Meet the Walkers
Accelerating Index Traversals for In-Memory Databases

Onur Kocberber[1] Boris Grot[2] Javier Picorel[1]
Babak Falsafi[1] Kevin Lim[3] Parthasarathy Ranganathan[4]

[1]EcoCloud, EPFL [2]University of Edinburgh [3]HP Labs [4]Google, Inc.

ABSTRACT

The explosive growth in digital data and its growing role in real-time decision support motivate the design of high-performance database management systems (DBMSs). Meanwhile, slowdown in supply voltage scaling has stymied improvements in core performance and ushered an era of power-limited chips. These developments motivate the design of DBMS accelerators that (a) maximize utility by accelerating the dominant operations, and (b) provide flexibility in the choice of DBMS, data layout, and data types.

We study data analytics workloads on contemporary in-memory databases and find hash index lookups to be the largest single contributor to the overall execution time. The critical path in hash index lookups consists of ALU-intensive key hashing followed by pointer chasing through a node list. Based on these observations, we introduce Widx, an on-chip accelerator for database hash index lookups, which achieves both high performance and flexibility by (1) decoupling key hashing from the list traversal, and (2) processing multiple keys in parallel on a set of programmable *walker* units. Widx reduces design cost and complexity through its tight integration with a conventional core, thus eliminating the need for a dedicated TLB and cache. An evaluation of Widx on a set of modern data analytics workloads (TPC-H, TPC-DS) using full-system simulation shows an average speedup of 3.1x over an aggressive OoO core on bulk hash table operations, while reducing the OoO core energy by 83%.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]:
Heterogeneous (hybrid) systems

General Terms

Design, Experimentation, Performance

Keywords

Energy efficiency, hardware accelerators, database indexing

[2]This work was done while the author was at EPFL.
[4]This work was done while the author was at HP Labs.

MICRO-46 December 07 - 11, 2013, Davis, CA, USA
Copyright 2013 ACM 978-1-4503-2638-4/13/12 ...$15.00.
http://dx.doi.org/10.1145/2540708.2540748

1. INTRODUCTION

The information revolution of the last decades is being fueled by the explosive growth in digital data. Enterprise server systems reportedly operated on over 9 zettabytes (1 zettabyte = 10^{21} bytes) of data in 2008 [29], with data volumes doubling every 12 to 18 months. As businesses such as Amazon and Wal-Mart use the data to drive business processing and decision support logic via databases with several petabytes of data, IDC estimates that more than 40% of global server revenue ($22 billion out of $57 billion) goes to supporting database workloads [10].

The rapid growth in data volumes necessitates a corresponding increase in compute resources to extract and serve the information from the raw data. Meanwhile, technology trends show a major slowdown in supply voltage scaling, which has historically been the primary mechanism for lowering the energy per transistor switching event. Constrained by energy at the chip level, architects have found it difficult to leverage the growing on-chip transistor budgets to improve the performance of conventional processor architectures. As a result, an increasing number of proposals are calling for specialized on-chip hardware to increase performance and energy efficiency in the face of dark silicon [9, 15]. Two critical challenges in the design of such dark silicon accelerators are: (1) identifying the codes that would benefit the most from acceleration by delivering significant value for a large number of users (i.e., maximizing utility), and (2) moving just the right functionality into hardware to provide significant performance and/or energy efficiency gain without limiting applicability (i.e., avoiding over-specialization).

This work proposes *Widx*, an on-chip accelerator for database hash index lookups. Hash indexes are fundamental to modern database management systems (DBMSs) and are widely used to convert linear-time search operations into near-constant-time ones. In practice; however, the sequential nature of an individual hash index lookup, composed of key hashing followed by pointer chasing through a list of nodes, results in significant time constants even in highly tuned in-memory DBMSs. Consequently, a recent study of data analytics on a state-of-the-art commercial DBMS found that 41% of the total execution time for a set of TPC-H queries goes to hash index lookups used in hash-join operations [16].

By accelerating hash index lookups, a functionality that is essential in modern DBMSs, Widx ensures high utility. Widx maximizes applicability by supporting a variety of schemas (i.e., data layouts) through limited programmability. Finally, Widx improves performance and offers high energy efficiency through simple parallel hardware.

Our contributions are as follows:

- We study modern in-memory databases and show that hash index (i.e., hash table) accesses are the most significant single source of runtime overhead, constituting 14-94% of total query execution time. Nearly all of indexing time is spent on two basic operations: (1) hashing keys to compute indices into the hash table (30% on average, 68% max), and (2) walking the in-memory hash table's node lists (70% on average, 97% max).
- Node list traversals are fundamentally a sequential pointer-chasing functionality characterized by long-latency memory operations and minimal computational effort. However, as indexing involves scanning a large number of keys, there is abundant inter-key parallelism to be exploited by walking multiple node lists concurrently. Using a simple analytical model, we show that in practical settings, inter-key parallelism is constrained by either L1 MSHRs or off-chip bandwidth (depending on the hash index size), limiting the number of concurrent node list traversals to around four per core.
- Finding the right node lists to walk requires hashing the input keys first. Key hashing exhibits high L1 locality as multiple keys fit in a single cache line. However, the use of complex hash functions requires many ALU cycles, which delay the start of the memory-intensive node list traversal. We find that decoupling key hashing from list traversal takes the hashing operation off the critical path, which reduces the time per list traversal by 29% on average. Moreover, by exploiting high L1-D locality in the hashing code, a single key generator can feed multiple concurrent node list walks.
- We introduce Widx, a programmable widget for accelerating hash index accesses. Widx features multiple *walkers* for traversing the node lists and a single *dispatcher* that maintains a list of hashed keys for the walkers. Both the walkers and a dispatcher share a common building block consisting of a custom 2-stage RISC core with a simple ISA. The limited programmability afforded by the simple core allows Widx to support a virtually limitless variety of schemas and hashing functions. Widx minimizes cost and complexity through its tight coupling with a conventional core, which eliminates the need for dedicated address translation and caching hardware.

Using full-system simulation and a suite of modern data analytics workloads, we show that Widx improves performance of indexing operations by an average of 3.1x over an OoO core, yielding a speedup of 50% at the application level. By synthesizing Widx in 40nm technology, we demonstrate that these performance gains come at negligible area costs (0.23mm^2), while delivering significant savings in energy (83% reduction) over an OoO core.

The rest of this paper is organized as follows. Section 2 motivates our focus on database indexing as a candidate for acceleration. Section 3 presents an analytical model for finding practical limits to acceleration in indexing operations. Section 4 describes the Widx architecture. Sections 5 and 6 present the evaluation methodology and results, respectively. Sections 7 and 8 discuss additional issues and prior work. Section 9 concludes the paper.

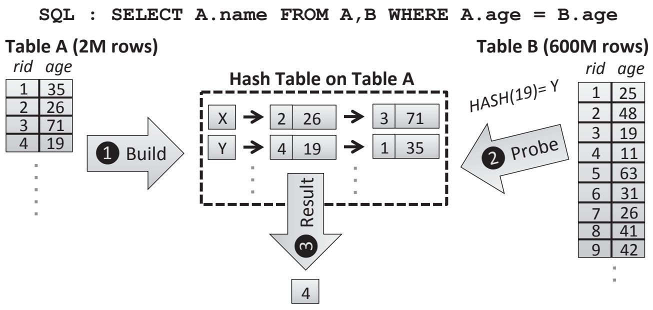


Figure 1: Table join via hash index.

2. MOTIVATION

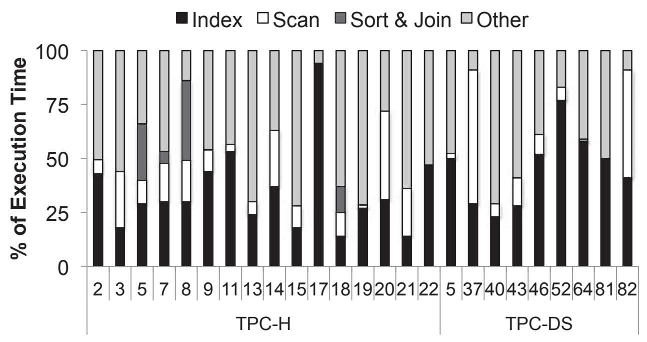
2.1 DBMS Basics

DBMSs play a critical role in providing structured semantics to access large amounts of stored data. Based on the relational model, data is organized in *tables*, where each table contains some number of *records*. Queries, written in a specialized query language (e.g., SQL) are submitted against the data and are converted to *physical operators* by the DBMS. The most fundamental physical operators are *scan, join* and *sort*. The scan operator reads through a table to select records that satisfy the selection condition. The join operator iterates over a pair of tables to produce a single table with the matching records that satisfy the join condition. The sort operator outputs a table sorted based on a set of attributes of the input table. As the tables grow, the lookup time for a single record increases linearly as the operator scans the entire table to find the required record.

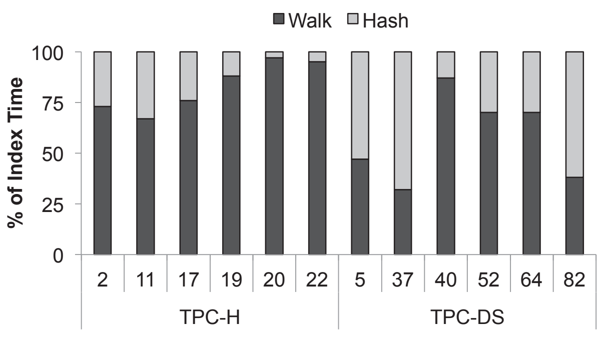
In order to accelerate accesses to the data, database management systems commonly employ an auxiliary *index* data structure with sub-linear access times. This auxiliary index can be built either by the database administrator or generated on the fly as a query plan optimization. One of the most common index data structures is hash table, preferred for its constant lookup time. Locating an item in a hash table first requires probing the table with a hashed key, followed by chasing node pointers looking for the node with the matching key value. To ensure low and predictable latency, DBMSs use a large number of buckets and rely on robust hashing functions to distribute the keys uniformly and reduce the number of nodes per bucket.

2.2 Database Indexing Example

Figure 1 shows a query resulting in a join of two tables, A and B, each containing several million rows in a column-store database. The tables must be joined to determine the tuples that match $A.age = B.age$. To find the matches by avoiding a sequential scan of all the tuples in Table A for each tuple in Table B, an index is created on the smaller table (i.e., Table A). This index places all the tuples of Table A into a hash table, hashed on $A.age$ (Step 1). The index executor is initialized with the location of the hash table, the key field being used for the probes, and the type of comparison (i.e., *is_equal*) being performed. The index executor then performs the query by using the tuples from Table B to *probe* the hash table to find the matching tuples in Table A (Step 2). The necessary fields from the matching tuples are then written to a separate output table (Step 3).



(a) Total execution time breakdown



(b) Index execution time breakdown

Figure 2: TPC-H & TPC-DS query execution time breakdown on MonetDB.

```
/* Constants used by the hashing function */
#define HPRIME 0xBIG
#define MASK 0xFFFF
/* Hashing function */
#define HASH(X) (((X) & MASK) ^ HPRIME)

/* Key iterator loop */
do_index(table_t *t, hashtable_t *ht) {
    for (uint i = 0; i < t->keys.size; i++)
        probe_hashtable(t->keys[i], ht);
}

/* Probe hash table with given key */
probe_hashtable(uint key, hashtable_t *ht) {
    uint idx = HASH(key);
    node_t *b = ht->buckets+idx;
    while(b) {
        if (key == b->key)
            { /* Emit b->id */ }
        b = b->next; /* next node */
    }
}
```

Listing 1: Indexing pseudo-code.

Listing 1 shows the pseudo-code for the core indexing functionality, corresponding to Step 2 in Figure 1. The *do_index* function takes as input table *t*, and for each key in the table, probes the hash table *ht*. The canonical *probe_hashtable* function hashes the input key and walks through the node list looking for a match.

In real database systems, the indexing code tends to differ from the abstraction in Listing 1 in a few important ways. First, the hashing function is typically more robust than what is shown above, employing a sequence of arithmetic operations with multiple constants to ensure a balanced key distribution. Second, each bucket has a special header node, which combines minimal status information (e.g., number of items per bucket) with the first node of the bucket, potentially eliminating a pointer dereference for the first node. Last, instead of storing the actual key, nodes can instead contain pointers to the original table entries, thus trading space (in case of large keys) for an extra memory access.

2.3 Profiling Analysis of a Modern DBMS

In order to understand the chief contributors to the execution time in database workloads, we study MonetDB [18], a popular in-memory DBMS designed to take advantage of modern processor and server architectures through the use of column-oriented storage and cache-optimized operators. We evaluate Decision Support System (DSS) workloads on a server-grade Xeon processor with TPC-H [31] and TPC-DS [26] benchmarks. Both DSS workloads were set up with a 100GB dataset. Experimental details are described in Section 5.

Figure 2a shows the total execution time for a set of TPC-H and TPC-DS queries. The TPC-H queries spend up to 94% (35% on average) and TPC-DS queries spend up to 77% (45% on average) of their execution time on indexing. Indexing is the single dominant functionality in these workloads, followed by scan and coupled sort&join operations. The rest of the query execution time is fragmented among a variety of tasks, including aggregation operators (e.g., sum, max), library code, and system calls.

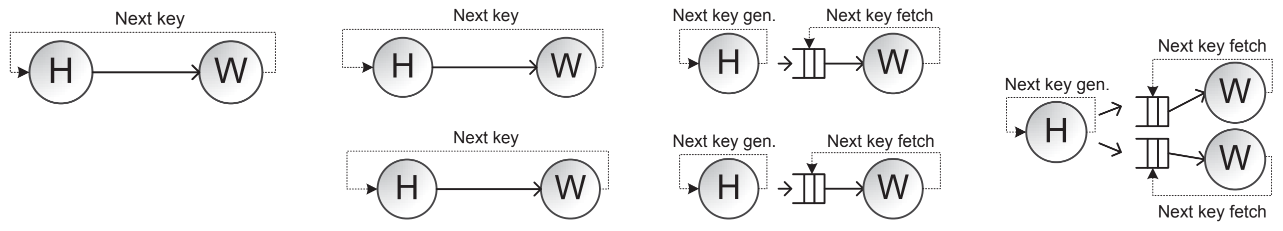
To gain insight into where the time goes in the indexing phase, we profile the index-dominant queries on a full-system cycle-accurate simulator (details in Section 5). We find that hash table lookups account for nearly all of the indexing time, corroborating earlier research [16]. Figure 2b shows the normalized hash table lookup time, broken down into its primitive operations: key hashing (Hash) and node list traversal (Walk). In general, node list traversals dominate the lookup time (70% on average, 97% max) due to their long-latency memory accesses. Key hashing contributes an average of 30% to the lookup latency; however, in cases when the index table exhibits high L1 locality (e.g., queries 5, 37, and 82), over 50% (68% max) of the lookup time is spent on key hashing.

Summary: Indexing is an essential database management system functionality that speeds up accesses to data through hash table lookups and is responsible for up to 94% of the query execution time. While the bulk of the indexing time is spent on memory-intensive node list traversals, key hashing contributes 30% on average, and up to 68%, to each indexing operation. Due to its significant contribution to the query execution time, indexing presents an attractive target for acceleration; however, maximizing the benefit of an indexing accelerator requires accommodating both key hashing and node list traversal functionalities.

3. DATABASE INDEXING ACCELERATION

3.1 Overview

Figure 3 highlights the key aspects of our approach to indexing acceleration. These can be summarized as (1) walk multiple hash buckets concurrently with dedicated *walker* units, (2) speed up bucket accesses by decoupling key hashing from the walk, and (3) share the hashing hardware among multiple walkers to reduce hardware cost. We next



(a) Baseline design (b) Parallel walkers (c) Parallel walkers each with a decoupled hashing unit (d) Parallel walkers with a shared decoupled hashing unit

Figure 3: Baseline and accelerated indexing hardware.

detail each of these optimizations by evolving the baseline design (Figure 3a) featuring a single hardware context that sequentially executes the code in Listing 1 with no special-purpose hardware.

The first step, shown in Figure 3b, is to accelerate the node list traversals that tend to dominate the indexing time. While each traversal is fundamentally a set of serial node accesses, we observe that there is an abundance of inter-key parallelism, as each individual key lookup can proceed independently of other keys. Consequently, multiple hash buckets can be walked concurrently. Assuming a set of parallel walker units, the expected reduction in indexing time is proportional to the number of concurrent traversals.

The next acceleration target is key hashing, which stands on the critical path of accessing the node list. We make a critical observation that because indexing operations involve multiple independent input keys, key hashing can be decoupled from bucket accesses. By overlapping the node walk for one input key with hashing of another key, the hashing operation can be removed from the critical path, as depicted in Figure 3c.

Finally, we observe that because the hashing operation has a lower latency than the list traversal (a difference that is especially pronounced for in-memory queries), the hashing functionality can be shared across multiple walker units as a way of reducing cost. We refer to a decoupled hashing unit shared by multiple cores as a *dispatcher* and show this design point in Figure 3d.

3.2 First-Order Performance Model

An indexing operation may touch millions of keys, offering enormous inter-key parallelism. In practice; however, parallelism is constrained by hardware and physical limitations. We thus need to understand the practical bottlenecks that may limit the performance of the indexing accelerator outlined in Section 3.1. We consider an accelerator design that is tightly coupled to the core and offers full offload capability of the indexing functionality, meaning that the accelerator uses the core's TLB and L1-D, but the core is otherwise idle whenever the accelerator is running.

We study three potential obstacles to performance scalability of a multi-walker design: (1) L1-D bandwidth, (2) L1-D MSHRs, and (3) off-chip memory bandwidth. The performance-limiting factor of the three elements is determined by the rate at which memory operations are generated at the individual walkers. This rate is a function of the average memory access time (AMAT), memory-level parallelism (MLP, i.e., the number of outstanding L1-D misses), and the computation operations standing on the critical path of each memory access. While the MLP and the number of computation operations are a function of the code, AMAT is affected by the miss ratios in the cache hierarchy. For a given cache organization, the miss ratio strongly depends on the size of the hash table being probed.

Our bottleneck analysis uses a simple analytical model following the observations above. We base our model on the accelerator design with parallel walkers and decoupled hashing units (Figure 3c) connected via an infinite queue. The indexing code, MLP analysis, and required computation cycles are based on Listing 1. We assume 64-bit keys, with eight keys per cache block. The first key to a given cache block always misses in the L1-D and LLC and goes to main memory. We focus on hash tables that significantly exceed the L1 capacity; thus, node pointer accesses always miss in the L1-D, but they might hit in the LLC. The LLC miss ratio is a parameter in our analysis.

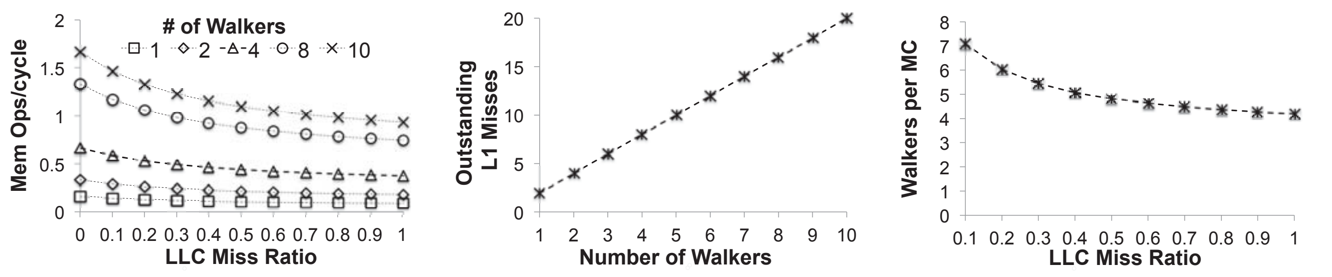
L1-D bandwidth: The L1-D pressure is determined by the rate at which key and node accesses are generated. First, we calculate the total number of cycles required to perform a fully pipelined probe operation for each step (i.e., hashing one key or walking one node in a bucket). Equation 1 shows the cycles required to perform each step as the sum of memory and computation cycles. As hashing and walking are different operations, we calculate the same metric for each of them (subscripted as H and W).

Equation 2 shows how the L1-D pressure is calculated in our model. In the equation, N defines the number of parallel walkers each with a decoupled hashing unit. $MemOps$ defines the L1-D accesses for each component (i.e., hashing one key and walking one node) per operation. As hashing and walking are performed concurrently, the total L1-D pressure is calculated by the addition of each component. We use a subscripted notation to represent the addition; for example: $(X)_{H,W} = (X)_H + (X)_W$.

$$Cycles = AMAT * MemOps + CompCycles \tag{1}$$

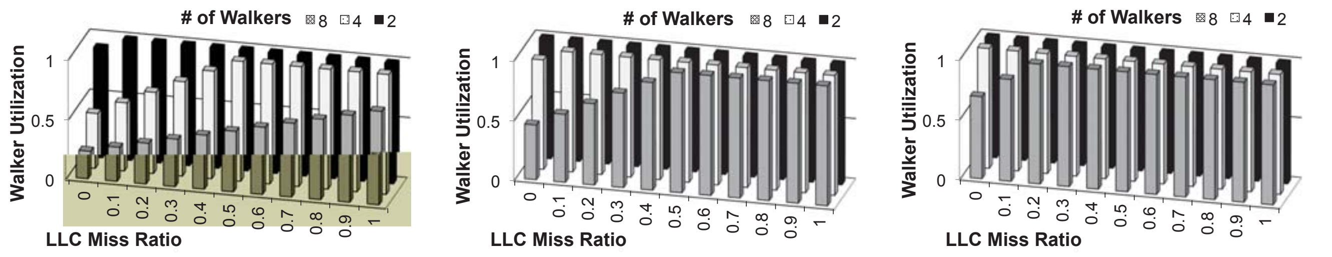
$$MemOps/cycle = (\frac{MemOps}{Cycles})_{H,W} * N \quad \leq \ L1ports \tag{2}$$

Figure 4a shows the L1-D bandwidth requirement as a function of the LLC miss ratio for a varying number of walkers. The number of L1-D ports (typically 1 or 2) limits the L1 accesses per cycle. When the LLC miss ratio is low, a single-ported L1-D becomes the bottleneck for more than six walkers. However, a two-ported L1-D can comfortably support 10 walkers even at low LLC miss ratios.



(a) Constraint: L1-D bandwidth (b) Constraint: L1-D MSHRs (c) Constraint: memory bandwidth

Figure 4: Accelerator bottleneck analysis.



(a) 1 node per bucket (b) 2 nodes per bucket (c) 3 nodes per bucket

Figure 5: Number of walkers that can be fed by a dispatcher as a function of bucket size and LLC miss ratio.

MSHRs: Memory accesses that miss in the L1-D reserve an MSHR for the duration of the miss. Multiple misses to the same cache block (a common occurrence for key fetches) are combined and share an MSHR. A typical number of MSHRs in the L1-D is 8-10; once these are exhausted, the cache stops accepting new memory requests. Equation 3 shows the relationship between the number of outstanding L1-D misses and the maximum MLP the hashing unit and walker can together achieve during a decoupled hash and walk.

$$L1_{Misses} = max(MLP_H + MLP_W) * N \quad \leq \quad MSHRs \quad (3)$$

Based on the equation, Figure 4b plots the pressure on the L1-D MSHRs as a function of the number of walkers. As the graph shows, the number of outstanding misses (and correspondingly, the MSHR requirements) grows linearly with the walker count. Assuming 8 to 10 MSHRs in the L1-D, corresponding to today's cache designs, the number of concurrent walkers is limited to four or five, respectively.

Off-chip bandwidth: Today's server processors tend to feature a memory controller (MC) for every 2-4 cores. The memory controllers serve LLC misses and are constrained by the available off-chip bandwidth, which is around 12.8GB/s with today's DDR3 interfaces. A unit of transfer is a 64B cache block, resulting in nearly 200 million cache block transfers per second. We express the maximum off-chip bandwidth per memory controller in terms of the maximum number of 64-byte blocks that could be transferred per cycle. Equation 4 calculates the number of blocks demanded from the off-chip per operation (i.e., hashing one key or walking one node in a bucket) as a function of L1-D and LLC miss ratio ($L1_{MR}$, LLC_{MR}) and memory operations. Equation 5 shows the model for computing memory bandwidth pressure, which is expressed as the ratio of the expected MC bandwidth in terms of blocks per cycle (BW_{MC}) and the number of demanded cache blocks from the off-chip memory per cycle. The latter is calculated for each component (i.e., hashing unit and walker).

$$OffChipDemands = L1_{MR} * LLC_{MR} * MemOps \quad (4)$$

$$WalkersPerMC \leq \frac{BW_{MC}}{(\frac{OffChipDemands}{Cycles})_{H,W}} \quad (5)$$

Figure 4c shows the number of walkers that can be served by a single DDR3 memory controller providing 9GB/s of effective bandwidth (70% of the 12.8GB/s peak bandwidth [7]). When LLC misses are rare, one memory controller can serve almost eight walkers, whereas at high LLC miss ratios, the number of walkers per MC drops to four. However, our model assumes an infinite buffer between the hashing unit and the walker, which allows the hashing unit to increase the bandwidth pressure. In practical designs, the bandwidth demands from the hashing unit will be throttled by the finite-capacity buffer, potentially affording more walkers within a given memory bandwidth budget.

Dispatcher: In addition to studying the bottlenecks to scalability in the number of walkers, we also consider the potential of sharing the key hashing logic in a dispatcher-based configuration shown in Figure 3d. The main observation behind this design point is that the hashing functionality is dominated by ALU operations and enjoys a regular memory access pattern with high spatial locality, as multiple keys fit in each cache line in column-oriented databases. Meanwhile, node accesses launched by the walkers have poor spatial locality but also have minimal ALU demands. As a result, the ability of a single dispatcher to keep up with multiple walkers is largely a function of (1) the hash table size, and (2) hash table bucket depth (i.e., the number of nodes per bucket). The larger the table, the more frequent the misses at lower levels of the cache hierarchy, and the longer the stall times at each walker. Similarly, the deeper the bucket, the

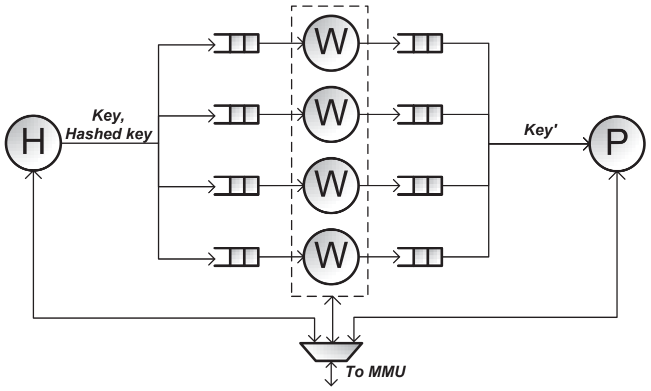


Figure 6: Widx overview. H: dispatcher, W: walker, P: output producer.

more nodes are traversed and the longer the walk time. As walkers stall, the dispatcher can run ahead with key hashing, allowing it to keep up with multiple walkers. This intuition is captured in Equation 6. Total cycles for dispatcher and walkers is a function of AMAT (Equation 1). We multiply the number of cycles needed to walk a node by the number of nodes per bucket to compute the total walking cycles required to locate one hashed key.

$$WalkerUtilization = \frac{Cycles_{node} * Nodes/bucket}{Cycles_{hash} * N} \qquad (6)$$

Based on Equation 6, Figure 5 plots the effective walker utilization given one dispatcher and a varying number of walkers (N). Whenever a dispatcher cannot keep up with the walkers, the walkers stall, lowering their effective utilization. The number of nodes per bucket affects the walkers' rate of consumption of the keys generated by the dispatcher; buckets with more nodes take longer to traverse, lowering the pressure on the dispatcher. The three subfigures show the walker utilization given 1, 2, and 3 nodes per bucket for varying LLC miss ratios. As the figure shows, one dispatcher is able to feed up to four walkers, except for very shallow buckets (1 node/bucket) with low LLC miss ratios.

Summary: Our bottleneck analysis shows that practical L1-D configurations and limitations on off-chip memory bandwidth constrain the number of walkers to around four per accelerator. A single decoupled hashing unit is sufficient to feed all four walkers.

4. WIDX

4.1 Architecture Overview

Figure 6 shows the high-level organization of our proposed indexing acceleration widget, Widx, which extends the decoupled accelerator in Figure 3d. The Widx design is based on three types of units that logically form a pipeline:
(1) a dispatcher unit that hashes the input keys,
(2) a set of walker units for traversing the node lists, and
(3) an output producer unit that writes out the matching keys and other data as specified by the indexing function.

To maximize concurrency, the units operate in a decoupled fashion and communicate via queues. Data flows from the dispatcher toward the output producer. All units share an interface to the host core's MMU and operate within the

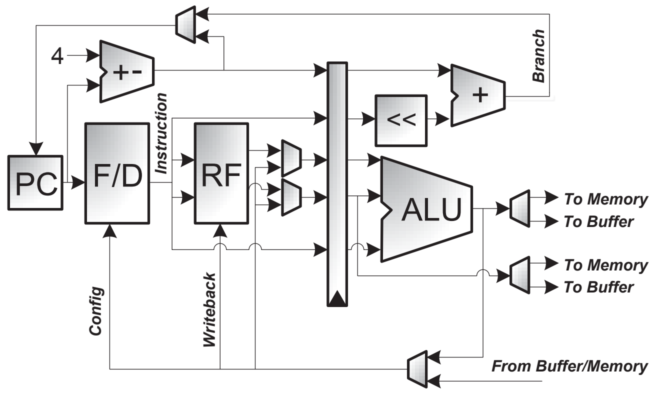


Figure 7: Schematic design of a single Widx unit.

active application's virtual address space. A single output producer generally suffices as store latency can be hidden and is not on the critical path of hash table probes.

A key requirement for Widx is the ability to support a variety of hashing functions, database schemas, and data types. As a result, Widx takes the programmable (instead of fixed-function) accelerator route. In designing the individual Widx units (dispatcher, walker, and output producer), we observe significant commonality in the set of operations that each must support. These include the ability to do simple arithmetic (e.g., address calculation), control flow, and to access memory via the MMU.

Based on these observations, each Widx unit employs a custom RISC core featuring a minimalistic ISA shown in Table 1. In addition to the essential RISC instructions, the ISA also includes a few unit-specific operations to accelerate hash functions with fused instructions (e.g., xor-shift) and an instruction to reduce memory time (i.e., touch) by demanding data blocks in advance of their use. This core serves as a common building block for each Widx unit shown in Figure 6. The compact ISA minimizes the implementation complexity and area cost while affording design reuse across the different units.

Figure 7 shows the internals of a Widx unit. We adopted a design featuring a 64-bit datapath, 2-stage pipeline, and 32 software-exposed registers. The relatively large number of registers is necessary for storing the constants used in key hashing. The three-operand ALU allows for efficient shift operations that are commonly used in hashing. The critical path of our design is the branch address calculation with relative addressing.

4.2 Programming API

To leverage Widx, a database system developer must specify three functions: one for key hashing, another for the node walk, and the last one for emitting the results. Either implicitly or explicitly, these functions specify the data layout (e.g., relative offset of the node pointer within a node data structure) and data types (e.g., key size) used for the indexing operations. Inputs to the functions include the hash table pointer and size, input keys' table pointer and size, and the destination pointer for emitting results.

The functions are written in a limited subset of C, although other programming languages (with restrictions) could also be used. Chief limitations imposed by the Widx programming model include the following: no dynamic memory allocation, no stack, and no writes except by the

Table 1: Widx ISA. The columns show which Widx units use a given instruction type.

Instruction	H	W	P
ADD	✓	✓	✓
AND	✓	✓	✓
BA	✓	✓	✓
BLE	✓	✓	✓
CMP	✓	✓	✓
CMP-LE	✓	✓	✓
LD	✓	✓	✓
SHL	✓	✓	✓
SHR	✓	✓	✓
ST			✓
TOUCH	✓	✓	✓
XOR	✓	✓	✓
ADD-SHF	✓	✓	
AND-SHF	✓		
XOR-SHF	✓		

Table 2: Evaluation parameters.

Parameter	Value
Technology	40nm, 2GHz
CMP Features	4 cores
Core Types	In-order (Cortex A8-like): 2-wide OoO (Xeon-like): 4-wide, 128-entry ROB
L1-I/D Caches	32KB, split, 2 ports, 64B blocks, 10 MSHRs, 2-cycle load-to-use latency
LLC	4MB, 6-cycle hit latency
TLB	2 in-flight translations
Interconnect	Crossbar, 4-cycle latency
Main Memory	32GB, 2 MCs, BW: 12.8GB/s 45ns access latency

output producer. One implication of these restrictions is that functions that exceed a Widx unit's register budget cannot be mapped, as the current architecture does not support push/pop operations. However, our analysis with several contemporary DBMSs shows that, in practice, this restriction is not a concern.

4.3 Additional Details

Configuration interface: In order to benefit from the Widx acceleration, the application binary must contain a Widx control block, composed of constants and instructions for each of the Widx dispatcher, walker, and output producer units. To configure Widx, the processor initializes memory-mapped registers inside Widx with the starting address (in the application's virtual address space) and length of the Widx control block. Widx then issues a series of loads to consecutive virtual addresses from the specified starting address to load the instructions and internal registers for each of its units.

To offload an indexing operation, the core (directed by the application) writes the following entries to Widx's configuration registers: base address and length of the input table, base address of the hash table, starting address of the results region, and a NULL value identifier. Once these are initialized, the core signals Widx to begin execution and enters an idle loop. The latency cost of configuring Widx is amortized over the millions of hash table probes that Widx executes.

Handling faults and exceptions: TLB misses are the most common faults encountered by Widx and are handled by the host core's MMU in its usual fashion. In architectures with software-walked page tables, the walk will happen on the core and not on Widx. Once the missing translation is available, the MMU will signal Widx to retry the memory access. In the case of the retry signal, Widx redirects PC to the previous PC and flushes the pipeline. The retry mechanism does not require any architectural checkpoint as nothing is modified in the first stage of the pipeline until an instruction completes in the second stage.

Other types of faults and exceptions trigger handler execution on the host core. Because Widx provides an atomic all-or-nothing execution model, the indexing operation is completely re-executed on the host core in case the accelerator execution is aborted.

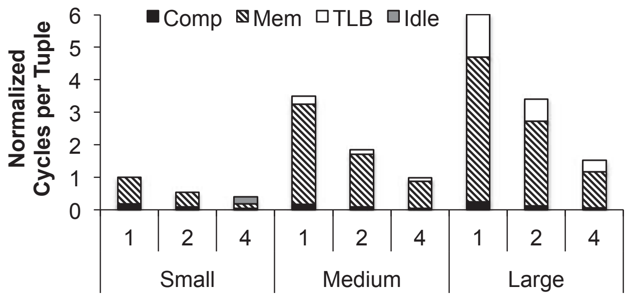
5. METHODOLOGY

Workloads: We evaluate three different benchmarks, namely the hash join kernel, TPC-H, and TPC-DS.

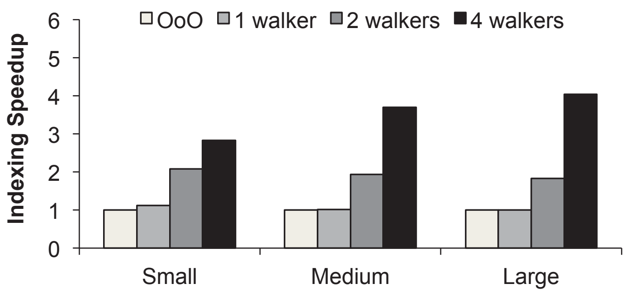
We use a highly optimized and publicly available hash join kernel code [3], which optimizes the "no partitioning" hash join algorithm [4]. We configure the kernel to run with four threads that probe a hash table with up to two nodes per bucket. Each node contains a tuple with 4B key and 4B payload [21]. We evaluate three index sizes, *Small, Medium* and *Large.* The Large benchmark contains 128M tuples (corresponding to 1GB dataset) [21]. The Medium and Small benchmarks contain 512K (4MB raw data) and 4K (32KB raw data) tuples respectively. In all configurations the outer relation contains 128M uniformly distributed 4B keys.

We run DSS queries from the TPC-H [31] and TPC-DS [26] benchmarks on MonetDB 11.5.9 [18] with a 100GB dataset (a scale factor of 100) both for hardware profiling and evaluation in the simulator. Our hardware profiling experiments are carried out on a six-core Xeon 5670 with 96GB of RAM and we used Vtune [19] to analyze the performance counters. Vtune allows us to break down the execution time into functions. To make sure that we correctly account for the time spent executing each database operator (e.g., scan, index), we examine the source code of those functions and group them according to their functionality. We warm up the DBMS and memory by executing all the queries once and then we execute the queries in succession and report the average of three runs. For each run, we randomly generate new inputs for queries with the *dbgen* tool [31].

For the TPC-H benchmark, we run all the queries and report the ones with the indexing execution time greater than 5% of the total query runtime (16 queries out of 22). Since there are a total of 99 queries in the TPC-DS benchmark, we select a subset of queries based on a classification found in previous work [26], considering the two most important query classes in TPC-DS, *Reporting* and *Ad Hoc.* Reporting queries are well-known, pre-defined business questions (queries 37, 40 & 81). Ad Hoc captures the dynamic nature of a DSS system with the queries constructed on the fly to answer immediate business questions (queries 43, 46, 52 & 82). We also choose queries that fall into both categories (queries 5 & 64). In our runs on the cycle-accurate simulator, we pick a representative subset of the queries based on the average time spent in indexing.



(a) Widx walkers cycle breakdown for the Hash Join kernel (normalized to Small running on Widx with one walker)



(b) Speedup for the Hash Join kernel

Figure 8: Hash Join kernel analysis.

Processor parameters: The evaluated designs are summarized in Table 2. Our baseline processor features aggressive out-of-order cores with a dual-ported MMU. We evaluate the Widx designs featuring one, two, and four walkers. Based on the results of the model of Section 3.2, we do not consider designs with more than four walkers. All Widx designs feature one shared dispatcher and one result producer. As described in Section 4, Widx offers full offload capability, meaning that the core stays idle (except for the MMU) while Widx is in use. For comparison, we also evaluate an in-order core modeled after ARM Cortex A8.

Simulation: We evaluate various processor and Widx designs using the Flexus full-system simulator [33]. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications.

We use the SimFlex multiprocessor sampling methodology [33], which extends the SMARTS statistical sampling framework [35]. Our samples are drawn over the entire index execution until the completion. For each measurement, we launch simulations from checkpoints with warmed caches and branch predictors, and run 100K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the subsequent 50K cycles. We measure the indexing throughput by aggregating the tuples processed per cycle both for the baseline and Widx. To measure the indexing throughput of the baseline, we mark the beginning and end of the indexing code region and track the progress of each tuple until its completion. Performance measurements are computed at 95% confidence with an average error of less than 5%.

Power and Area: To estimate Widx's area and power, we synthesize our Verilog implementation with the Synopsys Design Compiler [30]. We use the TSMC 45nm technology (Core library: TCBN45GSBWP, V_{dd}: 0.9V), which is perfectly shrinkable to the 40nm half node. We target a 2GHz clock rate. We set the compiler to the *high* area optimization target. We report the area and power for six Widx units: four walkers, one dispatcher, and one result producer, with 2-entry queues at the input and output of each walker unit.

We use published power estimates for OoO Xeon-like core and in-order A8-like core at 2GHz [22]. We assume the power consumption of the baseline OoO core to be equal to Xeon's nominal operating power [27]. Idle power is estimated to be 30% of the nominal power [20]. As the Widx-enabled design relies on the core's data caches, we estimate the core's private cache power using CACTI 6.5 [25].

6. EVALUATION

We first analyze the performance of Widx on an optimized hash join kernel code. We then present a case study on MonetDB with DSS workloads, followed by an area and energy analysis.

6.1 Performance on Hash Join Kernel

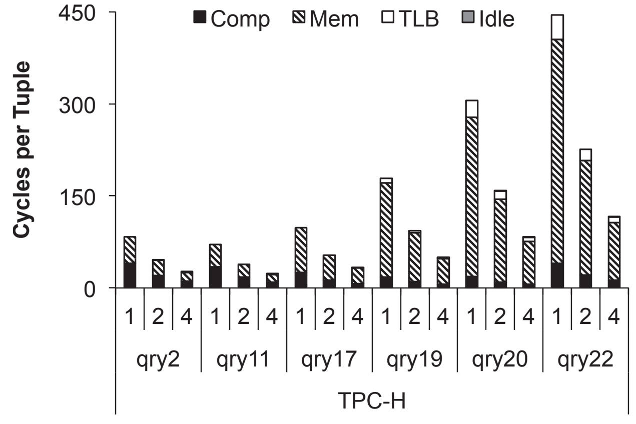
In order to analyze the performance implications of index walks with various dataset sizes, we evaluate three different index sizes; namely, Small, Medium, and Large, on a highly optimized hash join kernel as explained in Section 5.

To show where the Widx cycles are spent we divide the aggregate critical path cycles into four groups. *Comp* cycles go to computing effective addresses and comparing keys at each walker, *Mem* cycles count the time spent in the memory hierarchy, *TLB* quantifies the Widx stall cycles due to address translation misses, and *Idle* cycles account for the walker stall time waiting for a new key from the dispatcher. Presence of Idle cycles indicates that the dispatcher is unable to keep up with the walkers.

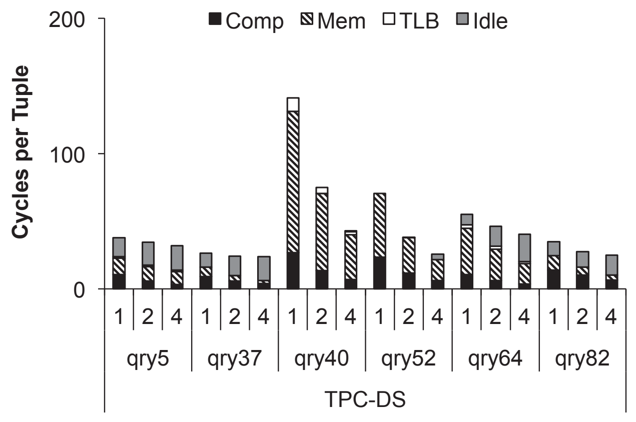
Figure 8a depicts the Widx walkers' execution cycles per tuple (normalized to Small running on Widx with one walker) as we increase the number of walkers from one to four. The dominant fraction of cycles is spent in memory and as the index size grows, the memory cycles increase commensurately. Not surprisingly, increasing the number of walkers reduces the memory time linearly due to the MLP exposed by multiple walkers. One exception is the Small index with four walkers; in this scenario, node accesses from the walkers tend to hit in the LLC, resulting in low AMAT. As a result, the dispatcher struggles to keep up with the walkers, causing the walkers to stall (shown as Idle in the graph). This behavior matches our model's results in Section 3.

The rest of the Widx cycles are spent on computation and TLB misses. Computation cycles constitute a small fraction of the total Widx cycles because the Hash Join kernel implements a simple memory layout, and hence requires trivial address calculation. We also observe that the fraction of TLB cycles per walker does not increase as we enable more walkers. Our baseline core's TLB supports two in-flight translations, and it is unlikely to encounter more than two TLB misses at the same time, given that the TLB miss ratio is 3% for our worst case (Large index).

Figure 8b illustrates the indexing speedup of Widx normalized to the OoO baseline. The one-walker Widx design improves performance by 4% (geometric mean) over the baseline. The one-walker improvements are marginal



(a) Widx walkers cycle breakdown for TPC-H queries



(b) Widx walkers cycle breakdown for TPC-DS queries

Figure 9: DSS on MonetDB. Note that Y-axis scales are different on the two subgraphs.

because the hash kernel implements an oversimplified hash function, which does not benefit from Widx's decoupled hash and walk mechanisms, which overlap the hashing and walking time. However, the performance improvement increases with the number of Widx walkers, which traverse buckets in parallel. Widx achieves a speedup of 4x at best for the Large index table, which performs poorly on the baseline cores due to the high LLC miss ratio and limited MLP.

6.2 Case study on MonetDB

In order to quantify the benefits of Widx on a complex system, we run Widx with the well-known TPC-H benchmark and with the successor benchmark TPC-DS on a state-of-the-art database management system, MonetDB.

Figure 9a breaks down the Widx cycles while running TPC-H queries. We observe that the fraction of the computation cycles in the breakdown increases compared to the hash join kernel due to MonetDB's complex hash table layout. MonetDB stores keys indirectly (i.e., pointers) in the index resulting in more computation for address calculation. However, the rest of the cycle breakdown follows the trends explained in the Hash Join kernel evaluation (Section 6.1). The queries enjoy a linear reduction in cycles per tuple with the increasing number of walkers. The queries with relatively small index sizes (query 2, 11 & 17) do not experience any TLB misses, while the memory-intensive queries (query 19, 20 & 22) experience TLB miss cycles up to 8% of the walker execution time.

Figure 9b presents the cycles per tuple breakdown for TPC-DS. Compared to TPC-H, a distinguishing aspect of the TPC-DS benchmark is the small-sized index tables.[1] Our results verify this fact as we observe consistently lower memory time compared to that of TPC-H (mind the y-axis scale change). As a consequence, some queries (query 5, 37, 64 & 82) go over indexes that can easily be accommodated in the L1-D cache. Widx walkers are partially idle given that they can run at equal or higher speed compared to the dispatcher due to the tiny index, a behavior explained by our model in Section 3.

[1]There are 429 columns in TPC-DS, while there are only 61 in TPC-H. Therefore, for a given dataset size, the index sizes are smaller per column because the same size of dataset is divided over a large number of columns.

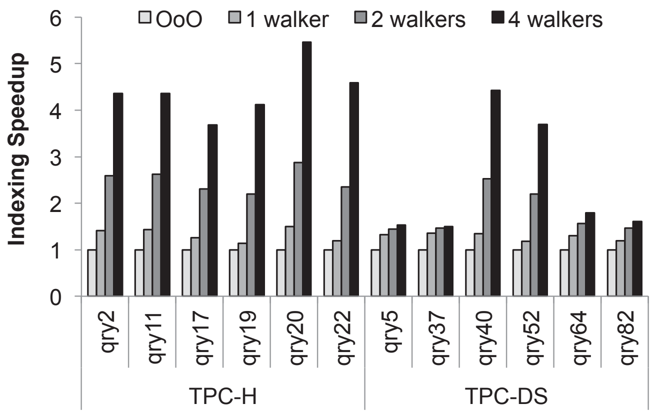


Figure 10: Performance of Widx on DSS queries.

Figure 10 illustrates the performance of Widx on both TPC-H and TPC-DS queries. Compared to OoO, four walkers improve the performance by 1.5x-5.5x (geometric mean of 3.1x). The maximum speedup (5.5x) is registered on TPC-H query 20, which works on a large index with double integers that require computationally intensive hashing. As a result, this query greatly benefits from Widx's features, namely, the decoupled hash and multiple walker units with custom ISA. The minimum speedup (1.5x) is observed on TPC-DS query 37 due to L1-resident index (L1-D miss ratio $<1\%$). We believe that this is the lower limit for our design because there are only a handful of unique index entries.

We estimate the benefits of indexing acceleration at the level of the entire query by projecting the speedups attained in the Widx-accelerated design onto the indexing portions of the TPC-H and TPC-DS queries presented in Figure 2a. By accelerating just the indexing portion of the query, Widx speeds up the query execution by a geometric mean of 1.5x and up to 3.1x (query 17). Our query speedups are limited by the fraction of the time spent in indexing throughout the overall execution. In query 17, the achieved overall speedup is close to the indexing-only speedup with the four-walker design as 94% of the execution time is spent indexing. The lowest overall speedup (10%) is obtained in query 37 because only 29% of the query execution is offloaded to Widx and as explained above, the query works on an L1-resident index.

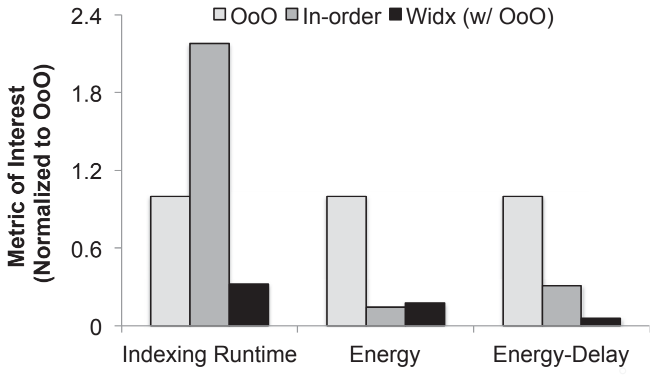


Figure 11: Indexing Runtime, Energy and Energy-Delay metric of Widx (lower is better).

6.3 Area and Energy Efficiency

To model the area overhead and power consumption of Widx, we synthesized our RTL design in the TSMC 40nm technology. Our analysis shows that a single Widx unit (including the two-entry input/output buffers) occupies $0.039mm^2$ with a peak power consumption of $53mW$ at 2GHz. Our power and area estimates are extremely conservative due to the lack of publicly available SRAM compilers in this technology. Therefore, the register file and instruction buffer constitute the main source of area and power consumption of Widx. The Widx design with six units (dispatcher, four walkers, and an output producer) occupies $0.24mm^2$ and draws $320mW$. To put these numbers into perspective, an in-order ARM Cortex A8 core in the same process technology occupies $1.3mm^2$, while drawing $480mW$ including the L1 caches [22]. Widx's area overhead is only 18% of Cortex A8 with comparable power consumption, despite our conservative estimates for Widx's area and power. As another point of comparison, an ARM M4 microcontroller [1] with full ARM Thumb ISA support and a floating-point unit occupies roughly the same area as the single Widx unit. We thus conclude that Widx hardware is extremely cost-effective even if paired with very simple cores.

Figure 11 summarizes the trade-offs of this study by comparing the average runtime, energy consumption, and energy-delay product of the indexing portion of DSS workloads. In addition to the out-of-order baseline, we also include an in-order core as an important point of comparison for understanding the performance/energy implications of the different design choices.

An important conclusion of the study is that the in-order core performs significantly worse (by 2.2x on average) than the baseline OoO design. Part of the performance difference can be attributed to the wide issue width and reordering capability of the OoO core, which benefits the hashing function. The reorder logic and large instruction window in the OoO core also help in exposing the inter-key parallelism between two consecutive hash table lookups. For queries that have cache-resident index data, the loads can be issued from the imminent key probe early enough to partially hide the cache access latency.

In terms of energy efficiency, we find that the in-order core reduces energy consumption by 86% over the OoO core. When coupled with Widx, the OoO core offers almost the same energy efficiency (83% reduction) as the in-order design. Despite the full offload capability offered by Widx and its high energy efficiency, the total energy savings are limited by the high idle power of the OoO core.

In addition to energy efficiency, QoS is a major concern for many database workloads. We thus study the efficiency of various designs on both performance and energy together via the energy-delay product metric. Due to its performance and energy-efficiency benefits, Widx improves the energy-delay product by 5.5x over the in-order core and by 17.5x over the OoO baseline.

7. DISCUSSION

Other join algorithms and software optimality: In this paper, we focused on hardware-oblivious hash join algorithms that run on the state-of-the-art software. In order to exploit on-chip cache locality, researchers have proposed hardware-conscious approaches that have a table-partitioning phase prior to the main join operation [23]. In this phase, a hash table is built on each small partition of the table, thus making the individual hash tables cache-resident. The optimal partition size changes across hardware platforms based on the cache size, TLB size, etc.

Widx's functionality does not require any form of data locality, and thus is independent of any form of data partitioning. Widx is, therefore, equally applicable to hash join algorithms that employ data partitioning prior to the main join operation [23]. Due to the significant computational overhead involved in table partitioning, specialized hardware accelerators that target partitioning [34] can go hand in hand with Widx.

Another approach to optimize join algorithms is the use of SIMD instructions. While the SIMD instructions aid hash-joins marginally [16, 21], another popular algorithm, sort-merge join, greatly benefits from SIMD optimizations during the sorting phase. However, prior work [2] has shown that hash join clearly outperforms the sort-merge join. In general, software optimizations target only performance, whereas Widx both improves performance and greatly reduces energy.

Broader applicability: Our study focused on MonetDB as a representative contemporary database management system; however, we believe that Widx is equally applicable to other DBMSs. Our profiling of HP Vertica and SAP HANA indicate that these systems rely on indexing strategies, akin to those discussed in this work, and consequently, can benefit from our design. Moreover, Widx can easily be extended to accelerate other index structures, such as balanced trees, which are also common in DBMSs.

LLC-side Widx: While this paper focused on a Widx design tightly coupled with a host core, Widx could potentially be located close to the LLC instead. The advantages of LLC-side placement include lower LLC access latencies and reduced MSHR pressure. The disadvantages include the need for a dedicated address translation logic, a dedicated low-latency storage next to Widx to exploit data locality, and a mechanism for handling exception events. We believe the balance is in favor of a core-coupled design; however, the key insights of this work are equally applicable to an LLC-side Widx.

8. RELATED WORK

There have been several optimizations for hash join algorithms both in hardware and software. Recent work proposes vector instruction extensions for hash table probes [16]. Although promising, the work has several important limitations. One major limitation is the DBMS-specific solution, which is limited to the *Vectorwise* DBMS. Another drawback is the vector-based approach, which limits its performance due to the lock-stepped execution in the vector unit. Moreover, the Vectorwise design does not accelerate key hashing, which constitutes up to 68% of lookup time. Finally, the vector-based approach keeps the core fully engaged, limiting the opportunity to save energy.

Software optimizations for hash join algorithms tend to focus on the memory subsystem (e.g., reducing cache miss rates through locality optimizations) [21, 23]. These techniques are orthogonal to our approach and our design would benefit from such optimizations. Another memory subsystem optimization is to insert prefetching instructions within the hash join code [5]. Given the limited intra-tuple parallelism, the benefits of prefetching in complex hash table organizations are relatively small compared to the inter-tuple parallelism as shown in [16].

Recent work has proposed on-chip accelerators for energy efficiency (dark silicon) in the context of general-purpose (i.e., desktop and mobile) workloads [12, 13, 14, 28, 32]. While these proposals try to improve the efficiency of the memory hierarchy, the applicability of the proposed techniques to big data workloads is limited due to the deep software stacks and vast datasets in today's server applications. Also, existing dark silicon accelerators are unable to extract memory-level parallelism, which is essential to boost the efficiency of indexing operations.

1980s witnessed proliferation of database machines, which sought to exploit the limited disk I/O bandwidth by coupling each disk directly with specialized processors [8]. However, high cost and long design turnaround time made custom designs unattractive in the face of cheap commodity hardware. Today, efficiency constraints are rekindling an interest in specialized hardware for DBMSs [6, 11, 17, 24, 34]. Some researchers proposed offloading hash-joins to network processors [11] or to FPGAs [6] for leveraging the highly parallel hardware. However, these solutions incur invocation overheads as they communicate through PCI or through high-latency buses, which affect the composition of multiple operators. Moreover, offloading the joins to network processors or FPGAs requires expensive dedicated hardware, while Widx utilizes the on-chip dark silicon. We believe that our approach for accelerating schema-aware indexing operations is insightful for the next-generation of data processing hardware.

9. CONCLUSION

Big data analytics lie at the core of today's business. DBMSs that run analytics workloads rely on indexing data structures to speed up data lookups. Our analysis of MonetDB, a modern in-memory database, on a set of data analytics workloads shows that hash-table-based indexing operations are the largest single contributor to the overall execution time. Nearly all of the indexing time is split between ALU-intensive key hashing operations and memory-intensive node list traversals. These observations, combined with a need for energy-efficient silicon mandated by the slowdown in supply voltage scaling, motivate Widx, an on-chip accelerator for indexing operations.

Widx uses a set of programmable hardware units to achieve high performance by (1) walking multiple hash buckets concurrently, and (2) hashing input keys in advance of their use, removing the hashing operations from the critical path of bucket accesses. By leveraging a custom RISC core as its building block, Widx ensures the flexibility needed to support a variety of schemas and data types. Widx minimizes area cost and integration complexity through its simple microarchitecture and through tight integration with a host core, allowing it to share the host core's address translation and caching hardware. Compared to an aggressive out-of-order core, the proposed Widx design improves indexing performance by 3.1x on average, while saving 83% of the energy by allowing the host core to be idle while Widx runs.

10. ACKNOWLEDGMENTS

We thank Cansu Kaynak, Alexandros Daglis, Stratos Idreos, Djordje Jevdjic, Pejman Lotfi-Kamran, Pınar Tözün, Stavros Volos, and the anonymous reviewers for their insightful feedback on the paper. This work is supported in part by the HP Labs Innovation Research Program and Research Promotion Foundation (RPF), with grants IRP-11472 and 0609(BIE)/09 respectively.

11. REFERENCES

[1] ARM M4 Embedded Microcontroller. http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php.

[2] C. Balkesen, G. Alonso, and M. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1), 2013.

[3] 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.

[4] 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.

[5] 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.

[6] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[7] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, 2011.

[8] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[10] Global Server Hardware Market 2010-2014. http://www.technavio.com/content/global-server-hardware-market-2010-2014.

[11] 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.

[12] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th Annual International Symposium on High Performance Computer Architecture*, 2011.

[13] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[14] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[15] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4), 2011.

[16] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. Vector extensions for decision support DBMS acceleration. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[17] IBM Netezza Data Warehouse Appliances. http://www-01.ibm.com/software/data/netezza/.

[18] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[19] Intel Vtune. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[20] Intel Xeon Processor 5600 Series Datasheet, Vol 2. http://www.intel.com/content/www/us/en-/processors/xeon/xeon-5600-vol-2-datasheet.html.

[21] 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.

[22] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

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

[24] R. Mueller, J. Teubner, and G. Alonso. Glacier: A query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[25] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[26] M. Poess, R. O. Nambiar, and D. Walrath. Why you should run TPC-DS: A workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[27] S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. A dual-core multi-threaded Xeon processor with 16MB L3 cache. In *Solid-State Circuits Conference*, 2006.

[28] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. Taylor. Efficient complex operators for irregular codes. In *Proceedings of the 17th Annual International Symposium on High Performance Computer Architecture*, 2011.

[29] J. E. Short, R. E. Bohn, and C. Baru. How Much Information? 2010 Report on Enterprise Server Information, 2011.

[30] Synopsys Design Compiler. http://www.synopsys.com/.

[31] The TPC-H Benchmark. http://www.tpc.org/tpch/.

[32] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[33] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.

[34] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[35] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.