processor is limited to 32 entries (for loads) [22], while the aggregate number of outstanding load misses generated by six cores can reach up to sixty (i.e., 10 L1-D MSHR entries per core).

To verify our hypothesis, we re-run the experiments with four threads, but this time we distribute the four threads to two sockets (i.e., two physical CPUs each with an LLC) with two threads per socket (Table 4, "2+2" column). Our results for four threads on two sockets indicate 50% lower L1-D MSHR hits compared to four threads on a single socket. Furthermore, the IPC and the L1-D MSHR hit behavior of the two-socket experiment is almost identical to two threads on a single socket showing that the contention in the LLC is resolved.

As a result, we conclude that utilizing the upper levels of the hierarchy causes severe contention in the LLC when all threads issue random off-chip memory accesses. Therefore, the throughput on Xeon saturates with four threads, explaining why prefetch-based techniques do not deliver the expected performance on fully loaded multi-core Xeon (Nehalem) processors.

5.2 Group-by

Figure 9 shows the group-by performance for two input relation sizes. We observe that for the skewed small inputs (2^{17}), the performance of GP and SPP is similar to or worse than the baseline, while AMAC provides 1.6x speedup on average. Under heavier data skew (z=1), read/write dependencies within the SPP pipeline force frequent serialization of the conflicting lookups and cause a severe performance degradation. Although GP suffers from the same problem, the group boundaries allow for relatively cheaper cleanup passes. In contrast, AMAC's performance is robust, as the conflicts do not require any additional serialization code.

For the big relation, the average speedups of GP, SPP, and AMAC are 2.1x, 2.2x, 2.6x, respectively. In this case, the performance impact of the extra work due to conflicts is relatively minor as the execution hits the memory-level parallelism limit of the underlying hardware.

5.3 Tree Search

Figure 10 depicts our results for binary search tree (BST) search. In general, the benefit of all prefetching techniques compared to the baseline increases with the height of the tree, as the baseline code fails to expose memory-level parallelism on long pointer chains. We observe that AMAC achieves a maximum speedup of 4.45x (2.8x geomean) over the baseline, compared to speedups of 3.4x (2.1x geomean) and 2.7x (1.8x geomean) for GP and SPP, respectively.

We also note that in contrast to the group-by case, where SPP and GPP perform almost identical, in BST search GP performs 20% better than SPP. The reason for the poor performance of SPP in tree search is the loss of memory-level parallelism in case of bailouts, which occur for the longest traversals. While SPP's pipeline can be stretched to match the height of the tree, we found that the average memory-

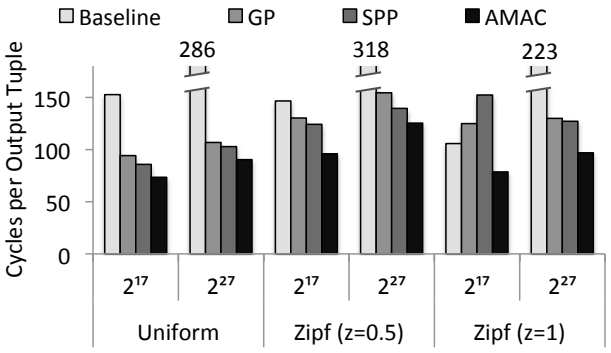


Figure 9: Group-by on Xeon x5670.

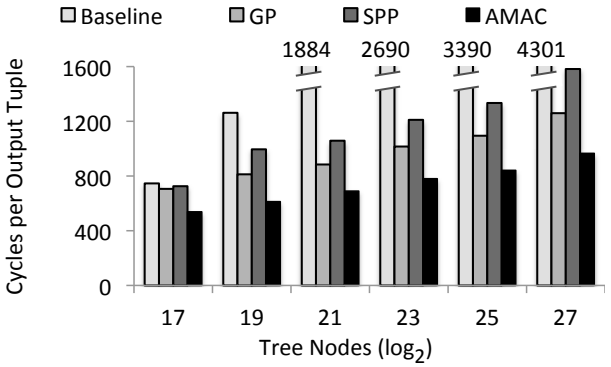


Figure 10: BST search on Xeon x5670.

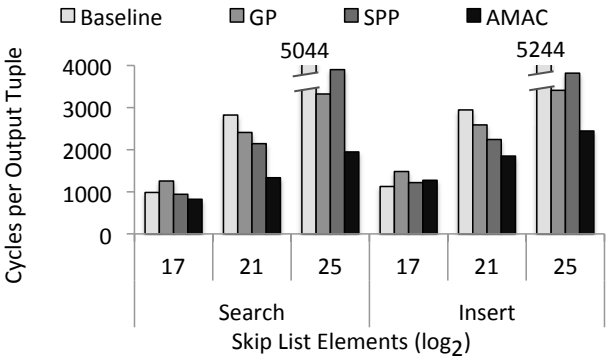


Figure 11: Skip list on Xeon x5670.

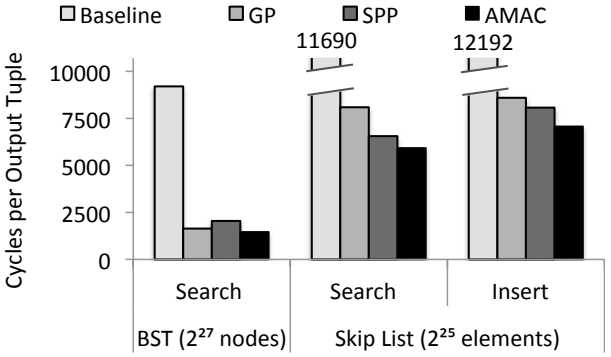


Figure 13: BST and skip list on SPARC T4.

level parallelism (and performance) attained in this configuration can be inferior to that with a slightly shorter pipeline that favors the common-case traversal length but incurs an occasional bailout.

5.4 Skip List Search and Insert

Although the algorithmic complexity of skip list search and insert is the same as the BST operations, skip list operations contain the most complex functionality we evaluate in this work. Especially the insert operation, which consists of a skip list search followed by the splice operation, iteratively constructs a vector that points to the correct insertion locations for each insertion. This vector occupies 0.5KB per lookup and is maintained in AMAC's circular buffer for each in-flight lookup.

The splice code also includes several function calls. Some function calls, such as allocating a new node and determining random numbers, possibly result in additional function calls during the insertion operation. In addition, the splice code contains loops that acquire and release the latches in appropriate skip list nodes via function calls. Overall, these operations result in CPU-intensive execution phases, while the rest of the execution is memory-intensive.

Figure 11 shows our results for skip list search and insert operations. During a search, the traversal at each skip list level terminates after an arbitrary number of node traversals, and this irregularity hurts the performance of GP and SPP. As a result, GP, SPP, and AMAC achieve average speedups of 1.15x (1.5x max), 1.2x (1.3x max), and 1.9x (2.6x max), respectively.

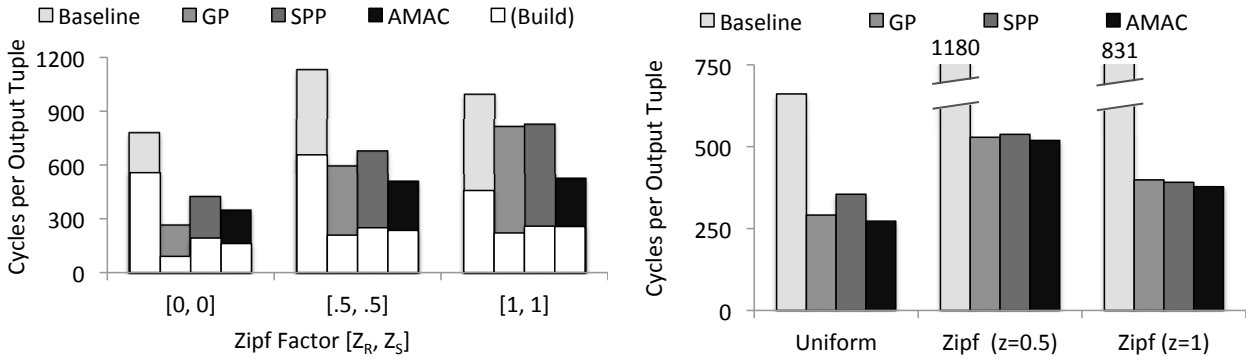
For the insert operations, GP, SPP, and AMAC deliver average speedups of 1.1x (1.5x max), 1.2x (1.3x max), and 1.4x (2.1x max), respectively. The reason for the more modest performance improvements is due to the additional and complex operations during inserts. As explained above, while these operations take extra CPU cycles, they are not memory-bound, and hence are not targeted by the prefetching instructions.

5.5 SPARC T4 Experiments

We run our workloads on SPARC T4 by using a single hardware context (the effect of SMT threads are discussed in Section 5.1). We also note that, unlike the Xeon processor, the T4 processor discards the demanded prefetch requests that hit in the on-chip caches, therefore we only perform experiments with large relation and do not study the behavior of the small relation used in the Xeon experiments.

Figure 12 depicts our results on the T4 machine for hash join and group-by workloads. For the hash join workload, GP, SPP, and AMAC improve the performance by an average of 1.9x, 1.5x, and 2.1x, respectively. Interestingly, in the uniform case, the GP build phase performs exceptionally well by outperforming both SPP and AMAC. However, in the rest of the experiments, GP and SPP deliver inconsistent performance with respect to each other, while AMAC, with a single exception we pointed out, delivers the highest performance. The trend for the group-by workload (shown in Figure 12b) is also similar. GP, SPP, and AMAC deliver average speedups of 2.2x, 2x, and 2.3x, respectively.

Figure 13 plots the tree search and skip list results on T4. Given the simple nature of the tree search, GP, SPP, and AMAC achieve impressive speedups of 5.6x, 4.5x, and 6.2x over the baseline, respectively. For the skip list operations, GP, SPP, and AMAC deliver average speedups of 1.4x, 1.6x,



(a) Hash join ($2GB \bowtie 2GB$) (b) Group-by

Figure 12: Performance of hash join and group-by with 2^{27} input relation keys on SPARC T4.

and 1.8x. Overall, these results follow a similar trend, with AMAC delivering the best and the most consistent performance, while the benefits of GP and SPP with respect to each other fluctuate.

The overall gains over a single T4 thread is lower compared to Xeon due to the relatively weak micro-architecture of a single T4 core. We conclude that regardless of the micro-architecture, AMAC delivers a robust performance in the face of irregularities across lookups, while the performance of other prefetching techniques is inconsistent.

6. DISCUSSION

Setting the number of in-flight lookups. The goal of AMAC is to extract peak memory-level parallelism available in a processor core. As seen in Figure 6c, AMAC's performance shows little sensitivity to the exact number of memory lookups beyond eight or so. We observed a very similar trend for SMT execution and did not perform any special tuning for it. Nevertheless, our experience shows that picking an extremely large number of in-flight memory accesses initiated by AMAC (e.g., greater than 32 on the Xeon) can be harmful due to TLB thrashing on large datasets with low TLB locality.

AMAC automation. In this work, we manually create AMAC stages, implement state save/restore functionality, and tune performance. Ideally this process should be automated and hidden from the software developer. We believe that portable performance tuning techniques [26] and event-driven programming language concepts such as *coroutines* that allow for cooperative multitasking within a thread (e.g., escape-and-reenter loops) can help creating a generalized software model and framework for AMAC-style execution. The benefits of such framework include minimal modifications to baseline code, easier programmability, and portability across platforms. The disadvantages can be the user-land threads' state maintenance and space overhead, which is an overkill as the thread state carries a lot of redundancy across the threads of the same data structure lookup.

7. RELATED WORK

Software prefetching instructions are effective in hiding the memory access latency when the required cache blocks can be demanded early enough. Unfortunately, prefetching within the traversal of single pointer chain is not possible due to dependent address calculations. To solve this problem, prior work has proposed data-linearization prefetching [19] to calculate the addresses without needing pointers so that the prefetches can be issued ahead of time. Similarly, history-based prefetching techniques [9] maintain an array of jump pointers containing the pointers from recent traversals. However, these techniques either assume that the data structure is traversed in a similar order more than once or incur both space and time overhead to increase the prefetch accuracy.

To take advantage of the extra hardware contexts on the processor cores, Zhou et al. [28] divide a single software thread into producer/consumer threads, which communicate through shared memory via software queues. Ideally, the two contexts can work cooperatively to overlap long-latency memory misses with useful work. However, the synchronization overhead in the software queue negates the benefits of overlapping, as the producer and consumer threads have read/write dependencies on the queue. To mitigate such synchronization overheads, speculative decoupled software pipelining [25] splits recursive data structure traversals into two hardware contexts and performs synchronization via specialized hardware arrays with support for resolving dependencies. As of now, such hardware support is not found in off-the-shelf processors. However, given the server efficiency constraints and reviving interest in specialized database hardware [12, 18], these techniques are insightful.

Hardware conscious algorithms and data structures have gained importance in the last decade [7, 15, 20, 21, 27, 29]. Tuning the data structure to the underlying hardware is an effective approach for reducing the number of cache and TLB misses. In addition, minimizing the number of individual memory accesses is possible by leveraging the SIMD instructions [16]. Our work is orthogonal and can be used in conjunction with these techniques. Moreover, our approach is a step towards robust performance and tuning [13] for hardware conscious algorithms.

8. CONCLUSION

This work introduced Asynchronous Memory Access Chaining, a new approach for achieving high MLP on pointer-intensive operations with irregular behavior across lookups. AMAC is effective in managing irregularity by maintaining the state of each lookup separately from other

in-flight lookups. This separation allows AMAC to react to the needs of each individual lookup, such as executing more or fewer memory accesses than the common case, without affecting the execution of other lookups.

While our evaluation focused on pointer-intensive operations in the context of relational databases, we believe that AMAC's applicability goes beyond that. Our future work will examine the efficacy of AMAC on graph workloads and operations over unstructured data. Our results also identified important bottlenecks in modern server processors that limit the amount of MLP that software can leverage. Given the growing importance of analytics in today's business and society, and the futility of caching the ever-growing datasets on-chip, it is imperative that processor designs evolve not to limit software's ability to exploit MLP.

9. ACKNOWLEDGMENTS

We would like to thank Ed Bugnion, Partha Ranganathan, Karu Sankaralingam, Alexandros Daglis, Djordje Jevdjic, Cansu Kaynak, Nooshin Mirzadeh, Javier Picorel, Danica Porobic, Georgios Psaropoulos, Pınar Tözün, Stavros Volos, and the anonymous reviewers for their insightful feedback on the paper. We also thank Tim Harris for the assistance with the SPARC T4 server access. This work is supported in part by the *Google Europe Doctoral Fellowship* and the *Workloads and Server Architectures for Green Datacenters* project of the Swiss National Science Foundation.

10. REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB Endowment*, 5(10), 2012.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering*, 2013.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on modern processor architectures. In *IEEE Transactions on Knowledge and Data Engineering*, volume PP, 2014.

[6] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. 8(4), 2014.

[7] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems*, 32(3), 2007.

[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001.

[10] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[11] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[12] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware*, 2005.

[13] G. Graefe. Robust query processing. In *Proceedings of the 27th International Conference on Data Engineering*, 2011.

[14] Intel Xeon Processor 5600 Series Datasheet, Vol 2. `http://intel.ly/1Am4SVw`.

[15] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. 8(6), 2015.

[16] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[17] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. In *Proceedings of the 35th International Conference on Very Large Data Bases*, 2009.

[18] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[19] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4), 2002.

[21] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3), Dec. 2000.

[22] Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. `http://intel.ly/1QFmyky`.

[23] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[24] SPARC Strong Prefetch. `http://bit.ly/1OceJlN`.

[25] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[26] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[27] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*, 2011.

[28] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[29] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.