Eighth International Workshop on Data Management on New Hardware (DaMoN 2012)

May 21, 2012
Scottsdale, Arizona, USA

In conjunction with ACM SIGMOD/PODS Conference

Shimin Chen and Stavros Harizopoulos
(Editors)

Industrial Sponsor

FOREWORD

Objective

The aim of this one-day workshop is to bring together researchers who are interested in optimizing database performance on modern computing infrastructure by designing new data management techniques and tools.

Topics of Interest

The continued evolution of computing hardware and infrastructure imposes new challenges and bottlenecks to program performance. As a result, traditional database architectures that focus solely on I/O optimization increasingly fail to utilize hardware resources efficiently. Multi-core CPUs, GPUs, new memory and storage technologies (such as flash and phase change memory), and low-power hardware impose a great challenge to optimizing database performance. Consequently, exploiting the characteristics of modern hardware has become an important topic of database systems research.

The goal is to make database systems adapt automatically to the sophisticated hardware characteristics, thus maximizing performance transparently to applications. To achieve this goal, the data management community needs interdisciplinary collaboration with computer architecture, compiler and operating systems researchers. This involves rethinking traditional data structures, query processing algorithms, and database software architectures to adapt to the advances in the underlying hardware infrastructure.

Workshop Co-Chairs

Shimin Chen, HP Labs China (shimin.chen@hp.com)
Stavros Harizopoulos, Nou Data (stavros@noudata.com)

Program Committee

Phil Gibbons	(Intel Labs)
Nikos Hardavellas	(Northwestern University)
Qiong Luo	(Hong Kong University of Science and Technology)
Ryan Johnson	(University of Toronto)
Bongki Moon	(University of Arizona)
Ippokratis Pandis	(IBM Research)
Kenneth Ross	(Columbia University)
Eric Sedlar	(Oracle Labs)

TABLE OF CONTENTS

A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-Intensive Workloads

Darius Šidlauskas
Aalborg University
darius@cs.aau.dk

Christian S. Jensen
Aarhus University
csj@cs.au.dk

Simonas Šaltenis
Aalborg University
simas@cs.aau.dk

ABSTRACT

Deployments of networked sensors fuel online applications that feed on real-time sensor data. This scenario calls for techniques that support the management of workloads that contain queries as well as very frequent updates. This paper compares two well-chosen approaches to exploiting the parallelism offered by modern processors for supporting such workloads. A general approach to avoiding contention among parallel hardware threads and thus exploiting the parallelism available in processors is to maintain two copies, or snapshots, of the data: one for the relatively long-duration queries and one for the frequent and very localized updates. The snapshot that receives the updates is frequently made available to queries, so that queries see up-to-date data. The snapshots may be physical or virtual. Physical snapshots are created using the C library `memcpy` function. Virtual snapshots are created by the `fork` system function that creates a new process that initially has the same data snapshot as the process it was forked from. When the new process carries out updates, this triggers the actual memory copying in a copy-on-write manner at memory page granularity. This paper characterizes the circumstances under which each technique is preferable. The use of physical snapshots is surprisingly efficient.

1. INTRODUCTION

The increasing number of chips per processor, cores per chip, and hardware threads per core in chip multiprocessors (CMPs) endows commodity computer systems with substantial parallel processing capabilities. However, due to the lack of concurrency in software, this parallelism is often not exploited. Parallelization of software is not trivial in practice because programmers must deal with the complications of concurrency control (CC), which often limits the performance gains by creating bottlenecks and serializing the code. This is especially difficult for update-intensive workloads, where frequent and rapid updates are intermixed with relatively long-running queries.

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

One way to enable thread-level parallelism in such workloads is to let threads operate on different data snapshots. A read-only snapshot is used to serve threads that process queries. A write-only snapshot is used to serve threads that process updates. Every so often a copy of the latter is made available for querying so that near up-to-date query results are obtained.

There are several reasons why snapshotting is attractive. First, it allows to isolate otherwise conflicting operations. In a single-updater scenario, the updater can operate on the latest data while multiple queries progress concurrently on the snapshot. Multiple updater scenarios, where updates are applied to individual objects (e.g., moving objects, tuples), can also be handled efficiently. A simple CC scheme suffices to parallelize single-object updates. Second, even entire data scans can be assumed to happen atomically and represent query results that are valid as of some time (when the underlying snapshot was taken). Third, by eliminating interference between conflicting operations, one can greatly simplify the design of concurrent algorithms and data structures. Consequently, the verification of their correctness is simplified. Finally, current technologies permit very frequent snapshotting, meaning that very up-to-date states are available to queries, enabling application in many domains.

A key question then how to create a snapshot? This paper investigates two fundamentally different approaches to data snapshotting in main memory. *Physical snapshotting* uses the standard C library `memcpy` operation to copy a contiguous region of memory. The source and destination regions must not overlap. As memory copying is used in many contexts, it is highly optimized on modern platforms, and `memcpy` uses a number of hardware-assisted optimizations. For example, it looks for opportunities to perform as much of the copying as possible with the widest data type supported by the hardware. With large copyings (bigger than half of the last-level cache, LLC), instructions that bypass the CPU cache (reduce cache misses and trashing) are employed. Some code parts are tuned in assembly.

Virtual snapshotting uses the `fork` operation available in Unix-like systems to create a snapshot by forking a child process that is an identical copy of the parent. The operating system achieves this by mapping the virtual address space of both processes to the same physical memory addresses. Subsequent modifications are handled in a copy-on-write manner at memory page granularity. Such virtual memory systems are highly optimized on modern processors and have built-in support. The dedicated memory management unit in a processor is responsible for the virtual-to-physical

address translation. The most recently used mappings of the operating system's page table are stored in a dedicated cache, a translation lookaside buffer (TLB), which further accelerates the translation process.

Both techniques accomplish the same result and share several attractive qualities. The techniques make use of operations (`memcpy` and `fork`) that are general purpose and therefore available across many platforms. A long history of usage in varying settings has resulted in them being highly optimized. Physical snapshotting represents eager copying and is a brute-force approach, while virtual snapshotting exemplifies well-known concepts such as incremental update and lazy copying or copying on demand.

Our study of virtual snapshotting is motivated by its recent successful application in the HyPer [13] main memory database prototype where `fork`-based snapshots are used to efficiently process OLTP and OLAP workloads on the same database. OLTP is performed in a regular manner, while periodical forking of child processes is used to support OLAP queries. This way, virtual snapshotting helps eliminate interference between analytic queries and transaction processing. As a result, very high OLTP and OLAP throughputs are achieved.

Mühe et al. [14] propose several other snapshotting techniques, including purely software-controlled and hybrid ones, and compare them against HyPer's `fork`-based approach. The `fork`-based technique is a clear winner in all considered aspects. We compare this `fork`-based approach with an approach based on the `memcpy` call.

Our study of physical snapshotting is motivated by its recent successful application in a memory-resident index structures [18]. There, a query-only snapshot is created by explicitly copying the whole index structure. This allows to eliminate the interference between rapid single-object updates and relatively long-running range queries. It is shown that `memcpy` outperforms a software-based snapshotting technique that uses bulk-loading [8]. It demonstrates that very frequent snapshotting, on the order of tens of milliseconds, is feasible.

Physical and virtual snapshotting accomplish the same result in different ways. Virtual snapshotting is expected to excell when snapshots are large and a small fraction of the data is being updated intensively. Physical snapshotting is expected to be most efficient when snapshots are smaller and are updated uniformly. The paper aims to quantify the following trade-offs between the two approaches:

1. Time per snapshot creation.
2. Snapshot updating under different update distributions.
3. Cost to query a snapshot.
4. Memory consumption.
5. Performance on four different platforms.

The remainder of the paper is organized as follows. Section 2 covers background on the `memcpy` and `fork` operations and explains how the two snapshotting techniques are implemented. Section 3 covers the empirical study, and Section 4 concludes the paper. Appendices contain auxiliary content.

2. IMPLEMENTATION

We describe first the built-in support for the `memcpy` and `fork` operations, then describe how physical and virtual snapshotting are implemented using these.

2.1 Built-in Support for `fork` and `memcpy`

To support virtual memory, operating systems use a page table that maps virtual memory page addresses to physical addresses, thus adding a layer of indirection that abstracts address spaces of different types of storage (e.g., CPU cache, main memory, disk). When a process is forked from a parent process, the operating system thus assigns a page table and a virtual address space to the new process. Initially, all the virtual addresses in the page table map to the same physical addresses as do the page table of the parent process. Each entry on the tables of both processes has a copy-on-write (CoW) flag. Before each subsequent page modification in either process, the flag is checked. If it is set, a new page is allocated, the data from the old page is copied, the modification is made on the new page, and the CoW flag of both pages is unset. Thus, independent page copies are made on demand.

To obtain efficient support for virtual memory, modern CPUs include a memory management unit (MMU) that is responsible for the efficient virtual-to-physical mapping of addresses. A cache within the MMU, the Translation Lookaside Buffer (TLB), stores recent translations. The TLB accelerates translations by reducing the need to reread entries from memory.

Large main memories result in large page tables. For example, 1GB address space with a typical 4KB page size requires a page table with 256k entries. Consequently, operating systems utilize the multiple page size capabilities of the underlying hardware. For example, the Intel machines used in this paper support 2MB pages, and Sun's T2 supports four different page sizes: 8KB, 64KB, 4MB, and 256MB (see Table 2).

To physically copy a contiguous region of memory the `memcpy` operation employs a number of software and hardware optimizations. Attempts are made to copy as much data as possible with the widest data type supported by the hardware. If the widest type is 16 bytes (e.g., the 128-bit SIMD registers on Nehalem), a common optimization is to switch to instructions that move chunks of 16 bytes at a time as soon as a suitable 16-byte aligned boundary is reached.

The data being copied often has very poor temporal locality (used "once"). To reduce CPU cache trashing, `memcpy` employs instructions with so-called non-temporal hints that bypass the cache sub-system [1, 12] and instead write to one of the cache-line-sized buffers. Since the data being copied has strong spatial locality, the writes to the same cache line are grouped quickly and written to memory. Thus, write cache misses are reduced significantly. Sun's T2 architecture is able to avoid similar write cache misses by using a block initializing store instruction [17].

Since `memcpy` exhibits a constant stride access pattern, where consecutive memory accesses are made to consecutive memory addresses, data prefetching is exploited heavily. Current processors support multiple outstanding cache misses simultaneously in order to hide memory latencies. A hardware prefetcher detects such access patterns and fetches data automatically right before it is actually needed.

There are many practical reasons that make it impossible to achieve the maximum memory bandwidth in practice [9]. However, given the above mentioned optimizations, one can expect to get really close.

2.2 Implementation Overview

We define a snapshot as an instantaneous view of the data that can be shared among multiple threads or processes. We consider a setting with two memory-resident data snapshots. One snapshot, S_1, is write-only and represents the most up-to-date state of the data. All incoming updates are applied to S_1. The second snapshot, S_2, is read-only, represents a slightly outdated state of the data, and is dedicated for querying. During workload processing, both snapshots are simultaneously accessed by p threads, where p is the maximum number of hardware threads available. Therefore, a thread accesses the snapshot that supports the operation it wants to perform. For S_2 to be a consistent copy of S_1, the processing threads are quiesced before the snapshotting.

The snapshotting frequency, F_s, defines how often changes in S_1 are reflected in S_2 and thus defines the freshness of S_2. F_s can be expressed in time units (continuous) or in the number of updates (discrete). For example, snapshotting can be done every 1 second or every 1000th update. With snapshotting according to time, no data item in S_2 is older than F_s time units. With discrete snapshotting at least $\frac{N-F_s}{N}$ of the data items in S_2 are up-to-date.

For simplicity, we assume the data is stored in an array of N items. An update occurs at a given (random) array index. To avoid locking/latching overhead, we assume data items have a fixed length of 64 bits and use atomic instructions to update the items. This way, updates always leave S_1 in a consistent state. We assume query operations are range scans. We chose the simplest possible "data structure" as well as update "algorithms" to make sure that the time to perform an update is dominated by the memory access cost as well as the cost of a possible copy-on-write.

It is trivial to extend our assumptions to maintain multiple read-only snapshots.

Physical snapshotting In physical snapshotting, an array of the size of S_1 is allocated to obtain the read-only snapshot S_2. That is, the snapshots have different virtual as well as physical addresses. We divide both arrays (snapshots) into t sections $s_1, \ldots, s_t$, where t is the number of threads that is used to copy the array. We assume N is a multiple of t. Thread t_i is responsible for copying items from section s_i in S_1 to section s_i in S_2. Each thread uses the standard C library `memcpy` call.

Since all p processing threads are suspended during snapshotting, all hardware threads can be used for copying ($t = p$). To minimize the time spent starting and stopping threads, $p + t$ software threads are initialized in advance once and are later used throughout the workload processing. During snapshotting, the t threads do the copying, while the p threads are "parked" idle. After snapshotting, the t threads are parked, while the p threads continue their job.

Extra care must be taken if data structures with pointers are to be supported. Otherwise, the `memcpy`'ed pointers in S_2 still reference the original structures, yielding an S_2 that is not a faithful copy of S_1. The solution is to use container-based memory allocation where all pointers in a container are interpreted as offsets from a container base address [18]. We do not consider pointers here.

Virtual snapshotting In virtual snapshotting, the memory for the S_2 is not explicitly allocated. Instead, using the `fork` system call, a child process is created by the process where S_1 is allocated. Initially, the virtual address space of the child process is mapped to the same physical addresses as in the parent process. A virtual copy S_2 of S_1 is thus created. Update processing follows the same procedure as in physical snapshotting: p threads perform single-item writes on S_1. As explained earlier, subsequent snapshot modifications are handled by the copy-on-write mechanism.

To enable querying of S_2, virtual snapshotting requires communication/synchronization between the two processes. The queries have to be delegated to the most recently forked process. The needed inter process communication (IPC) occurs via *shared memory*, which is considered to be the fastest form of IPC because there is no intermediation (e.g., a pipe, a message queue).

As in physical snapshotting, the p threads are suspended before a new snapshot is created. Only one thread is needed to create a virtual S_2, namely one that executes the `fork` system call. As soon as `fork` returns, the p threads continue their work. To support a number q of simultaneous queries in the S_2 process, the q threads have to be always created from scratch[1].

Since the entire address space is replicated in the S_2 process, there are no restrictions on snapshot contiguity in main memory and the use of pointers. (See the first two columns in Table 1).

3. EXPERIMENTS

We conduct the performance study on four multi-core platforms: a dual quad-core AMD Opteron 2350 (Barcelona), a dual quad-core Intel Xeon X5550 (Nehalem), a quad-core Intel Core i7-2600K (Sandy Bridge), and an 8-core Sun Niagara 2 (T2) with 64 hardware threads in total. Since the Sun machine runs Solaris 10 and the Intel/AMD machines run Linux, the used system calls are tested on two Unix flavors. Also our hardware setting captures two CMP design approaches: *fat-camp* (the Intel and AMD machines) and *lean-camp* (the Sun machine) [11]. We therefore believe the reported findings are relevant for a wide spectrum of platforms. More details are given in Table 2.

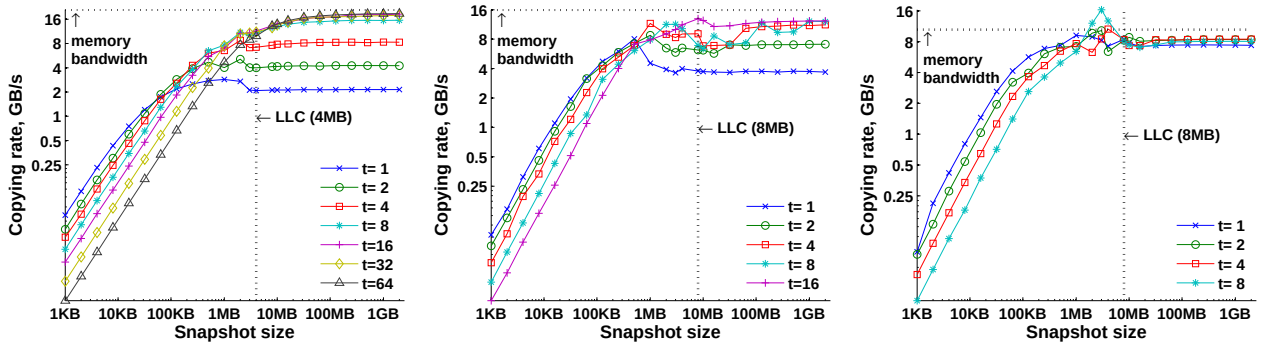
3.1 Snapshot Creation

We study the cost of snapshot creation when varying the snapshot size. In physical snapshotting, this is the amount of memory to be copied; and in virtual snapshotting, it determines the number of page table entries that need to be replicated.

`memcpy` performance Figure 1 shows the copying rate (GB/s) when varying the snapshot size when using different numbers t of threads for the copying.

On all platforms, `memcpy` throughput is increasing until the snapshot size approaches half of the LLC size, implying that both snapshots still fit in the LLC. At this point, `memcpy` starts to execute the code path that uses instructions with non-temporal hints (see Section 2.1). These instructions bypass the cache sub-system in order to minimize cache trashing, but are more expensive. At this point, the gain from additional copier threads becomes significant: when one thread is stalled on memory access, others can proceed. Therefore, with more threads it becomes possible to saturate the available memory bandwidth. However, none of the ma-

[1]Depending on the workload, the number of threads in S_2 for query processing can be different from the number of threads in S_1.



(a) T2 (b) Nehalem (c) SandyBridge

Figure 1: Physical snapshotting performance.

chines requires all hardware threads in order to sustain their maximum copying rates. For instance, on T2 the copying rate barely improves beyond 8 threads. We found that the number t of copier threads needed to saturate the memory bandwidth corresponds to the number of memory channels on a machine (i.e, one thread per channel).

On T2, a lean-camp processor, eight hardware threads (per core) share the core in a fair manner: a context switch occurs on each clock cycle so that an instruction is issued from the least recently used (and ready) thread. As a result, a constant performance gain is obtained with each additional thread (Figure 1(a)), and `memcpy` is able to reach 88% of the maximum memory bandwidth.

On the Intel machines, fat-camp processors, the two threads within a core are not utilized in a fair manner. The second thread only helps in using the pipeline slots that are unused by the first thread. Therefore, our division of the snapshot into equal sized sections and assigning each section to a separate hardware thread is vulnerable to load imbalances (discussed in Appendix A).

The second copier thread might slow the entire snapshotting down. This explains the sporadic performance drops with 4 and more threads in Figures 1(b) and 1(c). However, as soon as the memory bandwidth is depleted, the copying rate stabilizes. The memory bandwidth utilization on the Nehalem and Sandy Bridge machine reaches 76% and 80%, respectively.

With snapshot sizes within 2–4 MB, the copying rate when using all 8 hardware threads exceeds the memory bandwidth on the Sandy Bridge machine (Figure 1(c)). One might expect a similar behavior on all the machines when both snapshots reside completely in the LLC and operate at cache speeds. However, only Intel's new Sandy Bridge architecture exhibits this. Also, with snapshot sizes that exceed LLC, a single Sandy Bridge thread is able to saturate almost all memory bandwidth. This may be partly due to Intel's so-called Turbo Boost Technology that dynamically increases a busy core's frequency while disabling idle cores.

Similar performance trends are observed on the AMD Barcelona machine (not shown). At 76%, the utilization of the maximum memory bandwidth is also high. Overall, the memory bandwidth utilization in `memcpy` is high on all platforms. To achieve this, the number of threads must be chosen carefully on each machine.

`fork` performance Performance of virtual snapshotting is shown in Figure 2. The y-axis shows the time it takes to fork a child process. The bottom x-axis plots the snapshot size in bytes, while the top x-axis gives the size in a number of memory pages.

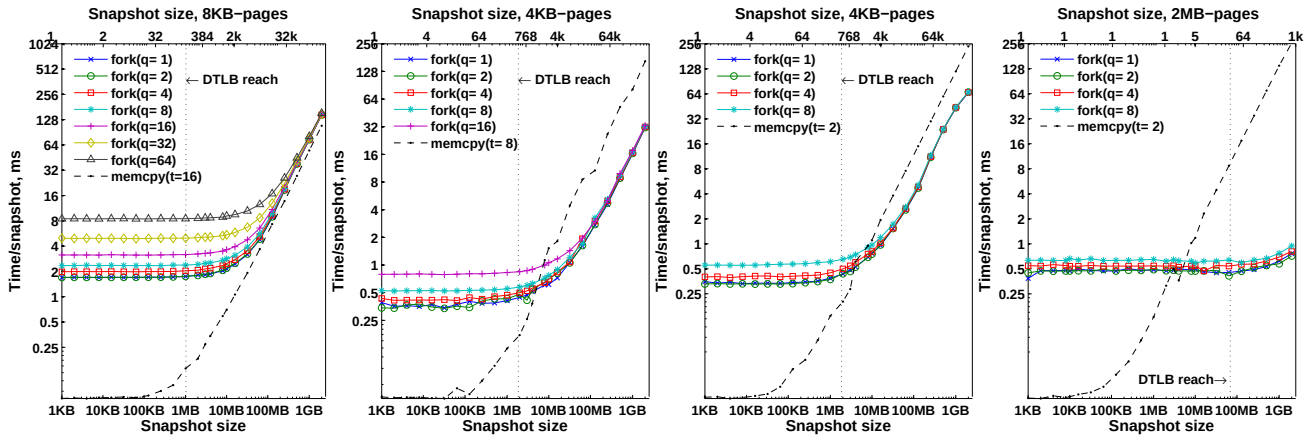
Since all q query threads must be recreated in a newly forked child process, we show how the cost increases with number of threads (different lines). To compare the performance of virtual and physical snapshotting, we additionally plot the physical copying time (the copying rate from Figure 1 converted to time/snapshot). A `memcpy(t=#)` curve with a given number # of parallel copier threads depicts this.

The TLB cache size has a major impact on virtual snapshotting performance. As long as the TLB reach (TLB size × memory page size) covers both snapshots, the `fork` cost remains low and stable. For example, to fork a single-threaded process ($q = 1$) takes less than 0.5 ms on the Intel and AMD machines and circa 2 ms on the T2 machine. With snapshot sizes exceeding the TLB reach, the cost grows linearly.

The number of threads one needs to spawn in a newly created child process can have a big impact. The performance difference is especially visible with small snapshot sizes (within TLB reach). The biggest differences are on T2, where one thread spawns many threads at a relatively low speed (1.2 GHz). With bigger snapshot, the impact decreases and is eventually dominated by the `fork` costs. Thus, if an application needs to use all hardware threads and operates within the TLB reach, the cost of forking can be excessive.

Comparing the two snapshotting techniques, we can see that with small snapshot sizes (half of LLC) physical snapshotting significantly outperforms virtual snapshotting on all platforms. With bigger snapshots, virtual snapshotting outperforms physical snapshotting on all machines except T2 (discussed in Appendix B.2). With snapshot sizes of 2 GB, the virtual snapshot creation on the AMD (not shown) and Intel machines is 4–5 times faster, while on the T2 machine, it is the same as that of physical snapshot creation.

Huge pages As explained in Section 2.1, modern processors and operating systems support different memory page sizes. For instance, our Linux machines use 4 KB page sizes by default, but support huge page sizes of 2 MB as well. This can shrink the number of virtual-to-physical translations by a factor of 256. Although the Sandy Bridge architecture separates TLB entries by type and has only 32 entries for huge pages, the TLB reach still increases by a factor of 32: small



(a) T2 (b) Nehalem (c) Sandy Bridge (d) Sandy Bridge

Figure 2: Virtual snapshotting performance.

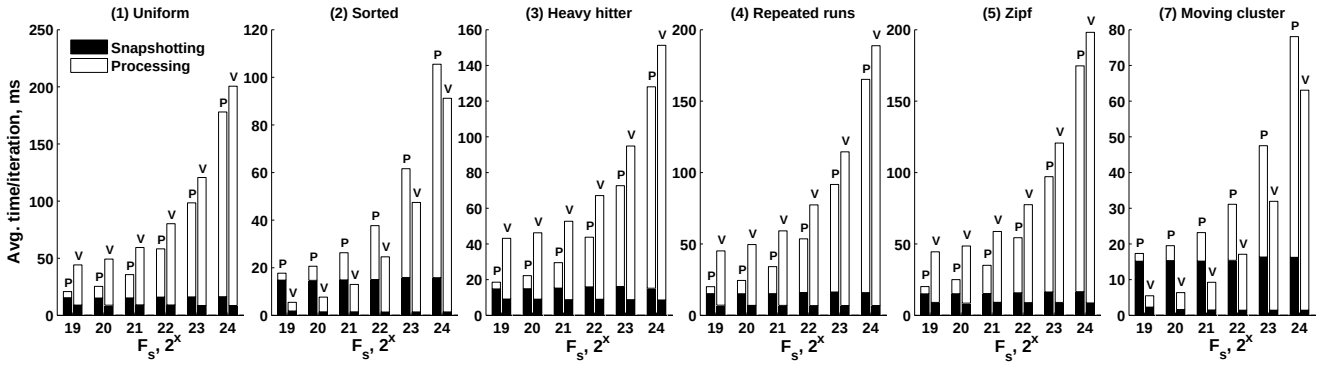


Figure 3: Snapshotting and update processing on Sandy Bridge (snapshot size = 2^{24} items = 128MB).

pages cover 512 × 4 KB = 2 MB, while huge pages cover 32 × 2 MB=64 MB. Figure 2(d) shows the results of the same experiment as in Figure 2(c), but using huge pages. The `fork` cost remains the same, but the bottleneck is indeed deferred by a factor of 32. Note that `memcpy` performance remains the same.

3.2 Virtual vs. Physical Snapshot Updating

To get the full picture of how each of the two techniques compare in terms of update throughput, we consider two more parameters. The first is the snapshotting frequency, F_s, expressed as the number of updates between snapshots. The second is the distribution of updated items. We expect the average cost of highly skewed updates on virtual snapshots to be lower, as the cost of relatively few page copyings are amortized across many updates. In contrast, uniformly distributed updates might end up causing page copying with each update and consequently be very expensive.

To quantify the penalty of the triggered copy-on-write in virtual snapshotting, we measure the average update time to a memory page with and without the CoW flag set. The results are shown in Figure 4. To handle the CoW of a single page takes circa 2μs on the Intel machines, circa 5μs on Barcelona, and almost 64μs on T2. This implies that the processing of a single update can be one to two orders of magnitude faster on a physical snapshot.

To evaluate how snapshot update costs depend on update skew, we apply synthetic distribution generation code that is used in other studies [3, 4, 7, 19] and is based on the work of Gray et al. [10]. The distributions are: (1) uniform, (2) sorted, (3) heavy hitter, (4) repeated runs, (5) Zipf, (6) self-similar[2], and (7) moving cluster. The generator produces the page numbers for each update, while the exact item within the page is chosen randomly. This setting allows to compare update performance based on page update skew and avoids performance hazards such as read interfer-

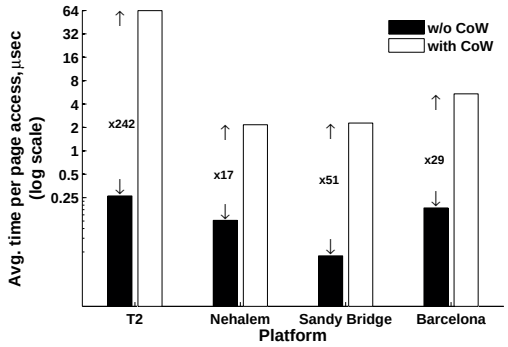


Figure 4: Copy-on-write cost.

[2]We omit results for the self-similar distribution, as they are similar to those of the heavy hitter.

Table 1: Guidelines when one technique is preferred over the other.

	Ease of implementation			Cross-platform	Small snapshots	Huge page support	Update skew (distribution)							Memory footprint
	pointers	non-contig.	queries				1	2	3	4	5	6	7	
Virtual	+	+	−	−	−	+	−	+	−	−	−	−	+	+
Physical	−	−	+	+	+	−	+	−	+	+	+	+	−	−

ence on T2 [15], the convoy problem [15], and parallel output writing [5]. The results for Sandy Bridge are shown in Figure 3. (The other machines are covered in Appendix B.) We fixed the snapshot size to significantly exceed the LLC size (128 MB or 2^{24} items). On the y-axis, we plot the average time per processing cycle (the snapshotting time plus the time for all update processing between two snapshots).

As expected, virtual snapshotting is outperformed by the biggest margin under uniform and repeated runs distributions. However, it is also outperformed under quite skewed distributions such as heavy hitter[3] and Zipf. As F_s increases, the margin tends to decrease. Only under very skewed distributions such as sorted and moving cluster, virtual snapshotting is able to outperform physical snapshotting. For these distributions, the virtual technique spends less time on the snapshotting than the physical one and, as only few pages are updated, the cost of the few CoWs are amortized across many updates.

3.3 Virtual vs. Physical Snapshot Querying

Our experiments confirm that the performance of read operations on virtual and physical snapshots are the same. However, implementing the query support in the virtual snapshotting approach is complicated by the communication and the synchronization between processes needed because queries have to be directed to a thread in the most recently forked process. In physical snapshotting, both snapshots can be accessed by multiple threads within the same process.

Our micro-benchmarks report a communication latency of 10–15μs on the AMD and Intel machines and circa 32μs on T2. Therefore, under virtual snapshotting only if queries are relatively long-running (e.g., at the order of milliseconds), the inter-process communication latency becomes insignificant. Commercial APIs for high-frequency trading achieve latencies for shared memory IPC as low as half a microsecond [16].

3.4 Memory Consumption

Virtual snapshotting is a clear winner in terms of memory consumption. In the best case, it consumes approximately half of the memory required by physical snapshotting. In the worst case, the memory consumption by both techniques is the same. This is similar to existing results [14].

4. CONCLUSIONS

Our findings are summarized in Table 1. When snapshots contain pointers or are not stored contiguously in main memory, virtual snapshotting is easier to implement. On the other hand, querying support is easier to implement on physical snapshots, as no inter-process communication is needed. Due to poor `fork` performance on T2 (and also the lack of support on Windows machines) physical snapshotting is preferable across a wider range of different platforms. With snapshots comparable in size to the last-level cache or smaller, physical snapshot creation is the fastest. With bigger snapshots, virtual snapshot creation becomes several times more efficient, and its performance can be further improved by using huge pages.

Our main finding is that, for most of the considered workloads, the highest overall update performance is achieved using physical snapshotting. This is true not only for small snapshots, but even for relatively large snapshots, an order of magnitude larger than the LLC. Only under very skewed update distributions (sorted and moving cluster) virtual snapshotting is preferable. These findings provide interesting directions for future studies of possibly more advanced and adaptive snapshotting approaches that exploit the efficiency of `memcpy` for on-demand and memory-saving copying.

[3]One half of the updates hit the same page, while the other half is uniformly distributed.

5. ACKNOWLEDGMENTS

This research was supported by grant 09-064218/FTP from the Danish Council for Independent Research | Technology and Production Sciences. We thank Kenneth A. Ross (supported by NSF grant IIS-1049898) for providing access to the experimental hardware.

6. REFERENCES

[1] AMD. *Software Optimization Guide for AMD Family 15h Processors*. 47414, 2012.

[2] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). In *ISCA*, pp. 48–59, 2010.

[3] J. Cieslewicz, W. Mee, and K. A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN*, pp. 43–51, 2009.

[4] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pp. 339–350, 2007.

[5] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pp. 25–34, 2008.

[6] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, pp. 1–10, 2007.

[7] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, pp. 483–494, 2010.

[8] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pp. 189–207, 2009.

[9] U. Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pp. 243–252, 1994.

[11] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pp. 79–87, 2007.

[12] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 248966-025, 2011.

[13] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pp. 195–206, 2011.

[14] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN*, pp. 17–26, 2011.

Table 2: Experimental platforms.

	Sun UltraSPARC-T2	Dual Intel Xeon 5550	Dual AMD Opteron 2350	Intel Core i7-2600K
Architecture	Niagara T2	Nehalem	Barcelona	Sandy Bridge
Operating System	Solaris 10 5/08	Linux 2.6.32-38	Linux 2.6.24-29	Linux 3.0.0-16
Chips×cores×threads	1×8×8	2×4×2	2×4×1	1×4×2
Clock rate (GHz)	1.2	2.67	2.0	3.4
RAM (GB)	32	24	16	8
RAM type	667 MHz DDR2 ECC	1333 MHz DDR3 ECC	667 MHz DDR2 ECC	1333 MHz DDR3 ECC
Memory Channels	8	3×2 (chips)	2×2 (chips)	2
Max Bandwidth (GB/s)	42	32	10.5	21
Cache line size	16 (L1), 64 (L2)	64	64	64
L1 data cache (KB)	8 (core)	32 (core)	64 (core)	32 (core)
L2 (unified) cache (KB)	4096 (chip)	256 (core)	512 (core)	256 (core)
L3 (unified) cache (MB)	none	8 (chip)	2 (chip)	8 (chip)
DTLB L1 (per core)	32 (multiple page sizes)	48 (4 KB pages) 32 (2 MB pages)	32 (4 KB pages) 8 (1 GB apges)	64 (4 KB pages) 32 (2 MB pages)
DTLB L2 (per core)	128 (multiple page sizes)	512 (4 KB pages)	256 (4 KB pages)	512 (4 KB pages)

[15] K. A. Ross. Optimizing read convoys in main-memory query processing. In *DaMoN*, pp. 27–33, 2010.

[16] Solace Systems. *Achieving Nanosecond Latency Between Applications with Shared Memory Messaging*. Whitepaper, 2011.

[17] Sun Microsystems. *OpenSPARC T2 Supplement to the UltraSPARC Architecture*. 950-5556-02, 2007.

[18] D. Šidlauskas, K. A. Ross, C. S. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *SSTD*, pp. 186–204, 2011.

[19] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pp. 1–9, 2011.

APPENDIX

A. EXPERIMENTAL SETTING

We conduct the performance study on four multi-core platforms: a dual quad-core AMD Opteron 2350, a dual quad-core Intel Xeon X5550, a quad-core Intel Core i7-2600K, and an 8-core Sun Niagara 2. The characteristics of these machines are summarized in Table 2.

The same code base, written in C/C++, is used on all platforms. The code base is small, only a few thousand lines, and is compiled with `g++` under maximum optimization and using `pthreads` for parallelism. The `fork` and `memcpy` calls are accessible through the C library function calls, e.g., via the GNU C Library[4]. The memory segments that are known to be untouched by the forked processes are flagged as `SHARED` so that any updates there by the main process do not trigger CoWs behind the scene (avoiding unwanted overhead).

The division of the physical snapshot into t equal sized sections and the assignment of each section to a separate hardware thread are vulnerable to load imbalances. This is especially true for the Intel machines, where the two threads within a core are not utilized in a fair manner. To avoid such load imbalances, we experimented with the techniques of parallel buffers [6] where each thread grabs a chunk of memory and does its processing before grabbing a new chunk. However, with chunk sizes that led to processing without slowdowns, the overall performance was lower. This is because, for example, nice features of spatial locality, favored by prefetching, then apply only to smaller memory regions in each `memcpy` call.

[4]http://www.gnu.org/software/libc.

B. ADDITIONAL EXPERIMENTS

B.1 Update Performance

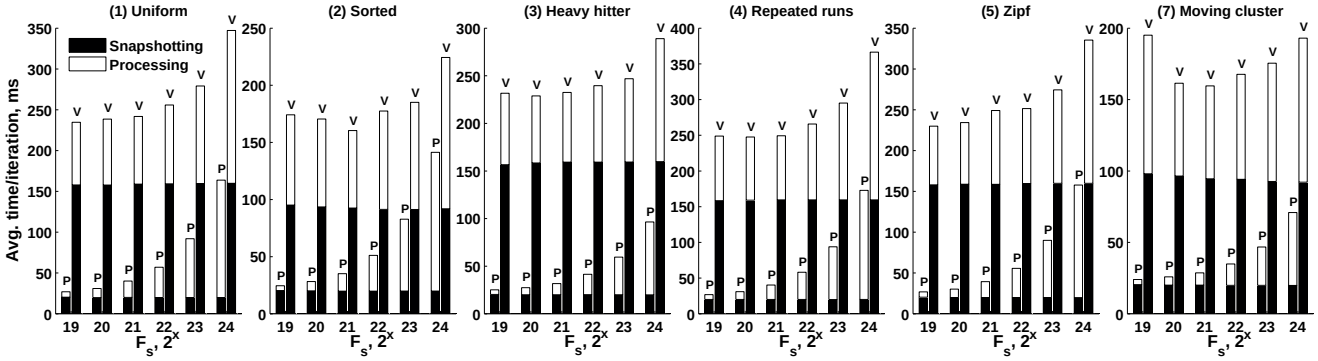
Figure 5 shows virtual and physical snapshot update performance on all our machines. The snapshot size is fixed to 128 MB (or 2^{24} items). On the x-axis, we plot the snapshotting frequency, F_s, expressed as the number of updates between snapshots. On the y-axis, we plot the average time per processing cycle (the snapshotting time plus the time for all update processing between two snapshots). Bars marked with "V" and "P" correspond to virtual and physical snapshotting, respectively.

Figure 5(d) shows the results from the same experiment as in Figure 3, but using huge page sizes. The time spent in snapshotting reduces significantly in virtual snapshotting. Consequently, under very skewed distributions (sorted and moving cluster) virtual snapshotting outperforms physical snapshotting by a bigger margin, while under the other distributions both techniques become very competitive.

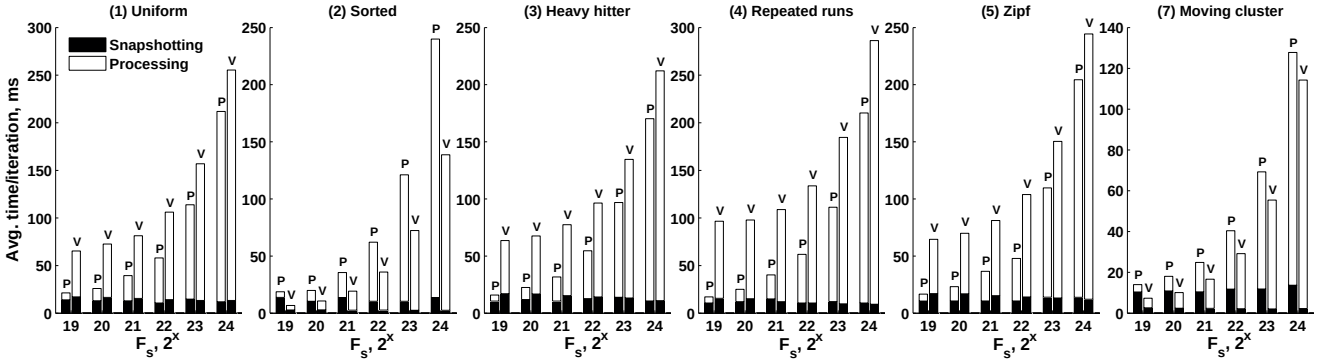
B.2 Poor `fork` Performance on T2

As shown in Figures 2 and 5, virtual snapshotting is always outperformed by physical snapshotting on T2, but not on the other machines considered. We found two possible explanations for that. First, Solaris, as opposed to Linux, does not support memory over-commit. Each time `fork` is called, the kernel checks its free memory lists thoroughly trying to find the requested amount of virtual memory. If found, the kernel reserves the swap space for it such that no other process can use it until the owner releases it. Figure 5(a) confirms this: no matter how many pages are updated between two snapshottings, the `fork` cost in virtual snapshotting is always the same. In Linux, memory allocations always succeed, and the actual checks are made (and actual physical memory frames are allocated) only with the first reference to that page.

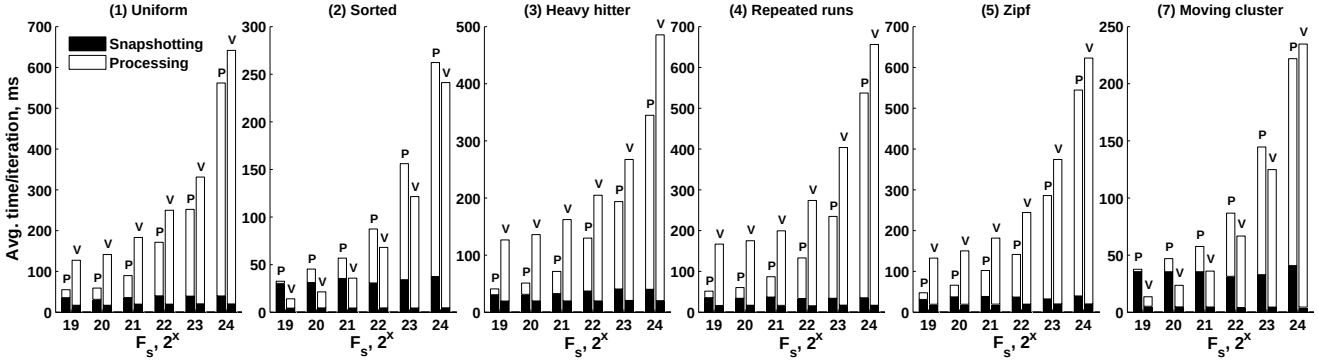
Second, the SPARC architecture has traditionally handled TLB misses in software using a software-managed, direct-mapped cache of translations, called a Translation Storage Buffer (TSB). To speed up look-ups in TSB, UltraSPARC T2 has hardware support for that. Nevertheless, other studies also show that it performs poorly when compared to the Intel and AMD counterparts [2].



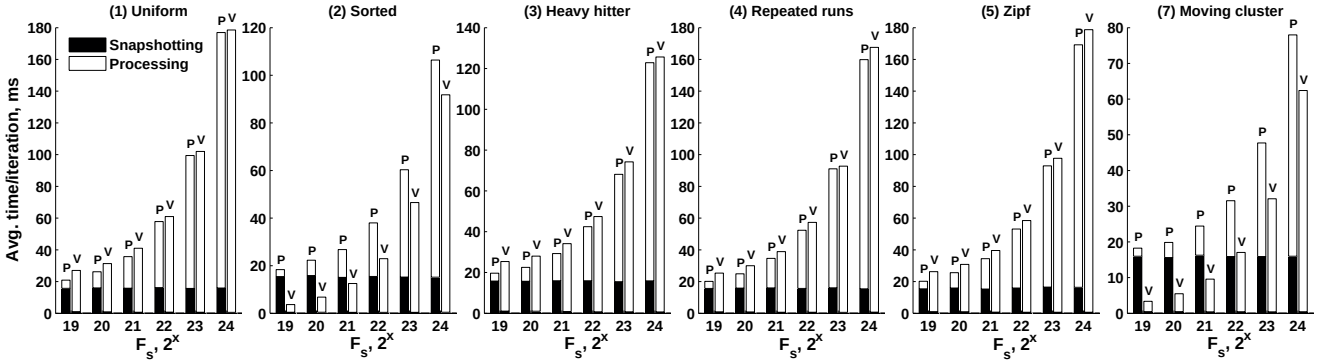
(a) Using 8KB-pages on Sun UltraSPARC-T2 CPU @ 1.2GHz



(b) Using 4KB-pages on dual Intel Xeon X5550 CPU @ 2.67GHz



(c) Using 4KB-pages dual AMD Opteron 2350 CPU @ 2.0GHz



(d) Using 2MB-pages on Intel Core i7-2600K CPU @ 3.40GHz

Figure 5: Snapshotting and update processing (snapshot size = 2^{24} items = 128MB).

Reducing OLTP Instruction Misses With Thread Migration

Islam Atta[†] Pınar Tözün[‡] Anastasia Ailamaki[‡] Andreas Moshovos[†]

[‡]École Polytechnique Fédérale de Lausanne [†]University of Toronto

ABSTRACT

During an instruction miss a processor is unable to fetch instructions. The more frequent instruction misses are the less able a modern processor is to find useful work to do and thus performance suffers. Online transaction processing (OLTP) suffers from high instruction miss rates since the instruction footprint of OLTP transactions does not fit in today's L1-I caches. However, modern many-core chips have ample aggregate L1 cache capacity across multiple cores. Looking at the code paths concurrently executing transactions follow, we observe a high degree of repetition both within and across transactions. This work presents *TMi* a technique that uses thread migration to reduce instruction misses by spreading the footprint of a transaction over multiple L1 caches. *TMi* is a software-transparent, hardware technique; *TMi* requires no code instrumentation, and efficiently utilizes available cache capacity. This work evaluates *TMi*'s potential and shows that it may reduce instruction misses by 51% on average. This work discusses the underlying tradeoffs and challenges, such as an increase in data misses, and points to potential solutions.

1. INTRODUCTION

Online transaction processing (OLTP) workloads are memory bound spending 80% of their time stalling for memory accesses [10]. Most of these stalls are due to first-level instruction cache misses [3]. Previous works tackle this problem in software [6] or hardware [9, 15, 2, 4].

Traditional OLTP systems randomly assign transactions to worker threads, each of which runs on one core of a modern multi-core system. As we also confirm, the instruction footprint of a typical OLTP transaction does not fit into a single L1-I cache, resulting in a high instruction miss rate. Enlarging the L1-I cache is not a viable solution; to avoid impacting the CPU clock frequency, the L1 caches of virtually all high-performance processors today are limited to about 32KB, despite the growing size of L2 and L3 caches. However, this work demonstrates that the footprint of a typical OLTP transaction would comfortably fit in the aggregate L1-I cache capacity of modern many core-chips. There is then an opportunity to reduce instruction misses by spreading the footprint of transactions over multiple L1-I caches provided that there is sufficient code reuse. Fortunately, in OLTP, there is a high-degree of instruction footprint reuse both within a transaction and across multiple, concurrently running transactions [2, 6].

This paper investigates *TMi*, a dynamic hardware solution that uses thread migration to minimize instruction misses for OLTP applications. *TMi*, spreads each transaction over multiple cores, so that each L1-I cache holds part the instruction footprint. *TMi* relies upon: (1) a hardware scheduler knowing the thread types, and (2) having a fast and efficient mechanism to determine whether a set of cache blocks (tags) exist in a given cache. *TMi* exploits intra- and inter-thread footprint reuse as follows: (1) If a thread loops over multiple code parts that are spread over multiple caches, the observed miss rate will be lower since in a conventional cache each part would evict the others resulting in thrashing, (2) The first thread of a particular type effectively prefetches and distributes the common instruction footprint for the rest. TMi differs from past OLTP instruction miss reduction techniques in that it a pure hardware, low-level solution that avoids undesirable instrumentation, utilizes available core and cache capacity resources, and requires no changes to the existing code or software system. TMi is not free of tradeoffs and challenges. Migrating threads takes times and results in more data misses and hence TMi must balance the cost of these overheads against the benefit gained from reducing instruction misses.

This work makes the following contributions:

- It analyzes instruction and data footprints for OLTP workloads (TPC-C and TPC-E [17]) and reaffirms that they suffer from instruction misses; on average 91% of L1 cache capacity misses are instruction misses.
- It demonstrates that the recently proposed replacement policies [14], which can reduce miss rates significantly for some workloads, are not effective for OLTP workloads (less than 6% reduction).
- It identifies the requirements and challenges for an ideal thread-migration-based solution.
- It presents and evaluates TMi, a practical thread migration algorithm. It shows that TMi can reduce instruction misses by 51% on average.
- It identifies the challenges associated with thread migration, namely data misses and migration overhead.

The remaining of this document is organized as follows.

Proceedings of the Eight International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA

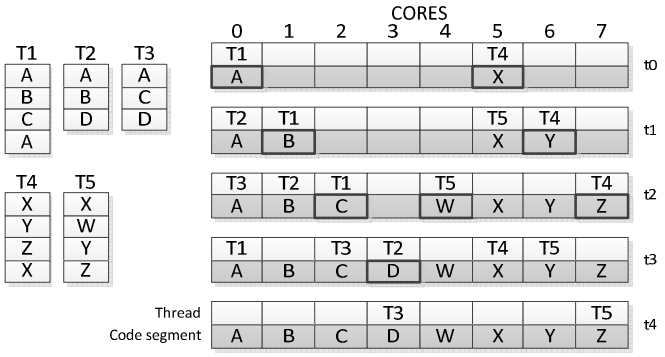


Figure 1: Example to illustrate thread migration and reuse of common code segments. Left: T1-3 and T4-5 are threads running similar transactions. Right: 8-core system. The shaded area is the cache activity (thick border = warm-up phase).

Section 2 analyzes the nature of the problem and the requirements for an ideal solution. Section 3 describes the proposed automated thread migration algorithm, TMi. In Section 4 we present the evaluation results. Section 5 surveys the related work. Section 6 discusses challenges and practical considerations. Finally, Section 7 concludes.

2. PROBLEM & OPPORTUNITY

In typical multi-threaded OLTP systems, a single thread handles a single transaction and multiple threads executing the same transaction type usually run concurrently. Individual threads, with memory footprints that do not fit in the L1-I cache, suffer from high miss rates. For the examined OLTP workloads, Ferdman *et. al.* [3] show that memory stalls account for 80% of the execution time. Section 4 shows that instruction capacity misses account for 83% of total L1 cache misses.

Although similar transactions (threads) vary in their control flow, they have common code segments, leading to 80% redundancy in L1-I caches across multiple cores [2]. We exploit the availability of many cores and multiple concurrent threads with inherent code commonality. Using thread migration, we aim to increase the reuse of instruction blocks brought in the cache. We propose a hardware scheduling algorithm that divides and distributes the instruction code footprint across multiple cores. The algorithm dynamically pipelines and migrates threads to cores that are predicted to hold the code blocks to be accessed next.

Figure 1 demonstrates thread migration using an example. On the left, there are five threads T1-T5 running on 8-cores, where T1-T3 and T4-T5 execute similar transactions, with slightly different instruction streams. The instruction footprint of T1 is 3× larger than L1-I cache size. It executes the following code segments in order: A-B-C-A, where each segment fits in the L1-I cache. T2 and T3 are of the same type as T1, hence they share common code segments with T1, but they are not identical. A typical system would assign T1, T2, and T3 to separate cores, and since their footprints do not fit in the L1-I cache, each individual thread would suffer instruction misses.

The ideal scenario for thread migration would be as follows: initially (at time t0), T1 starts execution on core-0. When all cache blocks for A are brought in the cache, T1 migrates to core-1 (at t1), where it continues executing filling the cache with blocks from B. T2 is then scheduled to start execution on core-0 (at t1), ideally observing hits for all blocks in A. At a higher level, the process continues and T1 warms-up caches 0, 1, and 2 with A, B, and C, respectively. If T2 and T3 where to follow the same path as T1 then they would experience no misses.

The threads do not need to follow identical paths for migration to be beneficial. Code segment D illustrates this. Since code segment D, accessed by T2, was not part of T1's instruction stream, T2 needs to warm-up core-3 and suffer some extra misses. Provided that T3 executes after T2 finishes filling the cache with D, T3 does not suffer any instruction misses. At t3, when T1 goes back to A, it gets rescheduled to core-0. T4-T5 exhibit similar behavior, but they get assigned to a different set of cores, avoiding conflict with T1-T3. The same process applies for all threads that arrive later: if part (or all) of their code segments exist in the L1-I cache, they migrate to the appropriate core and do not miss on these segments.

We observe that transactions vary in their control flow, hence we should not impose any specific pipelining; i.e., we do not restrict similar threads to follow the exact same path. Thread migration should dynamically detect: (a) *When* a thread should migrate (i.e., when is the cache *full*)? (b) *Where* to migrate (i.e., which cache holds the code segment we start to touch, if any)? Thus we need to maintain information about threads and caches to be able to make judicious migration decisions.

3. AUTOMATED THREAD MIGRATION

With the requirements we set in Section 2 in mind, here we detail our thread migration algorithm, TMi, and its challenges. Our main focus in this paper is the potential of thread migration in reducing instruction misses for OLTP. Hence, the naïve TMi as described here might seem inefficient. However, we discuss why we believe a practical implementation is possible in Section 3.2 and Section 6.

3.1 Migration Algorithm

The presented TMi design assumes it knows the *type* of each transaction so that it can group same type threads into teams. We discuss possible methods to detect transaction types in Section 6. TMi schedules each team without preemption. Threads within a team are independently migrated among cores without any central control. Migration decisions are made based on a set of rules and inputs.

Initially, the system operates normally and no migration takes place. First, TMi must detect when the L1-I cache of a given core becomes *FULL*, that is filled with enough instructions so it is better for the thread to migrate to another core to avoid thrashing. To detect a *FULL* cache, TMi uses a resettable miss counter (MC) and a full miss-threshold (FMT). When MC exceeds FMT, TMi enables migration.

Migration does not happen immediately after a cache becomes *FULL*. The thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path.

When migration is enabled, we need a mechanism to select a good target cache to migrate to. Ideally, we would migrate to a cache that has the instructions that this thread will execute next. TMi records recently missed tags in a FIFO

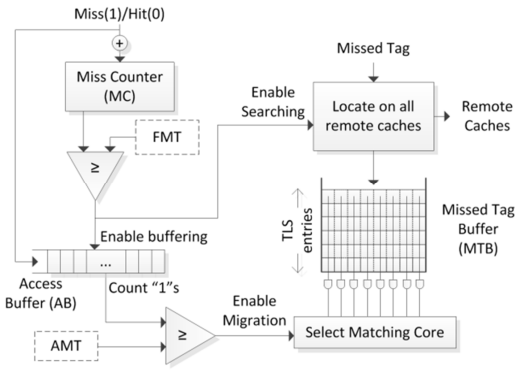


Figure 2: TMi organization.

missed-tags buffer (*MTB*). We refer to the size of *MTB* as the tag list size (*TLS*). For a given thread on a given core, if the *MTB* is full, TMi searches all remote L1-I caches for the missed tags, and makes one of the following decisions:

- If there is a match in a remote cache for *TLS* tags, TMi migrates the thread to the matched core.
- Else, the thread is migrated to an idle core (if any).
- Else, do not migrate.

An additional condition is required to restrict migration to the cases when a thread starts to miss more frequently. In a sub-optimal scenario, threads have to miss for a few tags before migrating (*TLS* tags must be located on a remote cache). This can lead to eviction of useful cache blocks that were fetched in the warm-up phase, creating gaps in the instruction stream. TMi mitigates this by not allowing threads to migrate for occasional misses that fill-up these gaps. Access buffer (*AB*) is a 100-bit FIFO queue recording the history of the last 100 cache accesses (enabled when cache is *FULL*). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reach a threshold (*AMT*), TMi enables migration. If *AMT = 0*, then TMi neglects this condition.

TMi resets *MTB* and *AB* with every migration. When a team of threads completes execution, TMi resets all *MC*s, *MTB*s and *AB*s.

Figure 2 describes the thread migration decision process. TMi components are: (a) Miss counter *MC*. (b) Access buffer *AB* is a 100-bit FIFO. (c) Missed Tag Buffer *MTB* is a FIFO of n-bit entries, where n is the number of cores. (d) Full miss threshold *FMT* determines when a cache is *FULL*. (e) Access miss-threshold *AMT* is the minimum number of misses in a window of 100 accesses to enable migration. *MC*, *AB* and *MTB* are resettable.

3.2 Migration Cost & Challenges

Tag Search. We believe it is possible to avoid individual tag searches (e.g., keeping track of signatures as opposed to a complete tag list) and searching all cores (e.g., serial node searching, history-based candidate node prediction).

Data Misses. Migrating across cores may increase the data miss rate; data that might be reused is left behind during a migration. Experiments show that the data miss rates increase significantly.

Existing architectures simply use caches to reduce instruction and data misses. No additional effort is made to balance the relative cost of instruction vs. data misses. A relevant question is whether the relative costs are appropriately balanced in modern systems. TMi facilitates balancing the relative costs of the two types of misses. There are reasons to believe that instruction misses, up to a point, impact performance more than data misses. For example, modern architectures use instruction level parallelism to hide the impact of data misses. Without instructions these ILP techniques remain ineffective. However, this has to be proven through experimental evaluation, and is left for future work.

Simultaneous Multithreading (SMT) is another way for reducing instruction stalls but it requires extra resources. Additionally, if threads are not synchronized, it increases pressure on instruction caches leading to more thrashing.

We analyze data misses and discuss potential solutions to mitigate their impact in Section 4.5.

Migration Overhead. Fast context switching is attainable in software [6] and we argue that the same is possible for the hardware. Estimating hardware migration overhead is out of the scope of this paper, but in Section 4 we seek to increase the number of instructions between migrations, thus reducing the relative importance of the context switching cost.

Overhead of data-structures used. All data structures described here are per core. *MC* exists in the Performance Monitoring Unit (PMU) of most modern processors. *MTB* is a list of *TLS* entries and we consider two implementations: In the first, an entry is simply a tag. A more compact and faster list consists of n-bits per entry, where n is the number of cores. A logic-1 bit at index i, of tag j, means that the cache of core-i holds a valid copy of tag j. By ANDing all bits mapped to core-i, we know whether it holds all tags, or not. This mechanism relies on the coherence protocol to identify shared cache blocks, and makes faster migration decisions. Figure 2 shows the latter implementation. *AB* is a 100-bit FIFO queue used to record the history of cache accesses. A *thread queue* holds a 12-bit identifier and context state. We assume virtualizing this information on lower cache levels [1]. Note that all logic operations for TMi are not on the critical path.

4. EVALUATION

Current real and full system simulation platforms do not support thread migration at the hardware level. The OS kernel assumes full control over thread assignment in multicore environments. To work around this limitation, we use trace simulation. We use PIN [12], an instrumentation tool that is able to inject itself in x86 binaries, to extract instruction and data access traces. Although PIN instruments only application level code, our proposal is generic and applies to system level code, too. Previous work shows that thread migration is beneficial for system level code [2].

The baseline configuration *NO-MGRT* assumes no thread migration and assigns threads to cores randomly. *TMi-ST* is the proposed thread migration algorithm, which groups *similar threads* into teams. We examine various thread migration parameters. Unless otherwise indicated, we simulate a 16-core system with 32KB 8-way set-associative instruction and data caches with 64B blocks, and LRU replacements.

We run TPC-C and TPC-E on top of a scalable open-source storage manager; Shore-MT [8]. Traces are extracted,

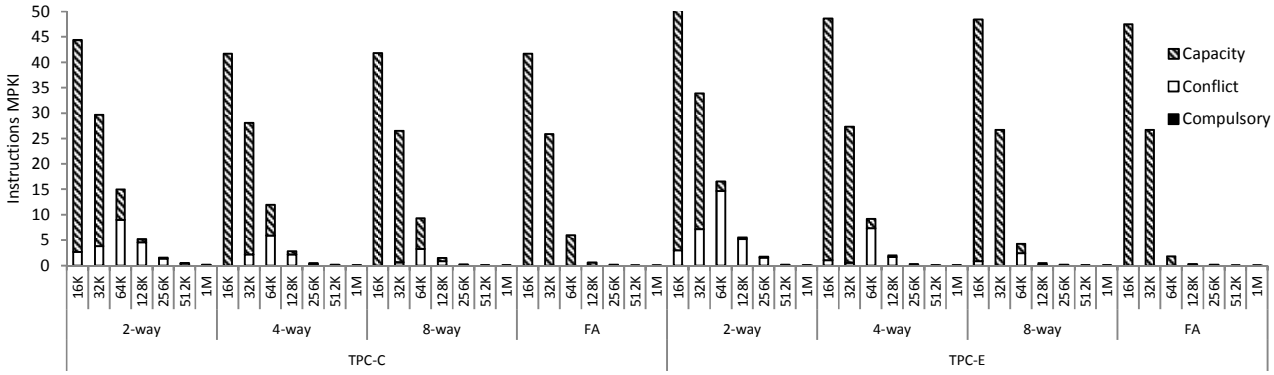


Figure 3: Breakdown of MPKI for different L1-I sizes and associativity (lower is better).

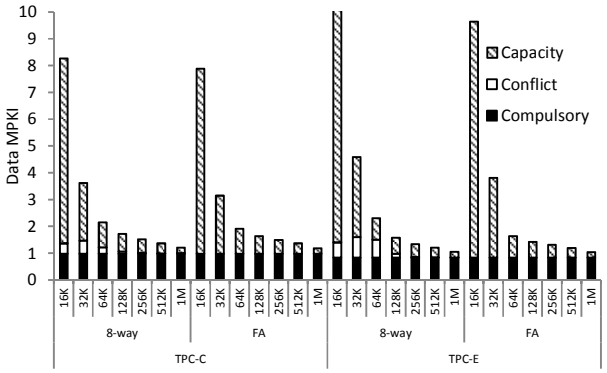


Figure 4: Breakdown of MPKI for different L1-D sizes and associativity (lower is better).

per transaction type, to form a mix of 1K threads, per benchmark. Teams of 20-30 similar threads are scheduled without preemption (threads are more than cores to consider thrashing). We use misses per kilo instructions (MPKI) as our metric for instruction (I-MPKI) and data (D-MPKI) misses. We use kilo instructions per migration (KIPM) as a migration overhead metric. Ignoring secondary effects, the higher the KIPM the less fraction of time is spent on migration. An ideal solution should minimize MPKI and maximize KIPM.

In our evaluation we address the following questions: (1) What are the L1 miss characteristics for OLTP workloads (Sections 4.1 and 4.2)? (2) What are good configuration parameters for TMi (Sections 4.3 and 4.4)? (3) How does TMi affect data misses (Section 4.5)? (4) What is the impact of a migration algorithm, which is unaware of thread types, on L1 misses (Section 4.6)?

4.1 Instruction and Data Footprints

This section examines the instruction and data footprints through an analysis of misses per kilo-instructions (MPKI) on full-associative (FA) and set-associative (SA) L1 caches. Hill and Smith [7] categorize cache misses into three *C*s: *Capacity* misses result from limited cache capacity (relatively smaller caches may increase capacity misses), *Conflict* misses are those resulting from reduced associativity (smaller sets may increase conflict misses), and *Compulsory* misses are for the first reference to each unique data block (you can avoid if you prefetch). By identifying where most misses come from, we highlight the reasons behind the memory stalls; is it cache size, associativity, or bad locality?

Figures 4 and 3 has the breakdown of MPKI into the three *C*s. Figure 3 shows that for FA caches (no conflict misses), the instructions fit in 128KB – 256KB caches. A 512KB cache incurs only compulsory misses (i.e., perfect cache). Today, most modern microprocessors use 32KB L1s. Thus, the full footprint can fit in 4 – 8 caches (or more).

For 32KB caches, instruction capacity misses are $9-12\times$ more than data capacity misses, while data compulsory misses are two orders of magnitude larger than instruction compulsory misses. We conclude that OLTP transactions have large instruction footprints with much lower locality than their data footprint. The instruction streams exhibit a recurring pattern (lots of reuse), which has a relatively long period, leading to eviction of useful blocks that are re-accessed.

These observations support TMi; it could avoid lots of instruction misses by virtually increasing the L1-I cache size per thread. In the rest of our analysis we use 32KB 8-way SA L1 caches. We do so as they typical in modern processors, e.g., the Intel ®Core ™2 Quad.

4.2 Replacement Policies

Qureshi *et al.* identify that not all workloads are LRU-friendly [14]. They propose static (LIP, BIP) and dynamic (DIP) insertion policies that are effective in reducing miss rates significantly for some workloads. We measure the impact of these policies on OLTP workloads and compare them against LRU. In Figure 5, we witness less than 6% improvement for DIP on the instruction MPKI for the baseline setup. Nevertheless, these policies are orthogonal to TMi.

4.3 Tuning Migration Parameters

This section explores the effect of *FMT* and *TLS* on instruction misses (I-MPKI) and instruction count between migrations (KIPM). As defined in Section 3.1, *FMT* sets the initial warm-up for an L1-I cache. When the miss counter (*MC*) is lower than *FMT*, a thread is not allowed to migrate. *TLS* sets the minimal number of tags a remote cache should hold before a thread migrates to it. Larger *TLS* limits migration, while smaller *TLS* triggers too frequent migration; we are optimizing values for *FMT* and *TLS* to reduce I-MPKI and increase KIPM.

Figure 6 shows the results for TMi-ST. We examine *FMT* values of 128 – 1024 and *TLS* values of 2, 4, and 6. We report relative I-MPKI with respect to the baseline. Larger

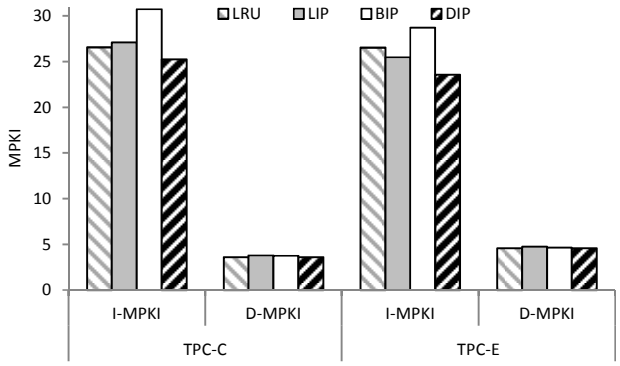


Figure 5: Effect of different replacement policies for OLTP workloads.

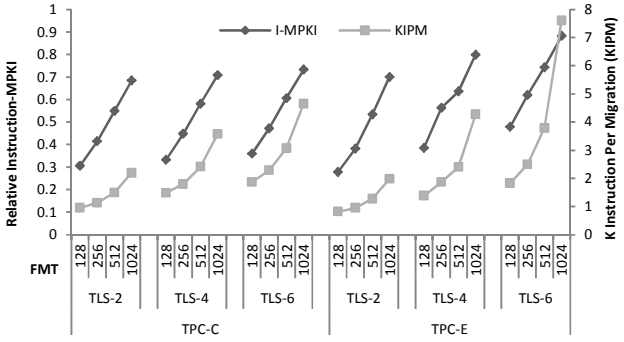


Figure 6: Exploring the parameter space of *FMT* and *TLS* values

TLS translates into fewer migrations, resulting in higher I-MPKI for all values of *FMT*, while smaller *TLS* reduces KIPM. TPC-E is more sensitive to larger *TLS* values than TPC-C (TPC-E has more non-uniform instruction streams). For *TLS* values larger than 6, TPC-E shows very few migrations with negligible I-MPKI improvement. *FMT* favors smaller values, assigning each cache a smaller working set and reducing conflicts. For 64B block size (512 blocks per cache) and *FMT* values larger than 512, cache blocks are definitely evicted in the warm-up phase.

To summarize, lower *FMT* and *TLS* reduce I-MPKI by 70%/73% for TPC-C/TPC-E, with slightly less than 1 KIPM. We conclude that a good combination would be 256/6 for *FMT/TLS*, resulting in 53%/38% I-MPKI reduction, and 2.3/2.5 KIPM, for TPC-C/TPC-E. The remaining of the evaluation section uses these values.

4.4 Access Miss Threshold

AMT restricts migration to the cases when a thread starts to miss more frequently. In a sub-optimal scenario, threads have to miss for a few tags before migrating (*TLS* tags must be located on a remote cache). This can lead to eviction of useful cache blocks that were fetched in the warm-up phase, creating gaps in the instruction stream. *AMT* mitigates this by not allowing threads to migrate for occasional misses that fill-up these gaps.

Figure 7 shows I-MPKI and KIPM for TMi-ST, with a range of *AMT* values (1 – 12). Using small *AMT* triggers more frequent migrations. Using too large *AMT* increases KIPM by reducing migrations, but with a possible I-MPKI

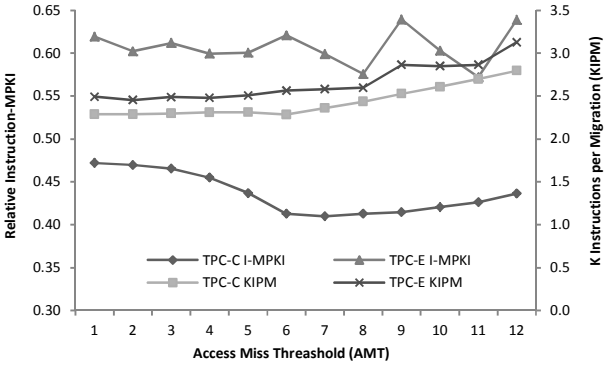


Figure 7: Exploring the parameter space of *AMT*

increase since it results in more evictions. For TPC-C, *AMT* values 6 to 9 are better. TPC-E shows a non-uniform behavior for *AMT* values 9 (high I-MPKI) and 11 (low I-MPKI). This behavior could be a result of an irregularity in the instruction stream. We have noticed a direct correlation between *AMT* and *TLS* values: the point $TLS = AMT$ is a sweet-spot for I-MPKI and for larger *AMT*, KIPM increases linearly. Due to limited space, we do not show graphs for different *TLS* values. For the remainder of the evaluation section, we set *AMT* to eight (59%/43% I-MPKI reduction and 2.4/2.6 KIPM for TPC-C/TPC-E).

4.5 Data Misses

While reducing instruction misses, TMi increases data misses in three ways; (1) a thread may access data that it fetched on one core, when it migrates to another core, (2) when a thread returns to a core, it may find that data that it originally fetched has since been evicted by another thread, (3) data writes of T1 on core-B to blocks fetched on core-A lead to invalidations that would not have occurred without migration. Although we believe that instruction misses are more expensive than data misses (performance-wise), we have to account for D-MPKI.

Figure 8 shows that TMi-ST incurs an average increase in D-MPKI of 4× over the baseline. We notice that the increase in D-MPKI is the result of writes (WR D-MPKI), while reads are nearly unaffected. The average total MPKI is reduced by 4% over the baseline. For an inclusive cache hierarchy, all data misses can be served on-chip via lower levels of cache (L2/L3), allowing out-of-order execution to partially hide it. In addition, we believe that data misses could be mitigated by techniques like prefetching, especially when we have a strong clue on what to prefetch.

4.6 Blind Migration

TMi-ST assumes knowledge of thread types. We experiment with an algorithm (TMi-B) that migrates threads without considering their types. TMi-B utilizes an aggressive scheduler (1K concurrent non-similar threads), while TMi-ST forms teams of similar threads, where each team is scheduled without preemption. Figure 8 shows I-MPKI/D-MPKI for TMi-B. We notice a 31% average reduction in I-MPKI over the baseline, and an average increase of 40% over TMi-ST. Read D-MPKI increases by 2× on average, while write D-MPKI is slightly reduced. Closer inspection shows that blind migration creates large access periods be-

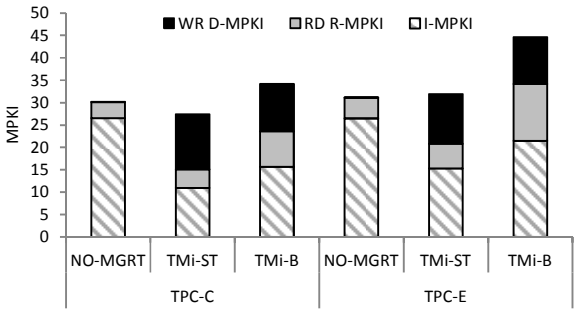


Figure 8: I-MPKI decrease vs. D-MPKI increase

fore a thread migrates back to a previous core, finding that most of its cache blocks are evicted. Conflicts occur when non-similar threads are assigned to the same core. We define *distance between similar threads* (DBST), as a metric to measure the conflict degree (lower is better). DBST is the distance between two similar threads executing on the same core. TMi-B and TMi-ST show an average DPST of 28.3 and 2.53, respectively. We conclude that identifying thread types is necessary for TMi to effectively reduce in I-MPKI.

5. RELATED WORK

STEPS [6] is a software solution that increases instruction reuse by grouping threads into teams of similar transactions. It breaks transaction instruction footprints into smaller cache chunks. Then instead of running a full transaction, it runs all transactions for the first chunk, then for the second and so on. STEPS identifies points in the code where context-switches should occur either manually or by using a profiling tool, which makes it hard to use in practice. TMi is similar to STEPS in the sense that we promote locality, but instead of context-switching on a single core, we migrate threads to other cores, utilizing available on-chip resources and reducing thread queuing delay. In addition, TMi does not require non-trivial manual intervention or instrumentation to detect synchronization points.

On the hardware side, instruction prefetching solutions have evolved from simple low accuracy stream buffers [9, 15] to highly accurate sophisticated stream predictors [5, 4]. Accurate prefetchers are expensive requiring ∼40KB of extra storage per L1 cache, which increases with the instruction footprint. In addition, they neglect the possible presence of idle cores, and do not avoid code and prediction redundancy underutilizing on-chip resources.

Chakraborty *et al.* [2] report high redundancy in instruction fragments across multiple threads executing on multiple cores. They propose CSP, which employs thread migration as a method to distribute the dissimilar fragments of the instructions executed by a thread and group the similar ones together. CSP is limited to separating application level (user) code from OS code, losing opportunities of separation within user code. They point to profile-driven annotation or high-level application directives to identify separation points within user code. TMi generalizes thread migration to include interleaved user-OS code separation points.

Some other recent thread migration proposals target power management, data cache, or memory coherence [13, 11, 16].

6. DISCUSSION

In this section we discuss some of practical implementation aspects and point out directions for future work.

Scalability. Due to limited space, we do not show a range of evaluation setups. We have seen that TMi is scalable on several dimensions: number of cores (4 – 64) and threads (100 – 1K), L1-I cache size (16K – 64K, optimized FMT), and instruction footprint sizes (different transaction types).

Workload Dependency. Section 4 shows that TPC-C and TPC-E favor different configuration parameters calling for a dynamic solution. Therefore, as a possible alternative to our algorithm, we might rely on feedback from migration and miss counters.

Impact of OS. Current OS kernels assume full control over thread assignment to cores. To support fast hardware thread migration, thread assignment should be transparent to higher layers, to avoid any software overhead. Otherwise, the OS scheduler, that needs to know where each thread is running, must be informed about these migrations. A different approach might rely on hardware mechanisms to provide counters and migration acceleration, and leave policy choice to software (enabling easier integration with existing schedulers, more flexible policies, etc.).

Thread Identification. A thread migration algorithm that can identify similar threads (TMi-ST) proved more effective than a blind algorithm (TMi-B). In practice, there are two approaches to identify similar threads: (1) relying on the application to transfer this knowledge to the hardware; requires undesirable modifications to the application, and (2) expecting hardware to detect the threads accessing common code segments and tag them as similar threads. We plan to explore these options.

7. CONCLUSIONS

OLTP workloads spend 80% of their time on memory stalls; L1 instruction misses, and L2 data misses. We corroborate these results and show that 91% of L1 capacity misses are for instructions. Additionally, we show that recently proposed replacement policies, which reduce miss rates for some workloads, are ineffective on OLTP workloads. Previous hardware or software proposals are either impractical (require code instrumentation) or relatively expensive (large on-chip data structures).

This work presented a solution based on thread migration, TMi. Similar to CSP [2] and STEPS [6], we exploit the code commonality observed across multiple concurrent threads. Unlike CSP, we do not limit code reuse to OS code segments, and extend that to application code. Unlike STEPS, instead of context switching, we distribute the instruction footprint across multiple cores and migrate execution. We identify the requirements for an ideal system, and implement TMi, an algorithm to evaluate the solution's potential. It is a low-level hardware algorithm which requires no code instrumentation and efficiently utilizes available cache capacity.

TMi was able to reduce the instruction misses by 51% on average. As expected, it impacted the data misses, increasing the total L1-D misses to 95% of the baseline. On an out-of-order pipeline, instruction misses are believed to be more expensive (performance-wise), lending potential to TMi. We believe that data misses could be mitigated by techniques like prefetching, especially when we have a strong clue on what to prefetch.

We identify the requirements for a full practical solution as follows: (a) the ability to identify similar threads, (b) prefetching data for migrating threads, and (c) using accurate cache signatures to locate cache blocks in remote cores. Our next steps will address these requirements and study the timing behavior of TMi.

8. ACKNOWLEDGMENTS

We thank Ioana Burcea and Jason Zebchuk on their invaluable feedback, the members of the AENAO and DIAS laboratories for their support and discussions, Stavros Harizopoulos for his help in the early days of this work, and the reviewers and Ippokratis Pandis for their constructive comments on the paper. This work was partially supported by an NSERC Discovery grant, NSERC CRD with IBM, a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

9. REFERENCES

[1] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *ASPLOS*, 2008.

[2] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. In *ASPLOS*, 2006.

[3] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, 2012.

[4] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO*, 2011.

[5] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *MICRO*, 2008.

[6] S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in OLTP. In *TODS*, 2006.

[7] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, Dec. 1989.

[8] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[9] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[10] K. Keeton, D. Patterson, Y. Q. He, R. Raphael, and W. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *ISCA*, 1998.

[11] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas. Directoryless shared memory coherence using execution migration. In *PDCS*, 2012.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[13] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, 2004.

[14] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[15] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS*, 1998.

[16] K. S. Shim, M. Lis, O. Khan, and S. Devadas. Judicious thread migration when accessing distributed shared caches. In *CAOS*, 2012.

[17] TPC. TPC transcation processing performance council. http://www.tpc.org.

KISS-Tree: Smart Latch-Free In-Memory Indexing on Modern Architectures

Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Growing main memory capacities and an increasing number of hardware threads in modern server systems led to fundamental changes in database architectures. Most importantly, query processing is nowadays performed on data that is often completely stored in main memory. Despite of a high main memory scan performance, index structures are still important components, but they have to be designed from scratch to cope with the specific characteristics of main memory and to exploit the high degree of parallelism. Current research mainly focused on adapting block-optimized B+-Trees, but these data structures were designed for secondary memory and involve comprehensive structural maintenance for updates.

In this paper, we present the *KISS-Tree*, a latch-free in-memory index that is optimized for a minimum number of memory accesses and a high number of concurrent updates. More specifically, we aim for the same performance as modern hash-based algorithms but keeping the order-preserving nature of trees. We achieve this by using a prefix tree that incorporates virtual memory management functionality and compression schemes. In our experiments, we evaluate the *KISS-Tree* on different workloads and hardware platforms and compare the results to existing in-memory indexes. The *KISS-Tree* offers the highest reported read performance on current architectures, a balanced read/write performance, and has a low memory footprint.

1. INTRODUCTION

Databases heavily leverage indexes to decrease access times for locating single records in large relations. On classic disk-based database systems, the B+-Tree [2] is typically the structure of choice. B+-Trees are suited for kinds of those systems because they are optimized for block-based disk accesses. However, modern server hardware is equipped with high capacities of main memory. Therefore, database architectures move from classic disk-based systems towards databases that keep the entire data pool (i.e., relations and indexes) in-memory and use secondary memory (e.g., disks) only for persistence. While disk block accesses constituted the bottleneck for disk-based systems, modern in-memory databases shift the memory hierarchy closer to the CPU and face the "memory wall" [13] as new bottleneck. Thus, they have to care for CPU data caches, TLBs, main memory accesses and access patterns. This essential change also affects indexes and forces us to design new index structures that are now optimized for these new design goals. Moreover, the movement from disks to main memory dramatically increased data access bandwidth and reduced latency. In combination with the increasing number of cores on modern hardware, parallel processing of operations on index structures imposes a new challenges for us, because the overhead of latches became a critical issue. Therefore, building future index structures in a latch-free way is essential for scalability on modern and future systems.

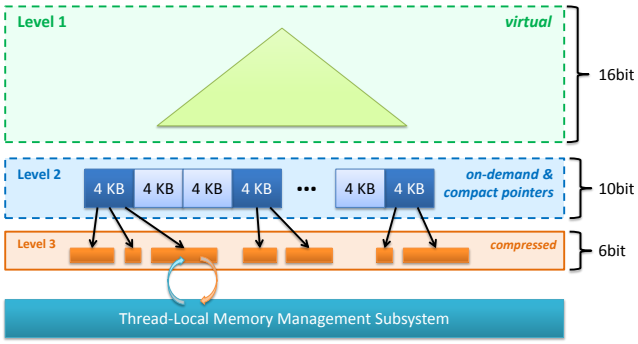
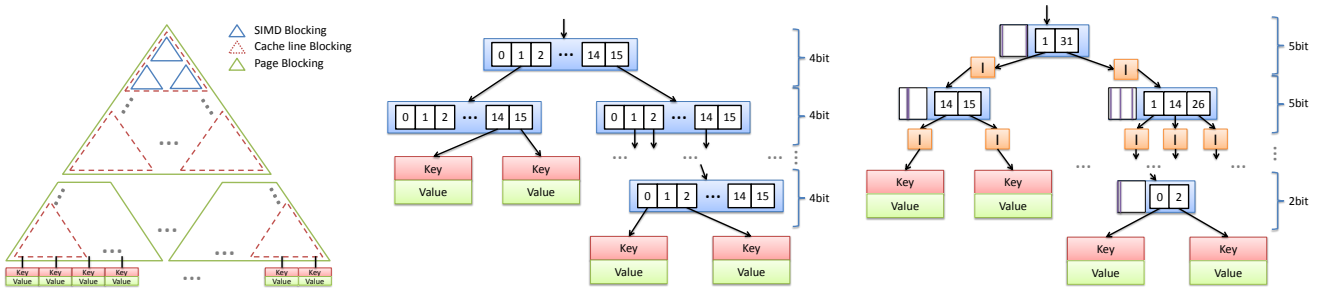


Figure 1: KISS-Tree Overview.

In this paper, we present the *KISS-Tree* [1] that is an order-preserving latch-free in-memory index structure for currently maximum 32bit wide keys and arbitrary values. It is up to 50% faster compared to the previously reported read performance on the same architecture[1] and exceeds its update performance by orders of magnitude. The *KISS-Tree* is based on the *Generalized Prefix Tree* [3] and advances it by minimizing the number of memory accesses needed for accessing a key's value. We achieve this reduction by taking advantage of the virtual memory management functionality provided by the operating system and the underlying hardware as well as compression mechanisms. Figure 1 shows an overview of the *KISS-Tree* structure. As shown, the entire

[1]Due to the unavailability of source code and binaries, we refer to the figures published in [5].

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012) , May 21, 2012, Scottsdale, AZ, USA



(a) FAST. (b) Prefix Tree ($k' = 4$). (c) CTrie.

Figure 2: Existing In-Memory Index Structures.

index structure comprises only three tree levels. Each level leverages different techniques to find a trade-off between performance and memory consumption on the respective level.

2. RELATED WORK

In-memory indexing is a well investigated topic in database research since years. Early research mainly focused on reducing the memory footprint of traditional data structures and making the data structures cache-conscious. For example, the T-Tree [8] reduces the number of pointers of traditional AVL-trees [6] while the CSB+-Tree [11] is an almost pointer-free and thus cache-conscious variant of the traditional B+-Tree [2]. These data structures usually offer a good read performance. However, they struggle with problems when updates are performed by multiple threads in parallel. All block-based data structures require comprehensive balancing operations when keys are removed or added. This implicates complex latching schemes [7] that cause an operation to latch across multiple index nodes and hence blocks other read/write operations even if they work on different keys. Even if there is no node splitting/merging required, B+-Tree structures have to latch coarse-grained at page-level, which increases contention. Moreover, it is not possible to apply optimistic latch-free mechanisms, for instance atomic instructions, without the help of hardware transactional memory(HTM). To overcome this critical point, PALM [12] took a synchronous processing approach on the B+-Tree. Instead of processing single operations directly, the operations are buffered and each thread processes the operations for its assigned portion of the tree. This approach eliminates the need for any latches on tree elements, but still suffers from the high structural maintenance overhead of the B+-Tree.

The FAST approach [5] gears even more in the direction of fast reads and slow updates. FAST uses an implicit data layout, as shown in Figure 2(a), which gets along without any pointers and is optimized for the memory hierarchy of todays CPUs. The search itself is performed using a highly optimized variant of binary search. The fastest reported numbers for read performance on modern CPUs were published for this approach. However, FAST achieves only a very low update performance because each update requires a rebuild of the entire data structure.

The Generalized Prefix Tree [3] takes a different approach than the B+-Tree. Here, the path inside the tree does not depend on the other keys present in the index, it is only determined by the key itself. Each node of the tree consists of a fixed number of pointers; the key is split into fragments of an equal length k' where each fragment selects one pointer of a certain node. Starting from the most significant fragment, the path through the nodes is then given by following the pointers that are selected by the fragments, i.e., the i-th fragment selects the pointer within the node on level i. Figure 2(b) shows an exemplary prefix tree with $k' = 4$. The first tree level differentiates the first 4 bits of the key, the second level the next 4 bits, and so on. Prefix trees do not have to care for cache line sizes because there is only one memory access per tree level necessary. Also the number of memory accesses is limited by the length of the key, for instance, a 32bit key involves a maximum of 8 memory accesses. In order to reduce the overall tree size and the memory accesses, a prefix tree allows *dynamic expansion* and only unrolls tree levels as far as needed like shown on the left hand side of the example. Because of the *dynamic expansion*, the prefix tree has to store the original key besides the value in the content node at the end of the path. The characteristics of the prefix tree allow a balanced read/update performance and parallel operations, because there is not much structural maintenance necessary.

A weak point of the prefix tree is the high memory overhead that originates from sparely occupied nodes. The CTrie [10] removes this overhead by compressing each node. This is achieved by adding a 32bit bitmap (limits the k' to 5) to the beginning of each node, which indicates the occupied buckets in that node. Using this bitmap, the node only has to store the buckets in use. Due to the compression, nodes grow and shrink. Thus, the CTrie raises additional costs for updates, because growing and shrinking nodes requires data copying. To allow efficient parallel operations on the CTrie, it was designed latch-free. Synchronization is done via atomic compare-and-swap instructions, like it is possible in the basic prefix tree. However, because nodes and buckets inside nodes are moving as a result of the compression, the CTrie uses *I-nodes* for indirection as depicted in Figure 2(c). This indirection creates a latch-free index structure, but doubles the number of memory accesses per key.

Another approach was introduced by Burst Tries [4]. Burst Tries utilize existing index structures and combine them to a heterogeneous index. The index starts on the topmost levels as a Prefix Tree and changes over to another structure like the B+-Tree. This combination lets the Burst Trie take advantage from the individual characteristics of the base structures.

3. KISS-TREE STRUCTURE

The *KISS-Tree* is an in-memory index structure that is based on the generalized prefix tree and improves it in terms of reduced memory accesses and memory consumption. Figure 1 shows an overview of the *KISS-Tree*. Like in the generalized prefix tree, the 32bit key is split into fragments f_{level} that identify the bucket within the node on the corresponding level. While the prefix tree uses an equal fragment length on each tree level, the *KISS-Tree* splits the key into exactly three fragments, each of a different length. This results in three tree levels, where each level implements different techniques for storing the data. The fragment sizes for each level play an important role because of the individual characteristics of each level. In the following, we describe the characteristics of each level and the reason for the respective fragment size in detail.

3.1 Level 1–virtual level

The first level uses a fragment length of 16. Therefore, this level differentiates the 16 most significant bits of the key. The technique we deploy here is *direct addressing*. *Direct addressing* means that there is no memory access necessary to obtain the pointer to the node on the next level. Instead, the pointer is directly calculated from the fragment f_1. In order to make *direct addressing* possible, the next tree level has to store all nodes sequentially. The major advantage we gain from *direct addressing* is that we neither have to allocate any memory for the first level nor have to issue a memory access when descending the tree. Thus, we call this level the *virtual level*.

3.2 Level 2–on-demand level

The fragment size for the second level is set to 10. Thus, we use the next 10 bits of the key to determine the bucket inside a node on this level, which contains a pointer to the corresponding node on the next level. On this level, we deploy two techniques. The first technique is the *on-demand* allocation, provided by the platform and the operating system. Todays computers know two address spaces, which are the virtual and the physical address space. All applications that are running get their own virtual address space that is transparently mapped to the shared physical one. This mapping is done by a piece of hardware known as the memory management unit (MMU). The MMU uses the page directory that is maintained by the operating system to translate virtual addresses into physical addresses. If an application tries to access a virtual address that is either not mapped to physical memory or is write protected, the MMU generates an interrupt and invokes kernel routines to handle this interrupt. The kernel has two options to handle this interrupt. Either it terminates the application or physically allocates a new piece of memory. The second choice is called *on-demand* allocation. Here, we explicitly ask the kernel to allocate a large consecutive segment of memory in the virtual address space and do not allocate any physical memory at all. Thus, the second level is entirely virtually allocated at startup and does not consume any physical memory at this point of time. As soon as we write something to a node on the second level, the physical memory for this node is allocated by the operating system and actually consumes memory. The benefit we gain from the *on-demand* allocation is that we fulfill the requirements given by the first level (sequentially stored nodes) and actually do not have to

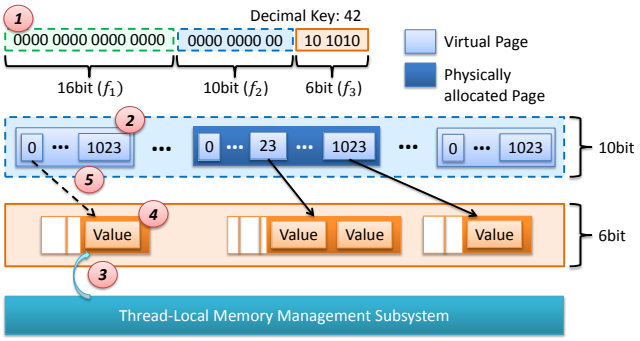


Figure 3: Update Operation on KISS-Tree.

waste the real physical memory for that. The critical point for *on-demand* allocation is the granularity of the memory mappings. For instance, if we write a 4Byte pointer on this level, the operating system has to allocate physical memory at the size of memory pages. The smallest available size for memory pages is currently 4KB on common architectures. Thus, when writing a small pointer, *on-demand* allocation has to map 4KB contiguous virtual address space to 4KB contiguous physical address space. To summarize, the complete second level is virtually allocated on index creation via one *mmap* call and does not claim any physical memory at this point of time. Only if a pointer is written to this level, the corresponding 4KB page gets physically allocated. Concurrent allocations of 4KB pages are solely handled by the operating system.

Because of the allocation granularity of 4KB, it makes sense to use 4KB nodes on this level. A node on this level consists of $2^{10} = 1024$ buckets because of the fragment length of 10. Thus, we need 4Byte pointers inside these buckets to get a node size of 4KB. Here we deploy our second technique called *compact pointer*. *Compact pointers* reduce standard 64bit pointer to 32bit pointer. The compaction is possible, because nodes on the third level are of one of the $2^6 = 64$ distinct sizes and the maximum number of child nodes of all nodes on the second level is 2^{26}. Thus, we are able to use the first 6 bits of a *compact pointer* for storing the size of the next node, which is later translated to an offset. The remaining 26 bits are used for storing the block number.

Using *compact pointers* for achieving a node size of 4KB on the second level has the effect that as soon as one pointer is written to a node, the entire node becomes physically allocated and consumes memory. The maximum number of nodes that are possible on this level are 2^{16}. This means that in the worst case scenario, the second level consumes 256MB of memory for 2^{16} keys. We investigate this issue of memory consumption further in Section 6.

3.3 Level 3–compressed nodes

On the last level, we use a fragment size of 6. Thus, the lowest 6 bits of the key determine the bucket inside a node on this level. Because the last level has the largest number of possibly present nodes (2^{26} at maximum), we apply a compression to the nodes on this level, which is similar to the compression scheme used for the CTrie. The compression works by adding a 64bit bitmap in front of each node. Because there is a maximum of $2^6 = 64$ buckets in a

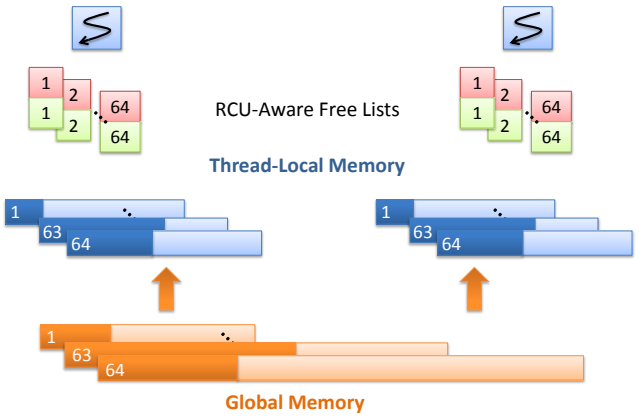


Figure 4: Memory Management.

node on the last level, the bitmap indicates which buckets are actually in use. Thus, we only have to store these buckets in use beyond the bitmap instead of storing all 64 buckets even if only a fraction of them has content. All buckets in a node on this level contain the value for the corresponding key. For duplicates and transactional isolation, it is also possible to store additional information as content.

Compression leads to nodes that shrink and grow over time. To handle this resizing and allowing latch-free compare-and-swap operations for updates, we use read-copy-updates (RCU) [9]. RCU creates copies of nodes and merges changes to this private copy. This allows readers to read consistent data from the original node, while the updated node is created. When the updated node becomes ready, we use an atomic compare-and-swap operation to exchange the corresponding pointer on the second level with a pointer to the new node. In order to allow arbitrary values and avoid lost updates, we do not make any in-place updates.

4. OPERATIONS

In this section, we describe how common index operations like updates and reads are processed on the *KISS-Tree*. The *KISS-Tree* also supports deletions and iterations, which are trivial to derive from the described operations.

4.1 Updates

We demonstrate the update respectively insertion of a key with the help of the *KISS-Tree* (8Byte Rids as values) depicted in Figure 3. At the beginning, the tree contains three key-value pairs and one physically allocated page on the second level. In this example, we insert the decimal key 42 into the *KISS-Tree*. The update operation starts with splitting the key into the three fragments f_1,f_2, and f_3 each of their respective length (step 1 in the figure). The first 16bit long fragment f_1 is used to identify the corresponding node on the second level (step 2). Because all nodes on the second level are of equal size and are sequentially stored, we can directly calculate the pointer to that node. Following the second step, we use the 10bit fragment f_2 to determine the bucket inside the second level node. Because the entire node is not physically allocated, the node contains only zero pointer[2], which indicates an empty bucket. The insert operation now remembers the zero pointer for the later compare-and-swap operation. Afterwards, we have to request a new node from the memory management subsystem that is able to store the nodes bitmap and exactly one value (step 3). In step 4, we prepare the new node for tree linkage. Therefore, we set the corresponding bit in the bitmap, which is determined by f_3. In our specific example, we have to set the 42th bit. Afterwards, we write the actual value after the bitmap. Finally, we try to exchange the pointer on the second level with the *compact pointer* to the new level three node using a compare-and-swap instruction (step 5). If the compare-and-swap is successful, the operation finished and the first 4KB node on level two is now backed with physical memory; otherwise the entire update operation has to repeat.

[2]The virtual memory is allocated via *mmap*, which initializes newly allocated pages for security reasons with zero

4.2 Reads

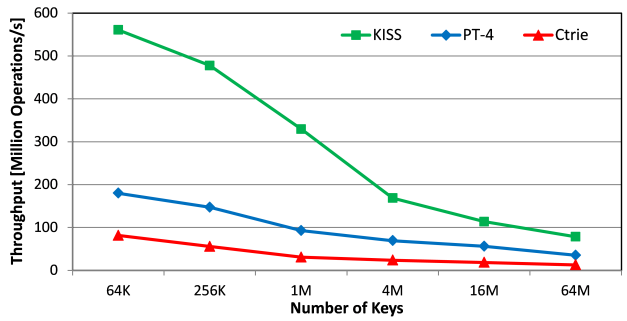
Assuming that the previously inserted decimal key 42 is now present in the *KISS-Tree* depicted in Figure 3, we now show how to read it again. We take f_1 to calculate the absolute address of the corresponding node on the second level and identifiy the bucket in this node using f_2. Because we read a non-zero pointer from this bucket, the bucket is in use and we translate the *compact pointer* to a 64bit pointer to the next node on the third level. To check whether the key is present, we test the 42th bit of the nodes bitmask. Because the bit is set, we apply a bitmask to the bitmap to unset all bits behind the 42th bit. Now we count the number of set bits using a population count instruction to obtain the bucket inside the compressed node, which contains the requested value.

4.3 Batched Reads

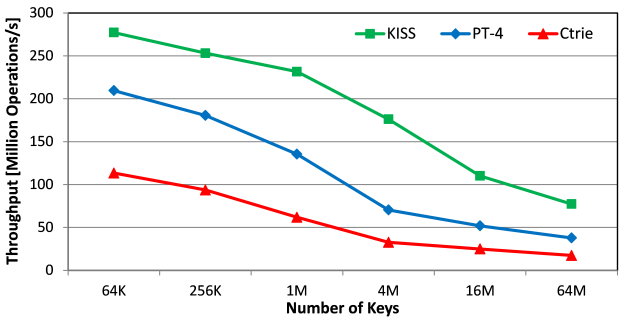
Especially on large *KISS-Trees* that do not fit into the CPU cache anymore, the memory latency plays an important role and decreases the performance of reads. In order to reduce the latency, we applied batched reads for latency hiding. With batched reads, each thread working on the tree executes multiple read operations at once. The number of simultaneous operations is the *batch size*. Executing multiple operations at once changes the way of processing. Instead of reading the nodes of level two and three alternately, the batched reads operation reads only the second level nodes at first, followed by the third level nodes. This gives us the advantage that nodes are potentially still in the cache when reading them multiple times. Moreover, we can issue prefetch instructions to bring the third level nodes in the cache while still reading level two nodes. The disadvantage of batching reads is an increased latency, which is traded for a higher throughput. However, certain database operations, like joins, are able to profit a lot from batch processing.

5. MEMORY MANAGEMENT

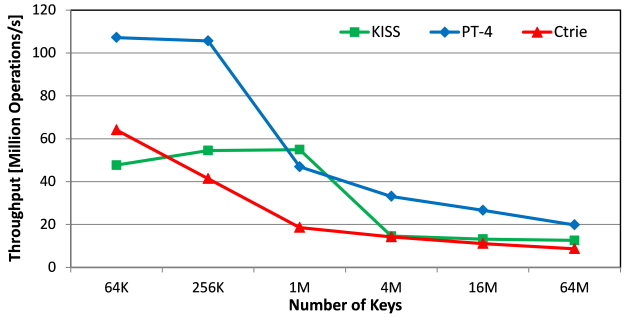
The memory management subsystem is tightly coupled to the *KISS-Tree*, because it is a critical component regarding the overall index performance. Especially the RCU mechanism deployed on the third level requires a responsive memory allocation. During the startup of the system, the memory management allocates consecutive virtual memory segments (physical memory is claimed on-demand) for each of the 64 node sizes possible on the third level as depicted at the bottom of Figure 4. Each segment consists of a maximum of 2^{26} blocks of the respective node size. We need the blocks of



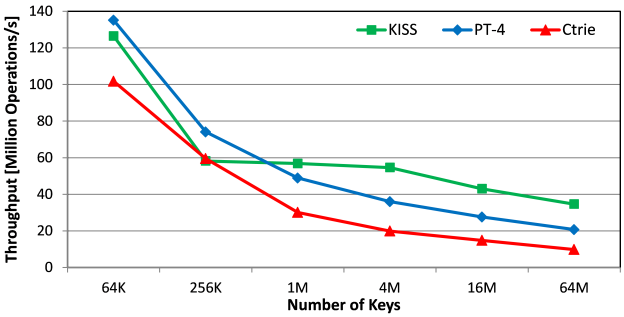
(a) Throughput for Reading Sequences.



(b) Throughput for Reading Uniform Data.



(c) Throughput for Updating Sequences



(d) Throughput for Updating Uniform Data

Figure 5: Read/Update Throughput for Sequences and Uniform Data.

each size to be in a row, because of the block-oriented compact pointers used on level two. In order to allow fast parallel access to the memory management subsystem, each thread allocates only a small number of blocks from each segment of the global memory and maintains them on its own (thread-local memory). This allows threads to administrate their memory independent from each other and removes synchronization bottlenecks. Only if a thread runs out of memory, it requests a new set of blocks from the global memory, which is synchronized via atomic operations. Moreover, every thread uses its own free lists for memory recycling. Those free lists have to be aware of the RCU mechanism, in order to prevent the recycling of a node that is still read by another thread. For our experiments, we used an active and an inactive free list per node size and thread. Freshly freed memory pieces are stored in the inactive free list and memory allocation requests are served by the active one. As soon as a *grace period* is detected, the memory management switches over to the inactive free list.

6. EVALUATION

In this section, we evaluate the *KISS-Tree* performance and memory usage for different workloads, tree sizes, update rates, and platforms. Moreover, we compare the results to the generalized prefix tree and the CTrie. We used 32bit keys and 64bit values as they are common for rids. All experiments, we conducted, were executed on an *Intel i7-2600 (3.4GHz clock rate, 3.8GHz max. turbo frequency, 4 cores, Hyper-Threading, 2 memory channels, 8MB LLC)* equipped with *16GB RAM (DDR3-1333)* running *Ubuntu Linux 11.10*.

In the first experiment, we compared the read/write throughput of the *KISS-Tree*, generalized prefix tree with

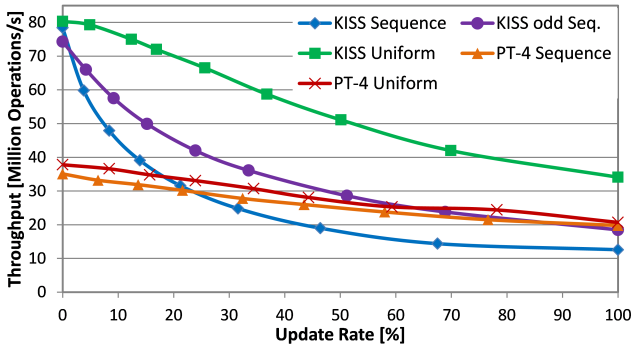


Figure 6: Throughput for different Update-Rates.

$k' = 4$ (PT-4), and CTrie for different tree sizes. We used all of the eight available hardware threads on the platform. For all experiments, we use a sequence and a uniformly distributed workload. Both workloads describe the upper respectively lower boundary for throughput and for instance, skewed distributions lie in between. The tree size reaches from 64 thousand (compute-bound) up to 64 million (memory-bound) keys present in the trees. All the three tree implementations are using batching for read operations as described in Section 4.3. Figure 5(a) shows the read throughput for sequences (keys are randomly picked from the sequence range). The *KISS-Tree* shows the highest throughput for small as well as for large trees. Figure 5(b) visualizes the throughput for uniformly distributes keys. Here, the *KISS-Tree* also shows the best performance, but the throughput for small trees is lower because the locality

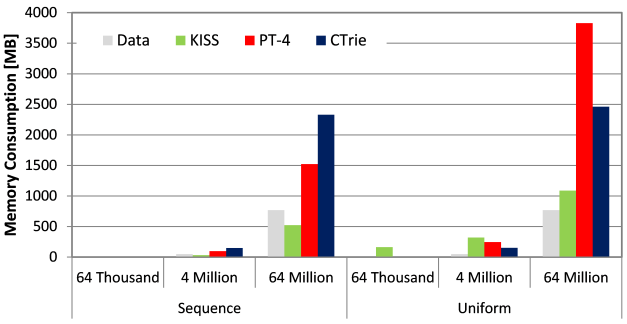


Figure 7: Comparison of Memory Consumptions.

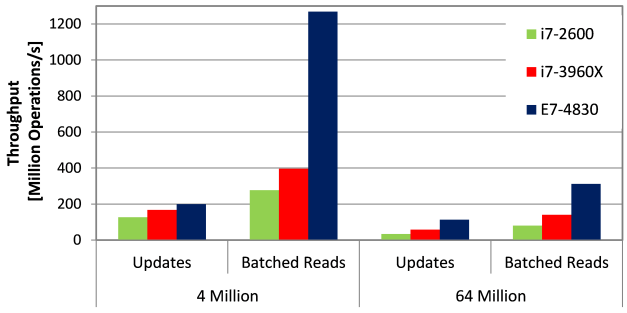


Figure 8: Throughput on different Platforms.

of data got worse. The main observation is that the *KISS-Tree* with the least memory accesses per read operation(2-3) has the highest throughput compared to the CTrie which requires up to 14 memory accesses. Figure 5(c) and Figure 5(d) show the update throughputs for both key distributions. The CTrie shows the worst results because of the high number of memory accesses per key. We measured the highest throughput for the PT-4, because it is able to do in-place updates where the *KISS-Tree* has to create copies of third level nodes. We see this effect when comparing the sequence workload to the uniform workload. With the sequence workload, all third level nodes contain 64 values that have to be copied for an update. The uniform workload on the other hand creates only sparely used nodes on the third level, which results in a better update performance. Moreover, we observe a lot of compare-and-swap failures on small trees for the sequence workload, because threads work on the same third level nodes. To summarize the first experiment, the *KISS-Tree* clearly shows the best read performance and an update performance that is able to keep pace with the CTrie and the PT-4.

With the second experiment, we investigate the overall throughput for workloads with different update rates on the *KISS-Tree* and the PT-4. Figure 6 shows the respective results. While the throughput of the PT-4 moves smoothly between 0% and 100% update rate, the behaviour of the *KISS-Tree* heavily depends on the key distribution. This is mainly caused by the RCU update mechanism, which has to copy entire nodes and so occupies the memory channels what also effects the other read operations. The one extreme is the uniform workload that does not require much overhead for copying nodes. Thus, the *KISS-Tree* is always faster than the PT-4. As worst case we identified the sequence

Processor	#threads	#channels	LLC(MB)	Freq.(GHz)
1x i7-2600	8	2	8MB	3.4
1x i7-3960X	12	4	15MB	3.3
4x E7-4830	4x16	4x4	4x24MB	2.13

Table 1: Evaluation Hardware

workload, where update operations always have to copy 64 values. Here, the overall throughput drops below the one of the PT-4 at an update rate of about 20%. To show the behavior between both extremes, we added the odd sequence key distribution, which only includes odd keys. This ends up in 32 values per third level node and a throughput that is better or equal to the PT-4.

The next experiment addresses the memory consumption of the *KISS-Tree*, PT-4, and CTrie for different key distributions and tree sizes. Figure 7 visualizes the measurements. For the sequence workload, the *KISS-Tree* turns out to have a very low memory consumption that is almost equal to the actual data size. The CTrie shows the highest memory usage, because the compression and the *I-Nodes* add additional overhead to the structure compared to the PT-4. When looking at the measurements for the uniform workload, the *KISS-Tree* wastes a lot of memory on small trees. This is caused by the second level nodes that are allocated at a 4KB granularity. For instance, a *KISS-Tree* that contains 4 million uniformly distributed keys uses 256MB on the fully expanded second level. This is the worst case scenario for memory consumption; with an increasing number of keys, the *KISS-Tree* starts to save a lot of memory compared to the other trees. The measurements for the uniform workload also show the worst case scenario for the uncompressed PT-4, which consists of very sparely used nodes. Here, the compression of the CTrie saves memory.

Finally, our last experiment investigates the performance of the *KISS-Tree* on the three different platforms listed in Table 1. Figure 8 contains the measurements for uniformly distributes keys. The experiment shows that the read performance scales with the total number of hardware threads and the clock rate. A problem we observed is the update performance on the massive parallel NUMA system. With 64 hardware threads updating a small tree, we have a high probability that threads try to update the same third level nodes. For this reason, updating a small *KISS-Tree* does not scale well on massive parallel hardware. As soon as the tree becomes larger, the scalability is given, because threads work on different nodes.

7. CONCLUSION AND FUTURE WORK

With the movement from disk-based database systems towards pure in-memory architectures, the design goals for index structures essentially changed. The new main optimizations targets are cache-awareness, memory access minimization and latch-freedom. Thus, classic disk-block optimized structures like B+-Trees are not suited for modern in-memory database architectures anymore. In this paper, we introduced the *KISS-Tree*, which addresses exactly these kinds of optimizations. With the *KISS-Tree*, we accomplished to reduce the number of memory accesses to 2–3 by deploying multiple techniques on the individual tree levels. The techniques are *direct addressing*, *on-demand allocation*,

compact pointer and *compression*. The evaluation revealed that the minimization of memory accesses is the most beneficial optimization for large indexes. In combination with latency hiding through batched reads, we were able to outperform existing tree-based in-memory index structures. Moreover, the *KISS-Tree* provides a high update rate, because it does not involve comprehensive structural maintenance and exploits parallelism with the help of optimistic compare-and-swap instructions. Regarding memory consumption, we showed that the *KISS-Tree* has very low indexing overhead, especially for large trees, and even consumes less memory than the actual data for sequential workloads.

Our future work will mostly focus on improving the update performance and adding support for larger keys. The critical point for the update performance is that we have to copy entire nodes for existing keys. This occupies the memory channels and increases the probability for the compare-and-swap to fail. To fix this issue, we will have to make changes in the second tree level. One possible solution for this problem would be the inclusion of version numbers into the compact pointers on the second level, so that each update changes the pointer. This way, we increase the update throughput that is independent of the workload.

The current main drawback of this *KISS-Tree* is, that it misses support for large integers or varchars. Therefore, we have to look for solutions to apply other techniques on the additionally required levels or consider combinations with other index structures. We expect those additional levels to deploy compression mechanisms for sparely occupied nodes as well as a vertical compression of node chains. Another important point for research is to push the performance on NUMA systems. NUMA architectures are completely different from SMP machines because there are additional costs for accessing memory on foreign sockets and especially the cache utilization on different sockets. Moreover, the cache-coherency has to be taken into account.

8. ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing”.

9. REFERENCES

[1] Dexter project. `http://wwwdb.inf.tu-dresden.de/dexter`.

[2] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes*, pages 245–262. Software pioneers, New York, NY, USA, 2002.

[3] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *BTW*, pages 227–246, 2011.

[4] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, Apr. 2002.

[5] 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 *SIGMOD*, pages 339–350, 2010.

[6] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[7] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6:650–670, December 1981.

[8] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. VLDB ’86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[9] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems.

[10] A. Prokopec, P. Bagwell, and M. Odersky. Lock-free resizeable concurrent tries. *LCPC*, 2011.

[11] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD Rec.*, 29:475–486, May 2000.

[12] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.

[13] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

APPENDIX

In the appendix, we conducted as set of experiments that investigate the behavior of batched reads and scalability as well as a comparison to the CSB+-Tree.

A. BATCHED READS EVALUATION

In this section, we evaluate the performance of batched reads. Compared to standard read operations, batched reads process multiple read operations per thread at once to hide the latency of memory accesses. All of the following experiments were conducted on an *Intel i7-2600* processor, which corresponds to the machine used in Section 6.

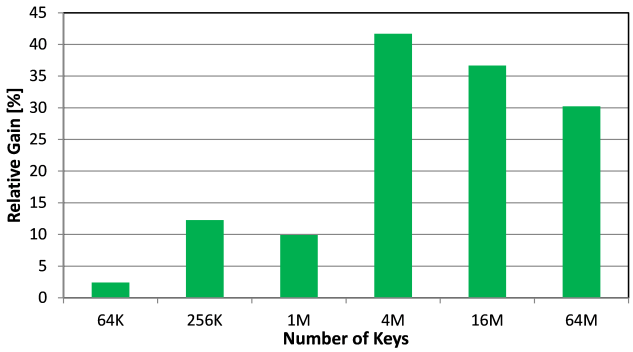


Figure 9: Relative Performance Gain through Batched Reads for Sequences.

In first two experiments, we measured the throughput of standard read operations and batched read operations for different tree sizes. Figure 9 shows the relative performance gain of batched reads over standard reads for sequential keys (keys randomly picked from the sequence range). We observe that this performance gain strongly depends on to size of the tree. A small tree, for instance, takes not much advantage from batched reads, because most of the data is in

the L1 cache of the CPU. Thus, we are unable to hide any latency. As soon as the L1 cache is exhausted, batched reads are able to hide latency between the L1 cache and the L2 cache. With more and more growing tree sizes, we hit the point where even the L2 cache and the LLC are exhausted. Thus, we have a high latency from the memory controller to the L1 cache, which gives us the highest performance gain for batched reads.

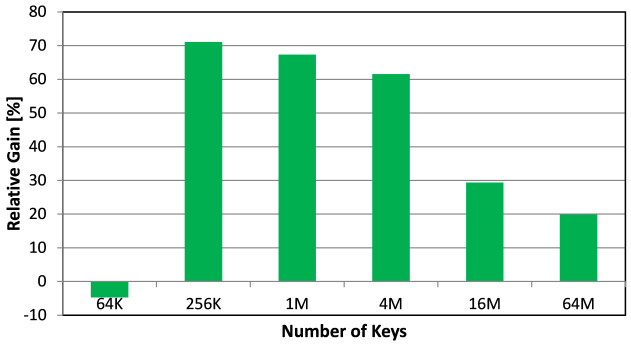


Figure 10: Relative Performance Gain through Batched Reads for Uniform Data.

Figure 10 visualizes the measurements for a uniform key distribution. On small trees that fit in the L1 cache of the CPU, we observe a performance loss by batching reads because prefetch instructions induce an additional overhead. When looking at trees of medium size, the performance gain is higher compared to the sequential keys. This is the case because prefetch instructions are unlikely to fetch same cache lines twice. Thus, batched reads are able to hide more latency.

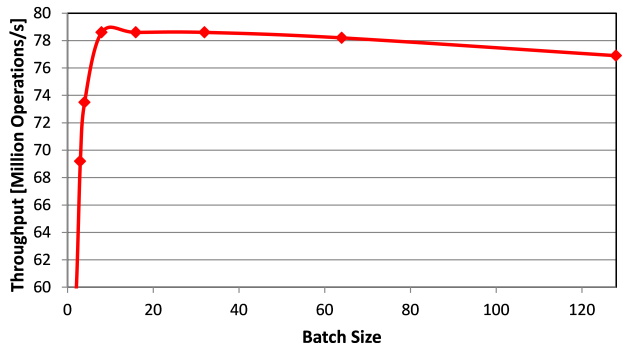


Figure 11: Throughput as a Function of Batch Size.

In the final experiment, we measured the throughput for different batch sizes. The batch size is the number of read operations that are simultaneously processed by a thread. The results are depicted in Figure 11. The experiment revealed that even small batch sizes dramatically increase the read throughput. With a batch size of eight, batched reads achieve the maximum throughput, which remains steady until it starts to slowly decrease with large batch sizes.

B. SCALABILITY

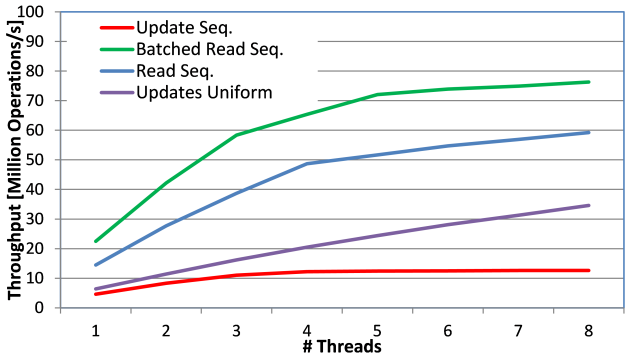


Figure 12: Scalability for Sequences (64M Keys).

To investigate the scalability of the *KISS-Tree*, we measured the throughput of different operations with a variable number of threads. The hardware we used for this experiment was an *Intel i7-2600*, which has four physical cores with Hyper-Threading (a total of 8 hardware threads) available. The experiment depicted in Fig. 12 revealed that a standard read operation scales with the number of physical cores. As soon as the physical cores are exhausted, the performance gain drops, because threads have to share cores. We observe another behavior for batched reads, which scale only well with the first three threads. This happens, because the batched operations have a better utilization of the memory controllers, which become the bottleneck with an increasing number of threads. The same effect can be observed, when comparing updates of sequential and uniform data. Because the sequential workload requires the copying of full nodes on the third level, the memory controllers become the limiting factor. However, the update operation of the uniform workload scales well with all available hardware threads.

C. COMPARISON TO THE CSB+-TREE

In this section, we compare the *KISS-Tree* to the CSB+-Tree implementation from [11]. This 32bit CSB+-Tree implementation neither implements concurrency control nor batched reads. Therefore, we measured the single-threaded throughput for simple uniformly distributed reads on an *Intel i7-2600*. The CSB+-Tree implementation performs about 3 million operations per second and thread with a tree size of 64 million keys. The *KISS-Tree* achieves 18.4 million operations per second and thread, which is more than 6 times faster.

Making Cost-Based Query Optimization Asymmetry-Aware

Daniel Bausch
bausch@dvs.tu-darmstadt.de

Ilia Petrov
petrov@dvs.tu-darmstadt.de

Alejandro Buchmann
buchmann@dvs.tu-darmstadt.de

Technische Universität Darmstadt
Department of Computer Science
Databases and Distributed Systems Group
Hochschulstraße 10
64289 Darmstadt, Germany

ABSTRACT

The architecture and algorithms of database systems have been built around the properties of existing hardware technologies. Many such elementary design assumptions are 20–30 years old. Over the last five years we witness multiple new I/O technologies (e.g. Flash SSDs, NV-Memories) that have the potential of changing these assumptions. Some of the key technological differences to traditional spinning disk storage are: (i) asymmetric read/write performance; (ii) low latencies; (iii) fast random reads; (iv) endurance issues.

Cost functions used by traditional database query optimizers are directly influenced by these properties. Most cost functions estimate the cost of algorithms based on metrics such as sequential and random I/O costs besides CPU and memory consumption. These do not account for asymmetry or high random read and inferior random write performance, which represents a significant mismatch.

In the present paper we show a new asymmetry-aware cost model for Flash SSDs with adapted cost functions for algorithms such as external sort, hash-join, sequential scan, index scan, etc. It has been implemented in PostgreSQL and tested with TPC-H. Additionally we describe a tool that automatically finds good settings for the base coefficients of cost models. After tuning the configuration of both the original and the asymmetry-aware cost model with that tool, the optimizer with the asymmetry-aware cost model selects faster execution plans for 14 out of the 22 TPC-H queries (the rest being the same or negligibly worse). We achieve an overall performance improvement of 48% on SSD.

1. INTRODUCTION

Database systems, their architecture and algorithms are built around the I/O properties of the storage. In contrast to Hard Disk Drives (HDD), Flash Solid State Disks (SSD) exhibit fundamentally different characteristics: high random and sequential throughput, low latency and power consumption [3]. SSD throughput is asymmetric in contrast to magnetic storage, i.e. reads are significantly faster than writes. Random writes exhibit low performance, which also degrades over time. Interestingly enough many of those properties also apply to other novel I/O technologies such as NV-Memories [4].

Precise cost estimation for query processing algorithms is of elementary importance for robust query processing and predictable database performance. Cost estimation is the basis for many query optimization approaches. The selection of the 'best' query execution plan correlates directly with database performance.

Taking hardware properties properly into account is essential for cost estimation and query optimization, besides the consideration of data properties (data distribution, ratios, access paths) and intra- and inter-query parallelism. Cost functions were built on assumptions about hardware properties which are now 20–30 years old. Some of these change fundamentally with the advent of new I/O technologies.

Due to the symmetric read/write characteristics and the high random I/O cost of traditional spinning disks, the I/O behavior is approximated by counting sequential and random storage accesses and weighting them with different factors. Flash SSDs as well as other new I/O technologies exhibit: read/write asymmetry (different for sequential and random I/O); and very good random read performance, i.e. random and sequential read costs converge. Hence the following factors can yield incorrect cost estimation: (i) not distinguishing between reads and writes; (ii) not accounting for random writes; (iii) not accounting for I/O parallelism and read operations.

In a previous study [1], we found that the optimal plan to answer a query indeed can depend on the used storage technology. In the present paper we report about our efforts to improve query optimization in respect to storage technologies. We show incremental improvements to the cost model of the open source DBMS PostgreSQL. The improvements are derived from observable behavior of the query processing algorithms of that DBMS and are supported by theoretical considerations. Besides sequential and random patterns the new model distinguishes reading and writing making it *asymmetry-aware*. Additionally a tool based on an iterative heuristic was built to automatically find good base coefficients for the configurable parameters of cost models. To simulate the situation of data-intensive systems under heavy load the test system was setup with tight memory settings relative to the size of the data. With tuned param-

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA

eters the DBMS with the new model is able to perform a TPC-H derived workload faster than a vanilla PostgreSQL with equally intensively tuned cost model parameters.

The rest of the paper is organized as follows: after a discussion of related work (Section 2) we describe the details of the asymmetry-aware optimizer cost model (Section 3). The adapted cost functions for Sorting and Hash-Join are presented in Sections 3.3.1 and 3.3.2 respectively. The asymmetry-aware model is implemented in PostgreSQL[1] and tested with an open source benchmark build around a TPC-H schema and data [16, 13]. The experimental setup, its results, and their discussion follows in Section 4.

2. RELATED WORK

Query optimization has been a research topic in database systems right from the beginning [15]. There has been a very large body of literature on the topic; some of the survey works are [8, 2, 7]. All these works provide the basis for the present paper.

The work by Pelley et al. [14] treats topics similar to the ones considered in our previous work [1]. Pelley et al. measure different scan and join algorithms' performance with varying query selectivity. Their results show only a small selectivity range where the optimal algorithm for an HDD is sub-optimal for an SSD. From that observation they extrapolate the generalized conclusion that optimizers do not need to be made SSD-aware for practical means. Their work also relates to [6]. In [1] we explored a similar problem setting but arrived at the conclusion that different query execution plans are best suited for different hardware. The model presented in the present paper features additional degrees of freedom to represent properties of asymmetric storage devices.

3. ASYMMETRY-AWARE COST MODEL

As a basis the open source DBMS PostgreSQL is used. PostgreSQL features a cost-based query optimizer. Cost functions in that optimizer calculate frequencies of expected operations, weight them, and are summed up to a scalar cost value per alternative plan. The plan which received the lowest cost value is executed.

3.1 Behavior of the Query Executor

Before discussing PostgreSQL's cost model and the suggested changes, we briefly introduce the behavior of the query execution algorithms in this section, while we focus on the I/O behavior.

3.1.1 Scanners

PostgreSQL features four ways to scan data, which produce read-only access patterns of different distributions.

The *Sequential Scan* algorithm implements the "full table scan". Disk is accessed in natural order, i.e. sequentially.

The *Index Scan* algorithm accesses tables indirectly by looking up the tuples' locations within an index first. Conditions specified in the query are used to narrow the range of relevant index and table portions. This may produce random patterns depending on index-to-table correlations and/or the order the keys are looked up in a complex plan.

Then there is *Bitmap Scan*, a variant of *Index Scan* which first records all potentially matching locations as a bitmap. After that only those portions are scanned in on-disk order. Bitmaps originating from different conditions or different indexes on the same table can be bit-logically combined prior to scanning. The amount of randomness induced by this algorithm depends on the number of holes in the bitmap.

Finally, the *Tuple-ID Scan* algorithm handles the special case when tuples are explicitely requested by conditions of the form "ctid = ..." and "WHERE CURRENT OF" expressions. Its access pattern is unpredictable.

3.1.2 Sorting and Hash Join

The sort and hash-joining algorithms produce mixed write and read I/O, if their temporary data does not fit in the main memory slice available to a single algorithm.

PostgreSQL's sort algorithm is basically an implementation of a combination of *Heap Sort* and *Polyphase Merge Sort With Horizontal Distribution*, both found in [10]. Its first partitioning phase produces mainly sequential writes[2]. The multiple merge phases tend to produce more randomly targeted read and write accesses because space occupied by read pages is directly reused for new sorted runs. The amount of randomness, however, depends on whether the input data is pre-sorted or not. We will experimentally show this in Section 3.3.1.

Hash Join in PostgreSQL means *Hybrid Hash Join*. *Hybrid Hash Join* first splits the data of the inner table into mulitple batches which can be processed in RAM at a time. The first batch is held in RAM so it can be processed directly after splitting. The tuples going to secondary batches are appended to multiple temporary files, one for each batch, in parallel. The hash table for each batch is created in memory when the batch is processed. Tuples of the outer table whose join field hashes to secodary batches are postponed and written to temporary files, too. Although the tuples are strictly appended to the temporary batches, this produces a lot of random writes as we will experimentally show in Section 3.3.2.

3.1.3 Materialization and Re-Scanning

If the identical result of an execution plan sub-tree is needed multiple times by a parent node, it is materialized, i.e. temporarily stored. If the intermediate result fits in the memory slice available for an algorithm, it is held in RAM, otherwise it is stored sequentially to temporary on-disk storage. When the data is used again, it is sequentially streamed from disk.

3.2 Original Model

PostgreSQL's cost model is organized parallel to its individual query processing algorithms so there is a one-to-one relationship between algorithms and elementary cost functions. A complete mathematical transcript of the cost model and its changes can be found in Appendix A. We focus on the I/O part of the functions as the computational part was unchanged. Some functions are notated slightly different to the calculations in the program code and some technical specialities are left out for easier understanding. In this section we give an idea of how the shown behavior of the executor has been translated into a cost model by the PostgreSQL developers.

[1] `http://www.postgresql.org`

[2] The cost for sourcing the data is accounted using the appropriate formulas respective to the sourcing plan.

The distinction of I/O access patterns is implemented as configurable weight factors. As of writing PostgreSQL's cost model accounts for sequential and random accesses using two different factors.

The cost functions for the scan algorithms estimate the number of pages to be read for the different tasks within the algorithms. These numbers are multiplied with one of the two configurable weight factors depending on whether the respective operations are expected to produce sequential or random accesses. For *Sequential Scan* the model multiplies the number of pages of the scanned relation with the factor for sequential accesses; for *Index Scan* sophisticated computations are performed to convert the estimated selectivity and a statistics value about the index-to-table correlation into the number of required page reads and a fraction for the randomness of these accesses; for *Bitmap Scan* the number of holes which would lead to skips is approximated based on the selectivity, i.e. small result fractions are assumed to produce more random accesses; the unpredictability of the *Tuple-ID Scan* algorithm is treated by the worst case assumption: every tuple expected to be accessed is counted as a random page read; and, finally, *re-scanning* a previously materialized intermediate result is counted by multiplying the expected size with the parameter for sequential accesses. Interestingly, the same cost formula models the *materialization* itself.

The I/O cost of a *sort* operation is modeled by a typical $O(n \log n)$ formula. The number of input pages times 2 (for write and read) is multiplied with the expected number of sort phases, which is the logarithm to base of the merge order[3] of the number of expected intial sorted runs. This estimates the total number of page accesses, which is then weighted with $\frac{3}{4}$ sequential and $\frac{1}{4}$ random.

The I/O cost for writing and further processing of additional external batches of data in the *Hash Join* algorithm is accounted by multiplying two times the relevant data size with the parameter for sequential accesses.

3.3 Modifications

The proposed *asymmetric cost model* provides four configuration parameters regarding I/O instead of only two. These allow to distinguish not only sequential and random operations but also whether data is read or written. Each of the old parameters is split into a parameter representing read operations of the given access pattern and another parameter that represents write operations performed with the same pattern. In the cost functions the old parameters are replaced with the new ones according to the behavior of the algorithms. For easier comparison we restrict our model modifications to parameter replacements which allow us to reproduce the cost values of the old model by the new model using certain parameter settings.

The scanners perform pure read loads while *materialization* performs a pure write load; however, in both cases the associated cost functions can be converted to the new parameter set by substituting old parameter variables with new ones: "sequential read" and "random read" instead of just "sequential" and "random" for the scanners and "sequential write" instead of "sequential" for materialization.

The cost functions for the *sort* and *hashjoin* algorithms need more attention because their algorithms perform read loads as well as write loads while their original cost functions do not model those loads separately. Sections 3.3.1 and 3.3.2 will show in-detail how this was resolved.

3.3.1 Adapted cost function for sort

As we focus on the parts of the cost functions representing storage accesses we will look at the external sort only.

The original cost function for this algorithm assumed $\frac{1}{4}$ of the block accesses as random and $\frac{3}{4}$ as sequential, together representing all the reads and writes happening for this external sort. It is obvious that everything that is written to the temporary storage is read later again. Therefore we assume that half of the accesses were originally counted as reads and the other half as writes.

To get a realistic assigment for the random-to-sequential ratio, we traced the requests on the block layer of the operating system using *blktrace*[4]. This revealed the first line of the statistics shown in Table 1 for an external sort by the `l_partkey` column of the LINEITEM table of a TPC-H data set. In these statistics, a request is counted as sequential, if its start address is immediately following the end address of the preceding request.

Often query plans include sort operations carried out on data that is already stored in the requested order. The second row of Table 1, therefore, shows the statistics for a sort of the LINEITEM table by `l_orderkey`. In freshly loaded data, the LINEITEM table physically contains monotonically ascending order keys. Sorting by that column the shares of sequential and random operations exchange. Where the sorting of unordered data showed a high random share, there is now a high sequential share. We conclude, there is a high data dependency for the sort algorithm. As a compromise we assume that $\frac{1}{2}$ of the requests are sequential and $\frac{1}{2}$ are random. However, we count the first partitioning phase as only sequential writes, because in that phase the algorithm appends all new sorted runs to the end of the temporary file only.

Sorting in a join context.

A single execution of TPC-H query number 12 reveals the third line of Table 1 when it is answered using a sort-merge join with external sort. These numbers are similar to the ordered case above, and indeed, the main data portion that is sorted here is already ordered on disk. Only the smaller table (ORDERS) involved in the join really needs the sort operation while the other bigger table (LINEITEM) is already stored physically in the requested order. This further supports the assumptions we made for the simple sort case.

3.3.2 Adapted Cost Function for HashJoin

A second algorithm featuring mixed read and write operations is the *Hash Join* algorithm.

To determine the cost function for the *hashjoin* algorithm we conducted block tracing experiments very similar to the ones we did for the *sort* algorithm. The join from TPC-H query number 12 performed with PostgreSQL's hash join algorithm shows access patterns which can be summarized to the statistics shown in line four of Table 1. These numbers show a strong random write tendency and a fair sequential read tendency.

[3]The implementable merge order depends on the memory slice available to the algorithm.

[4]`http://git.kernel.dk/?p=blktrace.git`

Table 1: Temporary I/O Access Patterns

	write		read	
	sequential	random	sequential	random
external sort of unordered data	30.42%	69.57%	5.47%	94.52%
external sort of ordered data	77.15%	22.84%	95.89%	4.10%
sort-merge join	75.40%	24.59%	91.71%	8.28%
hash join	5.61%	94.38%	71.40%	28.59%

So partitioning produces random writes mostly. This can be explained as the filesystem (ext3) keeps the temporary batch files separated from each other by pre-allocation of space. This way individual batches are stored mainly consecutive, what in turn explains why reading them produces sequential reads most of the time.

As an approximation we therefore treat all read operations in the join phase as sequential accesses. The writes in the partitioning phase we treat as random as clearly indicated by the statistics. The resulting typical $O(n+m)$ formula is shown in the appendix.

4. EXPERIMENTAL ANALYSIS

In this section we will present the experiments we conducted to show the efficiency of the new model. In Section 4.1 we will define the system configuration and the used benchmark. Section 4.2 will illustrate the way we compare the different models. The results of the experiments are presented in Section 4.3 and discussed in Section 4.4.

4.1 System and load

For our experiments we used two identical computer systems. One of them was dedicated to a PostgreSQL installation using the modified cost model, while the other was installed with a vanilla PostgreSQL for reference. The used base version of PostgreSQL was 9.0.1. The systems were equipped with 1GB of main memory, a common 7200RPM hard disk for operating system and DBMS software, and a 64GB Intel X25-E SSD on which a partition of 40GB contained the database data including temporary files. Initially PostgreSQL was configured with *pgtune*[5] using the datawarehousing profile with up to 30 sessions (required for loading the data).

As workload we used the DBT-3 benchmark, which is an open source implementation of the TPC-H specification [16]. It is not validated by the *Transaction Processing Performance Council (TPC)*, so its results are not guarranteed to be comparable with validated TPC-H results. For the purposes of our experimental analysis the benchmark's well-defined schema and data set, as well as the standardized set of queries suffice. TPC-H models a data warehousing or decision support scenario, so most of the queries are highly demanding. We partially modified the original benchmark control scripts to fit the needs of our testing procedure.

Data was generated for scale factor 5. That is about 5 GB raw data. When loaded into the PostgreSQL database this inflates to about 13 GB through the addition of indexes and statistics. So less than 10% of that database fits in memory, which is a typical ratio in large scale analytical systems. We used the default page size of 8KB.

[5]`http://pgfoundry.org/projects/pgtune`

As performance metric we use the sum of the execution times of the benchmark's read only queries. However, we exclude the time to process query 15, because it contains a view creation, for which PostgreSQL cannot output a plan description, which is needed for our optimization. This sum of the 21 execution times will be called "total time" in the rest of the paper. We do not use TPC-H's original geometric mean based metric, because it is not suitable for speed-up calculation and generally harder to compare (see also [5]).

4.2 Calibration

The new cost model provides an extended set of configurable coefficients within its functions. A fair way to demonstrate the improved configurability and its positive impact on the processing performance is to test both models with their individual optimal configuration. Unfortunately, the optimal parameter settings cannot be computed analytically because they depend on algorithmic operations as well as on unknown operating system and hardware properties. Additionally, there is also a virtually infinite space in which the settings have to be found, so an exhaustive evaluation is impossible, too. To still find very good settings for both models, we used a heuristic based on *simulated annealing*[9]. We instructed that search algorithm to find the configuration that minimizes the "total time" of the DBT-3 benchmark.

The implemented search algorithm repeatedly changes the configuration settings and runs the load; it observes a response time, and if that is lower than the previous current execution time, it accepts the new settings; if the new response time is higher it randomly chooses whether to accept it nevertheless or discard it. The probability for the acceptance of an inferior setting decreases over time as well as the average strength of setting modification. To prevent the algorithm from getting caught in a non-global local optimum, the modification strength level and acceptance probability is reset after a fixed number of iterations and the algorithm starts again with the currently best known settings while having the chance to escape it with a big step. Figure 3 in the appendix shows the parameter settings as they vary over time and the corresponding total execution time for the first cycles of calibration. The repeated increase of variance in the curves is caused by the mentioned restarts. Modification of the configuration settings is performed by multiplication with a logarithmic normal-distributed random variable to respect the totally different magnitudes of the parameter values and to prevent autonomous drift.

For the final comparison of the models we executed the benchmark with 25 varying random seeds different from the seed used for calibration. The seed value essentially modifies the selectivity of the queries and sub-queries. Therefore the optimizer may consider different plans as being the optimal strategy to answer the given query templates. The

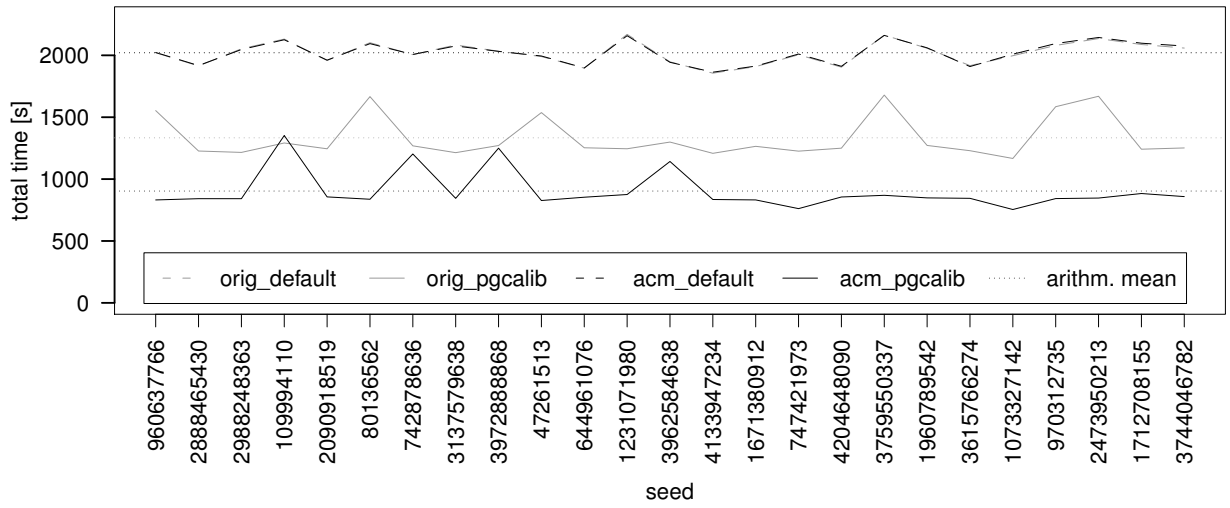


Figure 1: Total execution time of DBT-3 benchmark (excluding query 15) [orig = original model, default = initial pgtune'd configuration, acm = new asymmetric cost model, pgcalib = calibrated configuration that delivered minimal totaltime for a given model and the workload]. Different random seeds lead to different selectivities of the qualifications in the queries.

same 25 seeds are used to test (i) the original model with our data warehousing default configuration ('orig_default'), (ii) the calibrated original model ('orig_pgcalib'), (iii) the modified new model with a default configuration containing a converted set of the original weight factor settings ('acm_default'), and (iv) the new model with calibrated parameter settings ('acm_pgcalib').

4.3 Results

Figure 1 shows the total execution time for the different seeds, settings, and models. Finer grained information is shown in Figure 2 compares only the two calibrated configurations and shows the geometric mean of the speed-ups each query template received.

4.4 Discussion

From the chart in Figure 1 it is easily visible that the calibrated models both perform better than the same models with their initial ("default") configuration. Furthermore, the system with the calibrated new *asymmetric cost model* ("acm") completes the benchmark in even less time. There is a single case where the system with the new model seems to be a little bit slower and there are three additional peaks where the difference is not as prominent. Wilcoxon-Mann-Whitney's U-test [17, 12] computes a very very low probability (0.00001788%) that the results of both calibrated models might originate from the same distribution. The uncalibrated models, however, look very congruent and for these numbers the U-test delivers a probability of 20.99%. So one can not refute that they originate from the same distribution with a typical significance level of 5%.

The speed-up values seen in Figure 2 show that 14 out of 21 tested individual queries receive a positive speed-up while the performance of the others is only a litte bit decreased. The speed-up for the various query templates is very non-uniform. While query 21 gains more than 500% speed-up there are a lot of queries with much lower speed-up values and even one query whose speed is now 4% lower. Of the 21 queries (Q15 is excluded, see above), there are 10 queries that gain at least 5%. The average speed-up with respect to the total execution time of the 21 queries is 48%.

We did an in depth analysis of query 21 which had the highest relative speed-up and query 9 whose runtime difference was the biggest. Their detailed timing data reveals that in both cases the faster plan is doing index-nested-loop joins where the join attribute values change in the same order as the corresponding tuples are stored in the index-accessed table of the inner plan tree. In such a case the index scan results in sequential storage accesses. Although the slower plans used the same index, the join attribute values do not change in table order. In this case storage accesses are randomized what results in a reduced cache-hit rate. So the faster plan exploits a data-dependent inter-table correlation. Problematically, such inter-table correlations are not respected in PostgreSQL's optimizer at all. It only accounts for index-to-table correlations which are only relevant for range scans. Other reasons for speed-up are much less prominent or are hard to grasp because of unstable plan choices. Clearly device related reasons are thus hidden and it may be possible that the additional degrees of freedom got abused to compensate general optimizer deficiencies.

We performed a quick cross check on common HDDs. In a very time constrained experiment comprising only one cooling cycle of 100 iterations the calibration could not provoke a significant speed-up difference between the two models. With calibrated settings both systems perform the benchmark about 120% faster than with default settings. However, using the calibrated settings originally obtained for the SSDs, query 21 runs about 25 times faster (280s instead of 7227s) indicating that the optimal parameters for the HDDs were not found during this short calibration. With the same settings the whole benchmark is performed in 11895s using the old model and in 4975s using the new model – in both cases faster than with the quickly calibrated settings.

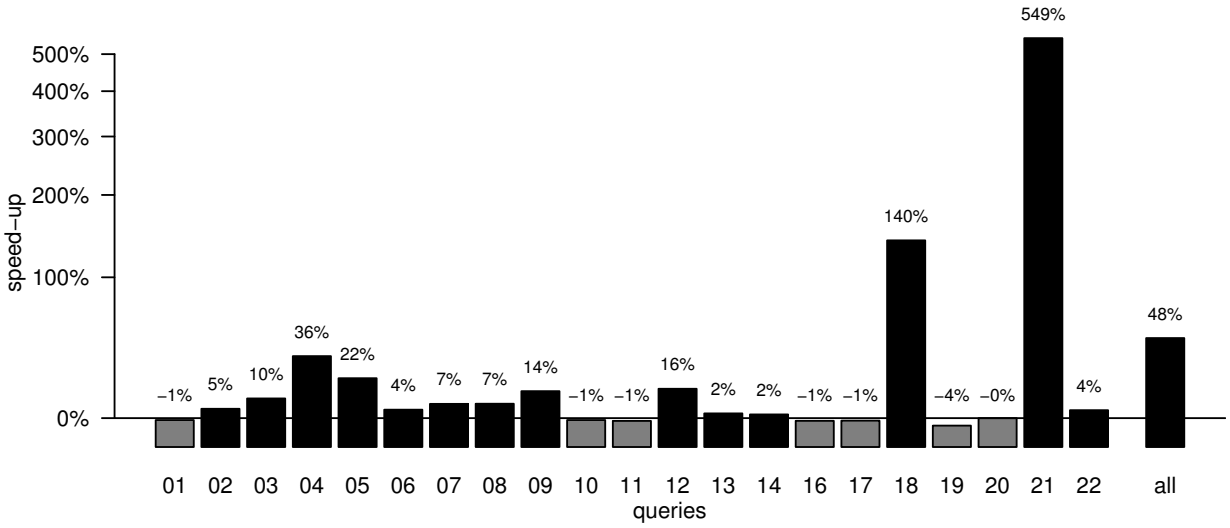


Figure 2: Speed-up from optimized old model to optimized new model per TPC-H query. [black = system with new model performs better, gray = system with old model performs better]

5. CONCLUSION

We presented a new cost model for PostgreSQL's cost-based query optimizer that distinguishes read and write operations. From our experimental results we conclude that this model can indeed deliver a higher performance if its weight parameters are configured to reflect the system and data properties. A tool was built and described that automates the configuration process.

We see significant performance improvements of an application class benchmark using the new model with calibrated parameters. However, we cannot relate the concrete plan changes observed in the experiments conducted so far to original SSD properties. The additional degrees of freedom available in the new model may be even useful to better tune the optimizer on systems based on common hard disks. With a simpler workload like the ones used in [1, 14] containing only data-dependencies respected by the used optimizer there might be a chance to explicitely demonstrate asymmetry-awareness. Such experiments are planned as a future work.

6. ACKNOWLEDGEMENTS

This work has been partially supported by the DFG project Flashy-DB. We thank Steven Pelley and Thomas Wenisch for their detailed and constructive critique of our work.

7. REFERENCES

[1] D. Bausch, I. Petrov, and A. Buchmann. On the performance of database query processing algorithms on flash solid state disks. In *FlexDBIST'11*, 2011.

[2] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS '98*, pages 34–43. ACM, 1998.

[3] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of SIGMETRICS '09*, pages 181–192, 2009.

[4] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CDIR'11*, Asilomar, CA, Jan. 2011.

[5] A. Crolotte. Issues in benchmark metric selection. *LNCS*, 5895:146–152, 2009.

[6] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *Proc. DaMoN 2009*, 2009.

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[8] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28:121–123, March 1996.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[10] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[11] L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated I/O model. *ACM Trans. Database Syst.*, 14(3):401–424, 1989.

[12] H. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[13] OSDLDBT Project. Database test 3 (dbt-3). `http://osdldbt.sourceforge.net`.

[14] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be ssd-aware? In *ADMS'11*, 2011.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. SIGMOD '79*, pages 23–34. ACM, 1979.

[16] Transaction Processing Performance Council. TPC benchmark H – standard specification. `http://www.tpc.org/tpch`.

[17] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

APPENDIX

A. TRANSCRIPT OF COST MODEL

This appendix contains a mathematical transcript of the cost model. These formulas are not required to understand the presented work as there are enough explanations in the main matter. Nevertheless, we include them for completeness or a brief overview. Table 2 displays the meaning of the used mathematical symbols.

Table 2: Notation

notation	meaning
c_{io}	compound cost value for I/O operations
$\hat{c}_{\text{io}}$	compound I/O cost specific to index type
$\mathbf{c_s}$	cost weight factor for sequential accesses
$\mathbf{c_r}$	cost weight factor for random accesses
$\mathbf{c_{sw}}$	cost weight factor for sequential page writes
$\mathbf{c_{rw}}$	cost weight factor for random page writes
$\mathbf{c_{sr}}$	cost weight factor for sequential page reads
$\mathbf{c_{rr}}$	cost weight factor for random page reads
$\dot{\mathbf{c}}_{\mathbf{cpu}}$	CPU cost for loading a single tuple
$\hat{\mathbf{c}}_{\mathbf{cpu}}$	additional CPU cost for index based acceses
$\mathbf{c_{op}}$	CPU cost for applying an operator
$\mathbf{e}$	total cache available to database
$\lVert\cdot\rVert_{\text{t}}$	number of tuples in an object
$\lVert\cdot\rVert_{\text{p}}$	number of pages required to store an object
$\lVert\cdot\rVert_{\text{l}}$	number of entries in the argument array
$\mathcal{R}$	all physical tables of database
R	base relation associated with algorithm
O	sibling outer relation in a nested loop join
P_i	relation produced by the inner deeper path
P_o	relation produced by the outer deeper path
S	to be sorted data
$\hat{r}$	correlation coefficient of index to table
$\hat{s}$	selectivity of index associated quals
$\hat{s}_B$	selectivity of index boundary quals
m	the *merge order* of the *polyphase merge sort*
n	the expected number of initial sorted runs

variable names shown in **bold** face are user configurable

For *sequential scan*, *index scan*, *bitmap scan*, *tuple-ID scan*, and *materialization* we show only the formulas of the original model and a replacement map (Table 3) because these formulas do not change structurally, i.e. only symbol names change. For *sort* and *hash join* contrarily we will show both, the old and the new function, at length.

Table 3: Parameter replacements in cost functions

function	original	new	affected equations
seqscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(1)
indexscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(4)
	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(3), (4), (16)
bitmapscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(13)
	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(13), (16)
tidscan	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(25)
material	$\mathbf{c_s}$	$\mathbf{c_{sw}}$	(31)
rescan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(32)
sort	*more complex, see Section 3.3.1*		
hashjoin	*more complex, see Section 3.3.2*		

A.1 Sequential Scan

Cost of a sequential scan is trivial: sequential page (read) cost times number of pages.

$$c_{\text{io}}(seqscan) = \mathbf{c_s} \left\lVert R \right\rVert_{\text{p}} \tag{1}$$

A.2 Index Scan

I/O cost of the index scan is calculated by a rather complex system of formulas. Trivially speaking, they interpolate between the sequential and random cost factors based on selectivity and index-to-table order correlation.

$$c_{\text{io}}(indexscan) = \hat{c}_{\text{io}} + \overline{c}_{\text{io}} + \hat{r}^2 \left(\underline{c}_{\text{io}} - \overline{c}_{\text{io}}\right) \tag{2}$$

$$\overline{c}_{\text{io}} = \mathbf{c_r} \cdot \hat{p}(i_o t) \cdot i_o^{-1} \tag{3}$$

$$\underline{c}_{\text{io}} = \begin{cases} \mathbf{c_r} \cdot \hat{p}\Big(\lceil i_o \hat{s} \left\lVert R \right\rVert_{\text{p}} \rceil\Big) \cdot i_o^{-1} & \text{if } i_o > 1 \\ \mathbf{c_r} + \mathbf{c_s} \cdot \Big(\lceil \hat{s} \left\lVert R \right\rVert_{\text{p}} \rceil - 1\Big) & \text{else} \end{cases} \tag{4}$$

$$t = \text{round}\big(\max\big(1.0,\ \hat{s} \left\lVert R \right\rVert_{\text{t}}\big)\big) \tag{5}$$

$$i_o = \max\big(\left\lVert O \right\rVert_{\text{t}},\ 1\big) \tag{6}$$

To estimate the number of page fetches required for the index scan the function rooted at Equation (7) from [11] is used. (In the original paper $\hat{p}$ is named Y_{APP}. PostgreSQL replaces the product Dx by `tuples_fetched` which is t in our symbol symbol system.)

$$\hat{p}(t) = \begin{cases} \min\Big(p_t,\ \left\lVert R \right\rVert_{\text{p}}\Big) & \text{if } \left\lVert R \right\rVert_{\text{p}} \le e_R \\ p_t & \text{else if } t \le p_e \\ e_R + (t - p_e)\,\frac{\left\lVert R \right\rVert_{\text{p}} - e_R}{\left\lVert R \right\rVert_{\text{p}}} & \text{else} \end{cases} \tag{7}$$

$$p_t = \frac{2 \left\lVert R \right\rVert_{\text{p}} t}{2 \left\lVert R \right\rVert_{\text{p}} + t} \tag{8}$$

$$p_e = \frac{2 \left\lVert R \right\rVert_{\text{p}} e_R}{2 \left\lVert R \right\rVert_{\text{p}} - e_R} \tag{9}$$

$$e_R = \mathbf{e} \cdot \frac{\left\lVert R \right\rVert_{\text{p}}}{\left\lVert \mathcal{R} \right\rVert_{\text{p}} + \left\lVert I \right\rVert_{\text{p}}} \tag{10}$$

A.3 Bitmap (Index) Scan

The I/O cost function of the *bitmap (index) scan* treats off the random and sequential factors based on the fraction of pages estimated to be needed from the relation. The function $\hat{p}(\cdot)$ from Equation (7) is reused in this context.

$$c_{\text{io}}(bitmapscan) = \overbrace{\hat{c}_{\text{io}}}^{\text{startup}} + pc' \tag{11}$$

$$p = \begin{cases} \min\left(\frac{\hat{p}\left(t \left\lVert O \right\rVert_{\text{t}}\right)}{\left\lVert O \right\rVert_{\text{t}}}, p_R\right) & \text{if } \left\lVert O \right\rVert_{\text{t}} > 1 \\ \min\left(\frac{2 p_R t}{2 p_R + t}, p_R\right) & \text{else} \end{cases} \tag{12}$$

$$c' = \begin{cases} \mathbf{c_r} - (\mathbf{c_r} - \mathbf{c_s}) \sqrt{\frac{p}{p_R}} & \text{if } p \ge 2 \\ \mathbf{c_r} & \text{else} \end{cases} \tag{13}$$

$$p_R = \max\Big(\left\lVert R \right\rVert_{\text{p}}, 1\Big) \tag{14}$$

$$t = \text{round}\big(\max\big(1.0,\ \hat{s} \left\lVert R \right\rVert_{\text{t}}\big)\big) \tag{15}$$

A.4 General B$^+$-Tree Functions

PostgreSQL supports different kinds of index structures. Therefore the cost functions for both index based algorithms call an abstract function to calculate the cost for accessing only the index structures. In the used version of PostgreSQL, all included index types base their cost function on a generic index cost function. Therefore we only present a cost function for the default B$^+$-tree index as it contains everything relevant for the other types of index, too.

$$\hat{c}_{\text{io}} = \begin{cases} \frac{p\mathbf{c_r}}{i_o} & \text{if } i > 1 \\ p_I \mathbf{c_r} & \text{else} \end{cases} + p_I \frac{\mathbf{c_r}}{100000} \tag{16}$$

$$i_o = \max\left(\|O\|_{\text{t}}, 1\right) \tag{17}$$

$$i = i_{sa} \cdot i_o \tag{18}$$

$$i_{sa} = \prod_{a \in \hat{Q}_{sa}} \|a\|_{\text{l}} \tag{19}$$

$$\hat{Q}_{sa} := \{q \in \hat{Q} \mid q \text{ is a scalar array operator}\} \tag{20}$$

$$p_I = \begin{cases} \left\lceil t_I \frac{\|I\|_{\text{p}}}{\|I\|_{\text{t}}} \right\rceil & \text{if } \|I\|_{\text{p}} > 1 \wedge \|I\|_{\text{t}} > 1 \\ 1 & \text{else} \end{cases} \tag{21}$$

$$t_I = \begin{cases} 1 & \text{if uniqe index } \wedge \\ & \text{only '=' quals } \wedge \\ & \text{no scalar array op } \wedge \\ & \text{no null test} \\ \text{round}\left(\hat{s}_B \frac{\|R\|_{\text{t}}}{i_{sa,B}}\right) & \text{else} \end{cases} \tag{22}$$

$$i_{sa,B} = \prod_{a \in \hat{Q}_{sa,B}} \|a\|_{\text{l}} \tag{23}$$

$$\hat{Q}_{sa,B} := \{q \in \hat{Q}_{sa} \mid q \text{ is an index boundary qual}\} \tag{24}$$

A.5 Tuple-ID Scan

The cost function for *tuple-ID scan* counts for every tuple access one page access. However, if it is combined with a "scalar array operator", e.g. an "in (...)" expression, then it counts a page access for every array entry.

$$c_{\text{io}}(tidscan) = t\mathbf{c_r} \tag{25}$$

$$t = \sum_{q \in Q} \begin{cases} \|q\|_{\text{l}} & \text{if } q \in Q_{sa} \\ 1 & \text{if } q \in Q_{co} \cup Q_{ctid} \\ 0 & \text{else} \end{cases} \tag{26}$$

$$Q : \text{all quals associated to the tidscan} \tag{27}$$

$$Q_{sa} := \{q \in Q \mid q \text{ is a scalar array qual}\} \tag{28}$$

$$Q_{co} := \{q \in Q \mid q \text{ is a 'current of' expr}\} \tag{29}$$

$$Q_{ctid} := \{q \in Q \mid q \text{ is a 'ctid = ...' expr}\} \tag{30}$$

A.6 Materialization and Re-Scan

Writing out an intermediate result and re-reading it from disk is counted with the same formula in the original model. However, after applying the replacements from Table 3 *materialization* uses the "sequential write" factor while *re-scan* uses the "sequential read" factor.

$$c_{\text{io}}(material) = \|P_i\|_{\text{p}} \mathbf{c_s} \tag{31}$$

$$c_{\text{io}}(rescan) = \|P_i\|_{\text{p}} \mathbf{c_s} \tag{32}$$

A.7 Sort

The I/O cost of PostgreSQL's external sort is computed by a typical $O(n \log n)$ formula. The leading factor in its original form accounts for writing plus reading. In the new asymmetry-aware (rw) form this is more differentiated.

The term $\lceil log_m n \rceil$ estimates the number of required sort phases, where m is the merge order and depends on the available memory. The number of expected initial sorted runs n was previously calculated under the assumption that the initial sorted runs grow to two times the memory slice available to the algorithm. They are generated by a heap sort. As all tuples need to be sorted before the algorithm can output the first, everything is counted as *startup cost*.

$$c_{\text{io}}(sort) = \overbrace{2 \|S\|_{\text{p}} \lceil log_m n \rceil \left(\frac{3}{4}\mathbf{c_s} + \frac{1}{4}\mathbf{c_r}\right)}^{\text{startup}} \tag{33}$$

$$c_{\text{io}}(sort)_{rw} = \overbrace{\|S\|_{\text{p}} \left(\mathbf{c_{sw}} + \left(\lceil log_m n \rceil - 1\right) \frac{\mathbf{c_{sw}} + \mathbf{c_{rw}}}{2}\right.}^{\text{startup}} \underbrace{\left. + \lceil log_m n \rceil \frac{\mathbf{c_{sr}} + \mathbf{c_{rr}}}{2}\right)}_{\text{startup}} \tag{34}$$

A.8 Hash Join

The cost function accounts for writing and reading all data one time each. In the original form accesses are counted as sequential, which does not match the real behavior. Our asymmetry-aware form corrects that and counts the write operations for batching as random writes.

$$c_{\text{io}}(hashjoin) = \overbrace{\|P_i\|_{\text{p}} \mathbf{c_s}}^{\text{startup}} + \left(\|P_i\|_{\text{p}} + 2\|P_o\|_{\text{p}}\right) \mathbf{c_s} \tag{35}$$

$$c_{\text{io}}(hashjoin)_{rw} = \overbrace{\|P_i\|_{\text{p}} \mathbf{c_{rw}}}^{\text{startup}} + \|P_i\|_{\text{p}} \mathbf{c_{sr}} + \|P_o\|_{\text{p}} \left(\mathbf{c_{rw}} + \mathbf{c_{sr}}\right) \tag{36}$$

B. CALIBRATED PARAMETER SETTINGS

Table 4 shows the parameters settings used for the experiments. They were obtained using the calibration algorithm described in Section 4.2.

Table 4: Optimal settings determined by calibration

old model	new model
$\mathbf{c_s} = 1.00000$	$\mathbf{c_{sr}} = 1.00000$
	$\mathbf{c_{sw}} = 49.91840$
$\mathbf{c_r} = 6.77405$	$\mathbf{c_{rr}} = 5.62724$
	$\mathbf{c_{rw}} = 19.08421$
$\dot{\mathbf{c}}_{\mathbf{cpu}} = 0.00121$	$\dot{\mathbf{c}}_{\mathbf{cpu}} = 0.00003$
$\hat{\dot{\mathbf{c}}}_{\mathbf{cpu}} = 0.03658$	$\hat{\dot{\mathbf{c}}}_{\mathbf{cpu}} = 0.01608$
$\mathbf{c_{op}} = 0.00016$	$\mathbf{c_{op}} = 0.00008$

For symbol explanation see Table 2.

Appendix B.1 shows a graphical plot of the first 1000 calibration cycles. The parameters shown in Table 4, however, are (one of) the best seen configurations after about 11000 cycles.

B.1 Calibration Dynamics

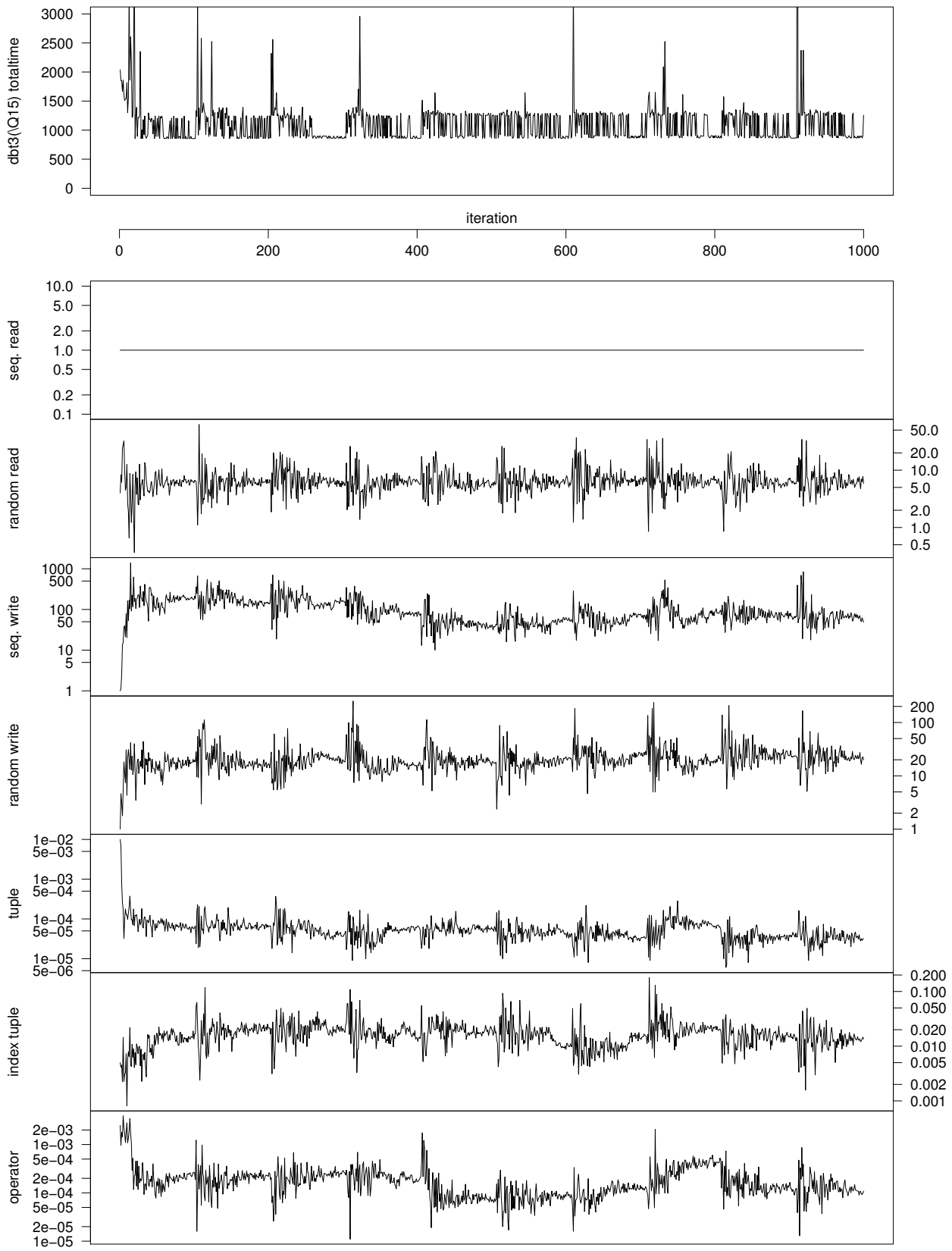


Figure 3: First 1000 iterations of calibration of new model

Hathi: Durable Transactions for Memory using Flash

Mohit Saxena
U. Wisconsin-Madison
msaxena@cs.wisc.edu

Mehul A. Shah, Stavros Harizopoulos
Nou Data
{mashah,stavros}@gmail.com

Michael M. Swift
U. Wisconsin-Madison
swift@cs.wisc.edu

Arif Merchant
Google
aamerchant@google.com

ABSTRACT

Recent architectural trends — cheap, fast solid-state storage, inexpensive DRAM, and multi-core CPUs — provide an opportunity to rethink the interface between applications and persistent storage. To leverage these advances, we propose a new system architecture called Hathi that provides an in-memory transactional heap made persistent using high-speed flash drives. With Hathi, programmers can make consistent concurrent updates to in-memory data structures that survive system failures.

Hathi focuses on three major design goals: ACID semantics, a simple programming interface, and fine-grained programmer control. Hathi relies on software transactional memory to provide a simple concurrent interface to in-memory data structures, and extends it with persistent logs and checkpoints to add durability.

To reduce the cost of durability, Hathi uses two main techniques. First, it provides *split-phase* and *partitioned commit* interfaces, that allow programmers to overlap commit I/O with computation and to avoid unnecessary synchronization. Second, it uses partitioned logging, which reduces contention on in-memory log buffers and exploits internal SSD parallelism. We find that our implementation of Hathi can achieve 1.25 million txns/s with a single SSD.

1. INTRODUCTION

Transactions serve two purposes that make it easier to build robust applications. First, transactions enable fine-grained concurrency control, allowing developers to scale applications more easily across multiple processors [8, 20]. Second, transactions provide a simple interface for managing the durability and consistency of application state in the face of failures [16].

However, the common interfaces to ACID transactions – DBMSs or distributed transaction systems – are a poor fit for many modern workloads, partially due to the cost and complexity of managing such systems. As a result, many applications, particularly web services, sacrifice strong consistency for simpler storage models, such as key-value stores [14]. We believe that a key challenge of existing transaction interfaces is that they require the use of large, complex systems (the DBMS or distributed transaction coordinator) even for lightweight operations. In contrast, most file systems reject transactional semantics and provide only coarse-grained control over durability in the form of flushing pages or files to disk.

In this paper, we describe a lightweight, high-performance and ACID-compliant transactional store based on *durable memory transactions*: programs can concurrently update in-memory data structures that are consistent and persistent across failures. We leverage three recent architectural trends in the design. First, DRAM prices have dropped to a point that even mid-tier servers support up to 4 TB of memory. At these sizes, many workloads can execute in core rather than from disk — an observation also made by others [19, 23]. Second, power considerations have driven processor manufacturers away from uni-processors towards multi-core chips, so concurrency between threads becomes a key concern. Third, flash-based solid-state drives (SSDs) provide scalable bandwidth and 1-2 orders of magnitude lower latency than the fastest disks.

Hathi is a high-speed, durable, main-memory transactional store that leverages these trends. It presents an in-memory transactional heap interface that is automatically made persistent on fast SSDs. Thus, programs can create and manipulate in-memory data structures, but ensure the data is durable with little extra effort. To do so, Hathi combines the simple and highly concurrent interface ("ACI") of transactional memory [20] with an SSD-optimized write-ahead logging and checkpointing scheme for durability ("D"). Thus, a programmer can wrap a section of program in a transaction to make updates durable and consistent. As flash storage is fast but still much slower than memory, Hathi implements two approaches to reduce and eliminate the overhead of persistence.

First, Hathi provides options at transaction commit to control whether the program blocks, similar to but more fine-grained than asynchronous file I/O. This control has not previously been available for memory transactions, and allows developers to leverage application knowledge for increased performance. Specifically, Hathi exposes *split-phase* commit, which decouples the installation of in-memory updates from the flush of transaction log records. Using this interface, applications can continue with other tasks and later check for completion of the commit, thereby overlapping computation and commit I/O.

Second, Hathi uses partitioned logging, in which each thread maintains a separate log. Partitioned logging leverages the SSD's internal parallelism and avoids contention on in-memory log buffers that hold the tail of the transaction log. For full consistency, Hathi ensures that all preceding transactions from all threads are durable before marking a transaction as durable. However, Hathi also provides *partitioned commit*, which allows application threads to commit transactions that operate on independent data structures without coordinating with the logs from other threads. With these interfaces, applications retain the recoverability guarantees of synchronous commit at speeds closer to asynchronous commit. Our

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

experiments with these interfaces show 4-5x throughput improvements over synchronous commit.

With these optimizations, we show that Hathi reaches 1.25 million txns/sec on a high-end FusionIO drive and nearly 200 K txns/sec on a consumer-grade Intel X-25M SSD. Thus, Hathi provides durable transactions at little additional cost over non-durable transactional memory. At these speeds, we believe Hathi is suitable for building not only user-facing applications, but also infrastructure applications like file systems, key-value stores, massively multiplayer online games and social-network graphs.

Persistent memory [24, 29] and persistent object stores [27, 30] have been proposed in the past. However, Hathi is the first to rely on commodity processor and storage technology (i.e., no phase-change memory or battery-backed DRAM), and operate at near-DRAM speed, courtesy of its interfaces for split-phase and partitioned commit. In addition, Hathi provides a low-level unstructured memory space that makes few demands of the programmer, allowing it to be used as a building block for these systems. In summary, Hathi provides a simple application interface, fast and scalable mechanisms for transaction commit, and fine-grained durable memory transactions.

2. BACKGROUND

Hathi is enabled by recent developments in solid-state drives (SSDs). Internally, SSDs are comprised of multiple flash chips accessed in parallel. As a result, they provide scalable bandwidths limited mostly by the interface between the drive and the host machine (generally SATA or PCIe) [7, 11]. In addition, SSDs provide access latencies orders of magnitude faster than traditional disks. For example, FusionIO enterprise ioDrives provide 25 μs read latencies and up to 600 MB/sec throughput [1]. Consumer grade SSDs provide latencies down to 50-85 μs and bandwidths up to 250 MB/sec [18]. To fully saturate the internal bandwidth of the device, enterprise SSDs support longer I/O request queues.

With current technology, SSD bandwidth is comparable to or exceeds network bandwidth (1.9 Gb/s read, 0.7 Gb/s write for a commodity SSD [18]). However, the performance of random writes may be much lower, consumer-grade devices can only provide 3000-8000 random writes per second. Thus, sustaining fine-grained updates at full network latency requires mechanisms to tolerate the write latency and queueing delays from prior writes.

3. INTERFACE AND DURABILITY OPTIONS

Hathi provides programmers with a familiar set of simple primitives that facilitates them to build fast, robust, and flexible persistent memory regions. Rather than forcing programs to use low-level file primitives or convert their data into a database format, Hathi enables a program to use any in-memory data structure for durable data with *persistent heaps*. Heaps are persistent memory regions that applications can read or write using a software transactional memory (STM)-like interface. Hathi provides a `pmalloc` interface to create a heap, which allocates a segment of memory and associates it with a checkpoint file on an SSD. A program can then perform consistent reads and writes to the heap using the interface shown in Figure 1. A read from a given heap address and size copies the memory region to a user-specified buffer, and a write to a heap updates an in-memory copy of the data. A transaction can abort, which erases all updates, or commit, which makes updates persistent and visible to other threads.

Persistent memory is an invaluable asset for applications that require both high throughput and durability. Some such applications include file system journaling, high-throughput main memory key-value stores, social network graph databases, main memory transaction processing systems, persistent logs for highly available network servers, and massively multiplayer online games [10, 28].

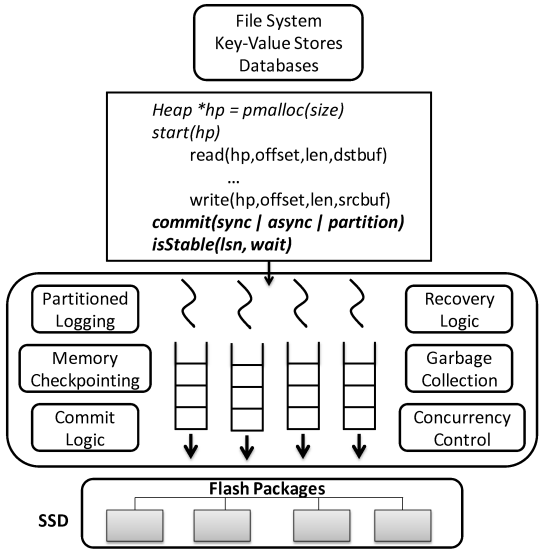


Figure 1: Hathi architecture and interface.

```
hash_insert(hashtable, key, value) {
    txn_start(hashtable);
    if( txn_read(hashtable->entries)
            > txn_read(hashtable->loadlimit) ) {
        hashtable_expand(hashtable);
    }
    bucket = pmalloc(sizeof(*bucket));
    hash = getHash(hashtable, key);
    index = indexFor(hashtable->length, hash);
    txn_write(bucket->key, key);
    txn_write(bucket->value, value);
    txn_write(bucket->next, hashtable->table[index]);
    txn_write(hashtable->table[index], bucket);
    txn_commit(sync);
}
```

Figure 2: Hathi hash table insert: example usage of memory transactions.

For example, Figure 2 demonstrates how to update a hash table within a transaction. The transaction encompasses both data reads, to provide concurrency control, memory allocation, and updates. At transaction commit, a runtime system ensures the updates are consistent (i.e., no interleaving with other transactions) and durable. As the transaction executes, Hathi writes the updates to a per-thread log, and then forces the log to an SSD to make the update durable.

File systems and databases provide disk-based journaling capabilities for transactional updates. Hathi provides durability for main

DEPENDENTTRANSACTIONS

START(hp)	START(hp)
READ(hp,0,10,buf)	READ(hp,11,10,buf)
for(alli)	for(alli)
buf[i]+=1	buf[i]=2
WRITE(hp,11,10,buf)	WRITE(hp,0,10,buf)
COMMIT **sync**	COMMIT **sync**

PARALLELTRANSACTIONS

START(hp)	START(hp)
READ(hp,0,10,buf)	READ(hp,11,10,buf)
for(alli)	for(alli)
buf[i]+=1	buf[i]=2
WRITE(hp,0,10,buf)	WRITE(hp,11,10,buf)
COMMIT **partition**	COMMIT **partition**

Figure 3: Dependent and parallel transactions.

memory transactions through a log stored on an SSD. Although flash latencies are low, they are much larger than latencies to main memory. So, programmers still must be careful about when to wait for updates to become durable. Hathi provides programmers with three options to commit that control its durability guarantee: *sync*, *async*, and *partition*.

Sync and Async. *Sync* and *async* are similar to database synchronous and asynchronous commit and the `fsync()` file system operation. Synchronous commit only returns after forcing the transaction's log records *and* the records of all preceding transactions' to stable storage. This ensures that the update is durable and application data structures are consistent, because all prior updates were also written to the SSD. In contrast, asynchronous commit returns as soon as the transaction finishes updating memory, and does not wait to force the log records to storage. Even with this option, the heap recovers to a consistent point in the transaction history, although it may lose recently committed asynchronous transactions.

Partition. Commit's third option, *partition*, relaxes the isolation guarantee for better performance. Hathi has a separate log partition for each thread (see Section 4). Partition commit simply forces the log for the local thread. This option is useful when an application uses transactions for atomicity and durability but *not* for isolation, and may be used when each application thread operates on different (partitioned) data. Such partitioning may be easy when updating regular data structures such as a hashtable or a matrix.

This commit option potentially allows for more overlap between computation and I/O across threads than synchronous commit, because a thread need not wait for preceding transactions in other threads to become stable. Although applications can mix this option with the others above, they must be careful to ensure the independence of partition commit. Figure 3 shows an example of dependent transactions, which cannot use partition, and parallel ones that can.

isStable. Hathi provides an additional interface to query whether a prior asynchronous transaction is durable. On success, async commit returns the logical sequence number (LSN) for the transaction. The *isStable(lsn,wait)* call indicates whether a transaction with that LSN is stable and recoverable, and optionally waits until it is. A transaction is recoverable if all transactions it is dependent on are durable, which may be all preceding transactions. This interface allows applications to make commit *split-phase*: initiate commit early, and then wait for it to complete later. For example, an event-driven server can continue with other client transactions and return results once the log flushes. In this way, Hathi can overlap I/O and computation, getting recoverability at nearly the same throughput as asynchronous commit.

4. DESIGN AND IMPLEMENTATION

Unlike traditional databases, Hathi does not maintain a backing store: storage contains only logs and checkpoints, and the logs contain only redo records of updates. Checkpoints are copies of the heap kept on an SSD, and allow trimming the logs to reduce recovery time. Hathi checkpoints the heap incrementally to avoid stalling the system.

We implemented a prototype of Hathi by modifying an existing software transactional memory system (TinySTM [15]), used for concurrency control, to add write-ahead logging and recovery for durability. Hathi's transaction API wraps the underlying STM calls, which maintain a log of updates to apply when a transaction completes. The STM ensures transactions are isolated by acquiring locks on each memory location referenced, and will abort one or more transactions when it detects conflicting lock request. On successful commit, Hathi tags the transaction with a logical sequence number (LSN) given by the STM and inserts them into the thread's log. The LSN is a global counter that the STM atomically increments before releasing all locks, thus ordering all transactions. For all commit types, Hathi reflects the transaction updates in-memory and releases locks before the log records reach the SSD to allow other threads to proceed.

Partitioned Logging. Hathi employs *partitioned logging* both in memory and in storage: each core maintains its own transaction log that can be independently flushed to a separate location in storage. Merging the logs provides a logical global log. Partitioned logging is well suited to both SSDs and multi-core architectures: SSDs require multiple outstanding requests to saturate their bandwidth, and partitioned logging allows multiple cores to generate requests simultaneously. In addition, partitioned logging reduces lock contention, since threads access their local log without synchronization. Hathi further reduces latency with direct I/O to bypass the file-system buffer pool.

Partitioned logging complicates recovery by raising the possibility that later transactions from one thread will become durable before earlier transactions of another. This potentially leaves a gap in the transaction sequence on failure, resulting in an inconsistent application data structure. During recovery, it may therefore be inconsistent to replay all committed transactions in all logs. On recovery, Hathi takes care to only replay transactions up to the first missing transaction.

Thus, Hathi maintains two invariants that tie together logging and checkpointing for correctness of its recovery algorithm. First, each transaction has an LSN that is consistent with the partial order of transaction dependencies; a transaction can only depend on transactions with lower LSNs. When logging, a transaction is not recoverable until all preceding transactions in this total order are recoverable. Second, to ensure that all updates are consistently applied to a checkpoint, we must maintain the *write-ahead logging discipline*: a chunk of memory in a checkpoint cannot be used for recovery until the effects of all transactions reflected in that chunk have been made recoverable.

The Hathi interface enables applications to control durability of their data. Hathi maintains a global variable, `min_lsn`, that tracks the youngest recoverable transaction. Each transaction log maintains the latest LSN that is on the non-volatile store; the `min_lsn` is the minimum or oldest of these. Each thread updates this variable after flushing its log. For synchronous commit, Hathi flushes the local log and waits until `min_lsn` exceeds or equals the transaction LSN. To improve throughput, synchronous commit batches multiple transactions into a single log flush, a technique called *group commit* [17].

Partitioned commit annotates the transaction's log record with a *partition* flag, flushes the log, and does not wait for `min_lsn`. The flag indicates that the transaction can safely be recovered, even if preceding transactions from other threads are not available. Asynchronous commit also does not wait and, like group commit, defers forcing the log until either a fixed time period elapses or a fixed

amount of log space is used. Thus, partitioned commit provides the durability semantics of synchronous commit for the local log, and the performance of asynchronous commit across logs. Finally, `isStable` compares the given LSN against `min_lsn` and waits if necessary.

Algorithm 1 Hathi Memory Checkpointing

```
1: for chunk in heap do
2:     tx_start
3:     tx_read(chunk,copyBuffer)
4:     chunkLSN = tx_commit(async)
5: end for
6: isStable(lastChunkLSN,true)
7: update checkpoint header
8: sleep(timer)
```

Checkpointing and Log Trimming. In checkpointing, Hathi periodically writes memory in configurable fixed-sized *chunks* to the SSD. When a checkpoint is needed, a separate checkpoint thread walks through the heap and writes out chunks, with each write protected by an STM transaction. This method ensures consistency with concurrent transactions, without the need to pause execution because both chunks and log records use the same log sequence number space. Although non-intrusive, this incremental checkpointing method allows for a transaction to straddle a chunk that has been checkpointed and one that has not. In such case, a chunk of memory in a checkpoint cannot be used for recovery until the effects of all transactions reflected in that chunk have been made recoverable, that is, written to disk so they can be re-applied to chunks that do not include their effects. Thus, a checkpoint is not valid until all transactions reflected in any of its chunks have been made recoverable (similar to write-ahead logging). Once Hathi has created a valid checkpoint of the entire heap, it discards unneeded older checkpoints and garbage collects log records prior to the earliest chunk in the new checkpoint since their effects are already included.

Algorithm 1 describes checkpointing. Since the workload fits in main memory, we can safely require that enough storage space is available for more than two checkpoints. Thus, we need not ensure that all transactions are durable before starting the checkpoint, as the previous checkpoint can still be used for recovery. Hathi first copies data from the persistent heap into a temporary buffer using an STM read, and then writes out the chunk and its LSN to checkpoint space. Once all chunks have been written, it writes a checkpoint header that indicates what checkpoints and log records can be garbage collected.

Recovery. Hathi performs recovery by loading a checkpoint and then replaying logs. It reads the checkpoint header to find the LSN of the oldest checkpoint chunk. Starting with that chunk, Hathi replays logs and checkpoints in LSN order until it reaches the end of one thread's log and a gap in the LSNs, which indicates a missing transaction. It then continues to scan logs and replays records labeled *partition*, which can still be safely applied.

In summary, Hathi provides novel interfaces and mechanisms designed to provide durability for memory-resident datasets and optimized for new flash SSDs and multi-core processors. Partitioned and split-phase commit interfaces provide low-level control over latency to application programmers. Partitioned logging and recovery, chunk-based incremental checkpointing and log trimming mechanisms optimize for the performance of flash SSDs which possess fast random access and high internal parallelism.

5. EVALUATION

Our current implementation of Hathi supports partitioned logging, incremental checkpointing, and recovery. In this section, we present experiments that show: (i) the cost of durability, (ii) the value of partitioned logging, and (iii) the performance tradeoffs of different durability options.

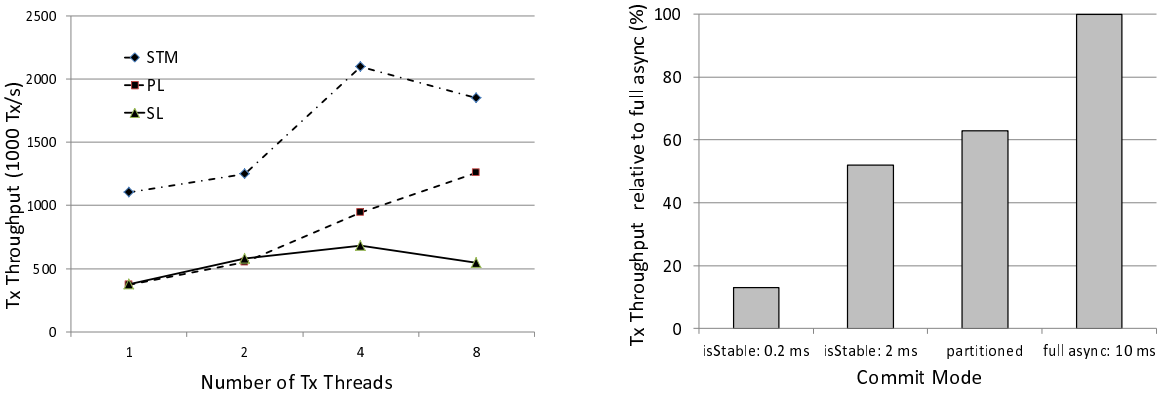
Methodology. We use two setups in these experiments. To emulate a high-end system, we use a 3.0 GHz Intel Xeon HP Proliant quad-core server with 8 GB DRAM and a 80 GB PCIe FusionIO ioDrive. On this, we run a synthetic workload of low-contention memory transactions for analyzing the overhead of providing durability. Each thread continuously executes transactions that read and write six random words in a 4 GB heap. The second system runs the travel reservation workload from STAMP transactional memory benchmark suite [22] that we ported to Hathi. This ran on a 2.5 GHz Intel Core 2 quad with 1 GB heap and a consumer-grade SSD, an Intel X-25M. We use a 10 ms group commit timer and maximum 512 KB log buffer for each thread.

Durability Costs. Figure 4(a) compares the performance of the high-end system running with Hathi with durable transactions using partitioned logging (PL) and asynchronous commit, against the base STM system without durability (STM). The STM system peaks in throughput at 4 threads with 100% CPU utilization. After 4 threads, the throughput drops because of increased contention over the STM's commit lock. With Hathi at 8 threads, we reach 1.25 M txns/sec with 70% CPU utilization, which is only 38% short of the peak STM throughput. Similarly, on the STAMP workload, Hathi reaches nearly 200 K txns/sec, only 15% short of the STM-only throughput. These results indicate that the added cost of durability is low for these workloads with Hathi. As a comparison, recent work on persistent memory using future projections of phase-change memory performance achieved only 1.3-1.6 M txns/sec with 4 cores [12, 29], not much better than Hathi in the high-end configuration.

Partitioned Logging. Figure 4(a) also compares the transaction throughput for single log (SL) and partitioned log (PL) on FusionIO. PL is more than 130% faster than SL, which utilizes less than 20% CPU. Thus, the single log is clearly the bottleneck. Single log performance degrades after 4 threads because of write serialization to ensure sequential I/O. In contrast, as we increase the number of threads and log partitions with PL, the throughput increases almost linearly. We note that partitioned logging benefits from more concurrent I/O requests and low access latencies of the storage device, as the seek costs in a disk would make partitioned logging more expensive.

We also investigate the impact of logging on the average transaction latency of the high-end system with asynchronous commit. With 4 threads on FusionIO, the average transaction latency for flash is only 14 ms. This is higher than the group-commit latency of 10 ms due to the queuing delay behind other log flushes. In comparison, using a single log raises latency to 25 ms, again demonstrating the value of partitioned logging in keeping latencies down.

Durability Tradeoffs. Using the STAMP workload, we investigate the performance impact of different durability options. Figure 4(b) shows the performance of split-phase and partitioned commit with different time intervals between calling `isStable` to wait for all previous transactions to become durable. As expected, when ensuring durability every 0.2 ms, transaction throughput is only 13% of asynchronous commit. However, allowing transactions to stay in memory for 2 ms achieves 50% of asynchronous performance. Using partitioned commit in addition to split-phase commit increases performance by 20% and is only 40% below asynchronous commit. These results demonstrate that the use of split-phase and partitioned



(a) Durability costs

(b) Commit mode performance

Figure 4: Durability costs and commit modes' performance.

commit to make a set of updates durable provide a middle-ground between the performance of asynchronous commit and the recoverability of synchronous commit.

6. RELATED WORK

Hathi derives inspiration from past work on storage-class memory (SCM), main-memory data stores, and persistent objects and databases.

NV-Heaps [12] and Mnemosyne [29], investigated the use of SCM and new processor primitives to provide support for persistent memory. Hathi achieves the same goal using existing hardware and commercially available flash memory technology. FlashVM and SSDAlloc [25, 9] are hybrid SSD/RAM memory managers optimized for the performance characteristics of SSDs. They do not support Hathi's transactional semantics to provide durability for arbitrary main-memory data structures. Hathi is similar to past work on durable memory transactions, such as eNVy [31], RVM [24] and Rio Vista [21]. Hathi differs in its architecture, providing word-level persistence rather than the page level of RVM, and persists data to flash rather than relying on a battery or specialized memory controller, as in Rio/Vista and eNVy (for uncommitted data).

Persistent object stores such as Texas [27], QuickStore [30], Grasshopper and Cricket [13, 26] provide a high-level structured object-interface to applications. They also provide safety properties, such as ensuring pointers in persistent data structures reference only persistent data. Hathi provides a low-level unstructured memory interface over which these systems can be layered to provide the same guarantees. In addition, Hathi operates at near-DRAM speeds using commodity hardware because of its low-level commit interface that allows fine-grained control on transaction latency. FusionIO's Auto Commit Memory [2] is similar to Hathi and provides atomic and durable data updates with memory semantics for programming. In addition, Hathi provides general-purpose transactions with support for flexible commit interfaces to persist data.

Closely related to Hathi are other main memory data stores. These fall into three categories: relational stores, *e.g.*, TimesTen and VoltDB [5, 6]; object stores, *e.g.* GemStone and RamCloud [3, 23]; and key-value stores, *e.g.* memcached [4]. The key-value stores tend to reject transactional semantics. The relational and object stores are tuned for high-throughput transactions, but focus on scaling across a cluster and provide durability through replication to other machines rather than to a local storage device, as Hathi does.

7. CONCLUSIONS

Programmers trained in the era of disks have learned that persisting data requires complex software and rich interfaces to overcome the long latency to storage. However, large memory sizes, multi-core processors, and high-speed flash storage enable a new generation of storage interfaces that reduce the gap between volatile and persistent data. Rather than maintaining two copies of data in different formats, durable memory stores enable a single data representation, optimized for in-memory access, that can also be recovered reliably when failure occurs.

In this paper, we describe Hathi, a high-speed, main-memory, durable transaction store that harnesses recent technology advances. We show the use of existing hardware and persistent memory available today, without the need to wait for next-generation non-volatile memory technologies. Hathi provides a powerful interface that eases application development, and still retains much of the performance of the underlying hardware. The flexible interface of Hathi also opens up new opportunities for application developers to explore the use of different persistent memory data structures and declarative programming styles.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) grant CNS-0834473. Swift has a significant financial interest in Microsoft.

8. REFERENCES

[1] Fusion-IO PCI-e ioDrive. `www.fusionio.com/products/iodrive`.

[2] FusionIO Auto-Commit Memory. `http://www.fusionio.com/blog/auto-commit-memory\\-cutting-latency-by-eliminating-block-i/o`.

[3] GemStone Object Server. `www.gemstone.com/products/gemstone`.

[4] memcached: High-performance Main-Memory Key-Value Store. `www.memcached.org`.

[5] Oracle TimesTen In-Memory Database. www.oracle.com/timesten.

[6] VoltDB: SQL DBMS with ACID. www.voltdb.com.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[8] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[9] A. Badam and V. S. Pai. SSDAlloc: Hybrid ssd/ram memory management made easy. In *NSDI*, 2011.

[10] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, 2011.

[11] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, 2011.

[12] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[13] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: an orthogonally persistent operating system. In *Journal of Computer Systems*, volume 7, pages 289–312, 1994.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[15] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[16] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[17] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. Group commit timers and high-volume transaction systems. In *Tandem TR 88.1*, 1988.

[18] Intel. X-25 mainstream ssd datasheet. http://www.intel.com/design/flash/nand/mainstream/index.htm.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[20] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[21] D. Lowell and P. Chen. Free transactions with rio vista. In *SOSP*, 1997.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[23] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43:92–105, January 2010.

[24] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. In *ACM Transactions on Computer Systems*, 1994.

[25] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Usenix Annual Technical Conference*, 2010.

[26] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *Workshop on Persistent Object Systems*, 1990.

[27] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: good, fast, cheap persistence for c++. In *SIGPLAN OOPS Mess*, 1993.

[28] M. Vaz Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White. An evaluation of checkpoint recovery for massively multiplayer online games. In *VLDB*, 2009.

[29] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[30] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *VLDB Journal*, pages 629–673, 1995.

[31] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS-VI*, 1994.

Ameliorating Memory Contention of OLAP operators on GPU Processors

Evangelia A. Sitaridi, Kenneth A. Ross *
Dept. of Computer Science, Columbia University
(eva,kar)@cs.columbia.edu

ABSTRACT

Implementations of database operators on GPU processors have shown dramatic performance improvement compared to multicore-CPU implementations. GPU threads can cooperate using shared memory, which is organized in interleaved banks and is fast only when threads read and modify addresses belonging to distinct memory banks. Therefore, data processing operators implemented on a GPU, in addition to contention caused by popular values, have to deal with a new performance limiting factor: thread serialization when accessing values belonging to the same bank.

Here, we define the problem of bank and value conflict optimization for data processing operators using the CUDA platform. To analyze the impact of these two factors on operator performance we use two database operations: foreign-key join and grouped aggregation. We suggest and evaluate techniques for optimizing the data arrangement offline by creating clones of values to reduce overall memory contention. Results indicate that columns used for writes, as grouping columns, need be optimized to fully exploit the maximum bandwidth of shared memory.

1. INTRODUCTION

GPU processors have been applied to various data management applications: scientific data management, OLAP and relational databases. The recent increase in GPU memories (e.g., 6GB per GPU for an Nvidia C2070 machine) makes GPU processors suitable for small to moderate size databases where the data working set is GPU resident. Such memory sizes are not so different from the CPU RAM capacities of just a few years ago that drove the development of main memory databases.

Another motivation for keeping the data resident in GPU memory comes from the increasing popularity of cloud computing. The Amazon Elastic Compute Cloud (EC2) allows one to reserve and use machines by the hour. The EC2 offers machines with 2 Nvidia M2050 GPUs for about $2.10 per hour (reservation price) or about one quarter of this rate (spot price).[1] Users do not have to provision local systems for peak loads. Instead, they adapt their computing requirements (and their costs) to their needs at different points in time. For example, at the end of a reporting period, an organization may reserve a large number of machines to do intensive data analysis. In such a context, users could reserve sufficiently many GPU machines to store their entire analysis data set in GPU RAM for a limited time, during which a large number of analytic queries would be run. The higher performance of GPU-based analytics makes this choice cost-effective, because shorter reservations would be needed than with CPU-based computations.

Due to the particular thread organization, complex memory hierarchy and high degree of parallelization of GPU processors, more effort is required to design database operators that reach maximum memory bandwidth. Efficient programming for GPU processors requires understanding of thread organization and different memory types. We focus on Nvidia's CUDA architecture, although other popular general purpose GPU architectures are similar in terms of thread-organization and memory-hierarchy. Our target platform is the Nvidia Tesla C2070, which has 14 Streaming Multiprocessors (SMs). There are 448 cores: each SM can process a warp of 32 threads simultaneously using a common instruction stream on different data.

GPUs have a radically different memory-hierarchy from a traditional CPU. Global memory is the largest type of memory but it has high latency: 400–600 cycles. Shared memory is used as a parallel, software controlled cache. Its size on the C2070 is 16KB or 48KB depending on the kernel configuration. The access time of each shared memory bank is 4-bytes per 2 cycles. To maximize performance, shared memory is organized into 32 banks, so that all threads in a warp can access different memory banks in parallel. However, if two threads in a warp access different items in the same memory bank, a *bank conflict* occurs, and accesses to this bank are serialized, potentially hurting performance.

The C2070 offers atomic operations on global and shared memory, where each thread that calls an atomic operation on a variable is promised that this variable will not be accessed by another thread until this operation is complete. Other threads trying to access the same address get serialized. This creates another possible form of contention be-

*Supported by NSF Grants IIS-1049898 and IIS-0915956, and by an equipment gift from Nvidia Corp.

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

[1]Prices taken from `http://aws.amazon.com/ec2/` in March 2012.

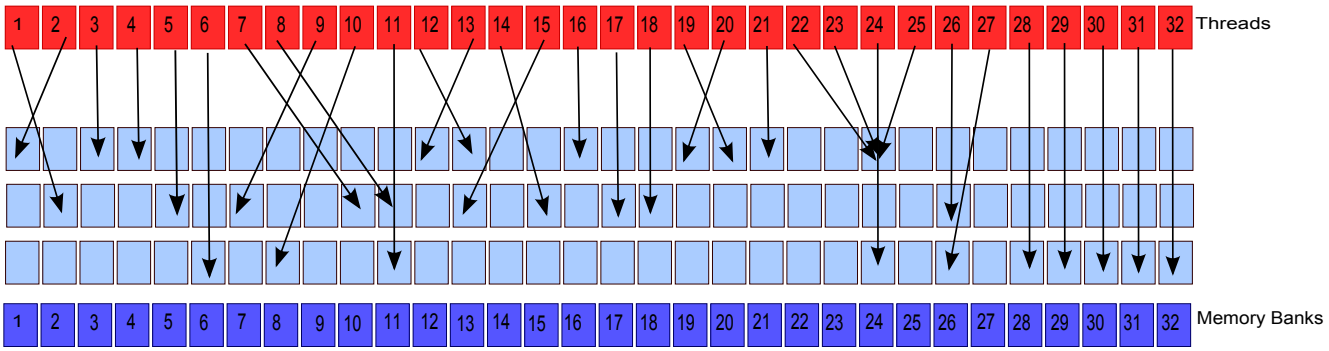


Figure 1: An example bank/value access pattern for 32 threads in a warp. Two serialization rounds are needed for reads, due to bank conflicts. Three serialization rounds resolve value conflicts for writes.

tween threads when at least one is writing data. We refer to this kind of serialization as a *value conflict*. Value conflicts can span warps, because atomic operations may still be in flight when new warps get scheduled to the SM.

Figure 1 illustrates a possible shared memory access pattern by the 32 threads in a warp. There are 2-way bank conflicts in banks 11, 13, 24, and 26. The three accesses to a single item by threads 22, 23 and 25 represent value conflicts. If these three accesses are reads, then there is no performance penalty; the system will broadcast the common data item to all three threads in one round. However, if these were write accesses, then two additional serialization rounds would be necessary for value conflicts. For additional discussion of the GPU architecture, see Appendix A.

Our goal in this paper is to examine the role of bank and value conflicts for database processing on GPUs, and to suggest bank optimizations for improving data access behavior. We focus on two core database operators: foreign-key join and grouped aggregation. For foreign key joins, shared memory is used to contain the needed fragment of the dimension table. For grouped aggregation, shared memory is used to contain the running aggregates for each group. In both cases, we make a scan through a fact table, consulting the structure in shared memory for each row.

Our key insight is that if there is additional shared memory available beyond that needed for the basic structure described above, we can use that memory to store extra copies of the underlying data. For foreign key joins, we store duplicate dimension table rows; for grouped aggregation, we store duplicate groups. In either case, we make sure that the duplicates and the original item all occupy different banks. We modify the base fact table so that some rows refer to one of the duplicates rather than the original item, in order to reduce the number of bank and value conflicts.[2] This modification is done once at data loading time, so the cost of optimization and the modification of the fact table is amortized over many queries.

Section 2 describes in more detail the problem of bank and value conflicts. Section 3 presents our solution resolving shared memory conflicts. Section 4 is an evaluation of our suggested techniques and Section 5 gives an overview of related work. Finally, in Section 6 we conclude and discuss future work.

[2]For grouped aggregation, a final pass combines the aggregates for the duplicates into a single aggregate for each item.

2. PROBLEM DESCRIPTION

We assume an in-memory OLAP setting and a star schema. Using coding techniques commonly used in OLAP databases [4, 23, 21], we assume that foreign key columns and grouping columns are coded with consecutive integer codes starting from 0. That way, dimension tables and aggregate structures can be organized as simple arrays rather than as hash tables.[3] There is a direct relationship between the array index and memory bank, since memory banks on the C2070 are distributed in a round robin fashion every four bytes. We assume all data tables are stored columnwise with 4-byte datatypes, maximizing the potential for bank parallelism.

If d is a foreign key column, then the domain of d in the initial fact table will be the integers between 0 and $c-1$ where c is the cardinality of the referenced table. If d is a grouping column, then the maximum value in the column is one less than the effective size of the aggregation table.[4] We will process a "warp's worth" of contiguous data at a time, a unit we shall call a "chunk." On the C2070, the chunk size is 32 data elements.

Now suppose that column d takes values 3 and 35 at two places in a single chunk of elements from column d. Because $3 \equiv 35 \mod 32$, both references will map to the same bank, leading to a bank conflict. For this example, we might create a new version of the element in slot 35, and put it in slot 3207, say, at the end of the table. In the fact table row with the conflict, we re-map 35 to 3207 and the conflict no longer holds because $3 \not\equiv 3207 \mod 32$. We keep track of this new row in slot 3207, which could be used for subsequent fact table rows as an alternative to slot 35 if slot 35 causes another conflict.

In the unbounded version of the problem, we do not limit the number of copies generated. If we're only concerned about bank conflicts within a warp, then 32 copies of each data item, one per bank, would guarantee that we could avoid bank conflicts altogether. In practice, fewer than 32 copies are needed. In the bounded version of the problem, we observe that the shared memory capacity puts a limit on how many values can be efficiently handled. Based on this

[3]Such optimizations are particularly important in GPUs. If different threads in a warp need to follow hash overflow chains of different length, then the execution paths will diverge and threads will be partially serialized for the length of this divergence.

[4]If some intermediate values don't appear at all in the table, then the actual grouping cardinality may be smaller.

Theta	Distinct Banks	Write SR	Read SR
0.00	20.42	3.54	3.44
0.25	20.41	3.53	3.42
0.50	20.36	3.57	3.38
0.75	19.94	3.81	3.18
1.0	18.24	5.33	2.77

Table 1: Average number of distinct banks, read serialization rounds and write serialization rounds in a chunk.

capacity, we set a budget on the average number of copies. For example, a budget of 5 would mean that the total size of the table including duplicates cannot exceed 5 times the size of the table without duplicates.

In the aggregation case, where we need to perform writes on the shared-memory-resident array, we also need to create copies to resolve value conflicts, where the same value appears more than once in a warp. We will also extend the analysis beyond the warp, looking for value conflicts between nearby warps within a fixed "window," on the grounds that an in-flight atomic update of a value might conflict with that same value in subsequent warps. In the worst case, more than 32 copies may be needed to completely avoid value conflicts.

Data partitioning between threads is static. Each thread in a thread block processes a certain number of records, so we know beforehand which records a thread is going to process. We can detect bank conflicts by scanning chunk-by-chunk and inter-warp value conflicts by remembering the set of values in the last chunks. The number of chunks we remember defines the optimization window. Our algorithm is easily extended for different regular access patterns, e.g., an access pattern where each thread reads four integers at a time using built-in CUDA vector types.

Static partitioning means that our optimization may not be effective if data items "move" from their original fact table grouping before being processed. This may limit our choices for other operators. For example, a selection operator that scanned the fact table and wrote an intermediate result containing only the matching records would change the chunking pattern. Alternative selection operators are compatible with retaining physical order. One option would be to combine the selection and aggregation into one joint kernel, so that the aggregation operator sees the data in the original locations. Another option would be to use a clustering scheme such as multidimensional clustering [19, 18] so that the records matching the selection conditions tend to be contiguous.

We consider a variety of Zipfian distributions for the fact table column, ranging from $\theta = 0$ (uniform) to $\theta = 2$ (very skewed). To give a better sense of the problem, we provide in Table 2 some statistics for a column of cardinality 1024 for different θ parameters. Without performing any optimization, we analyze the column and compute how many read and write serialization rounds are required, together with the number of distinct banks in a chunk. Note that skew hurts write serialization due to an increase in the number of value conflicts, but helps read serialization due to improved locality (since shared items can be broadcast to multiple threads).

3. ALGORITHMS

Before we discuss our main algorithms, we remark that it might be possible to reduce bank and value conflicts by reordering the fact table so that rows that would cause a conflict in the current chunk are held back until a later chunk. Reordering can only be a partial solution, because if a value occurs with a frequency higher than 1/32, then value conflicts cannot be eliminated by simple reordering. Further, there are often criteria more important than bank conflicts for ordering a fact table, so assuming the ability to reorder the table may be unreasonable.

3.1 Write Conflicts

Depending on the data distribution and data ordering each value should be assigned a different number of copies. Intuitively, frequently occurring values should get more copies, because those items are more likely to conflict, and because the extra copies are the most valuable when they can be used by many rows. Rather than statically choose a prioritization scheme for the number of copies based on frequency, we use a dynamic scheme to determine the number of copies for each value based on the "demand" for extra copies.

Initially each value is assigned one copy. We process a chunk of fact table rows at a time, until each chunk has been processed. For each chunk we proceed as follows.

We first try to assign as many values in the chunk as we can to one of its available copies, without causing any bank or value conflicts relative to previous choices. If we succeed at assigning all values, we move to the next chunk. If not, which is more likely, some values remain whose copies conflict with previous assignments. For each such value v, we assign v into one of the occupied banks and unassign the value v' that was previously there. We then try to reassign v' into one of its other copies, which could lead to a recursive sequence of reassignments. We do not consider reassignments at random. Instead, we use a breadth-first-search (BFS) algorithm to find the shortest sequence of reassignments that allows the value to be inserted without conflicts.

Assignment of a value in a chunk fails for one of the following reasons: 1) the distinct number of banks among all copies in the chunk is less than the number of banks, or 2) none of the keys can be assigned to an empty bank, because an empty bank is not reachable given the current set of copies. In both cases, to resolve the conflict a new copy of the value is created in one of the empty banks. In this way, we generate new copies of the values that are hardest to place. If we have already spent our space budget, we place the item without creating a new copy, and accept that this chunk will need multiple serialization rounds.

Two important choices affecting the space and time efficiency of the algorithm are:

- When failing, multiple bank-slots might be available. We want to assign the same number of copies to each bank-slot so that the replicated table is stored contiguously in the memory without gaps. If more copies are assigned to certain banks then there will be "holes" in the memory in the less popular banks. These holes still consume shared memory, and should be avoided.

- The order according to which we insert the copies into the BFS queue matters. If we always enqueue the lower numbered banks first, then there is a high probability that the available slots upon failure will be the higher

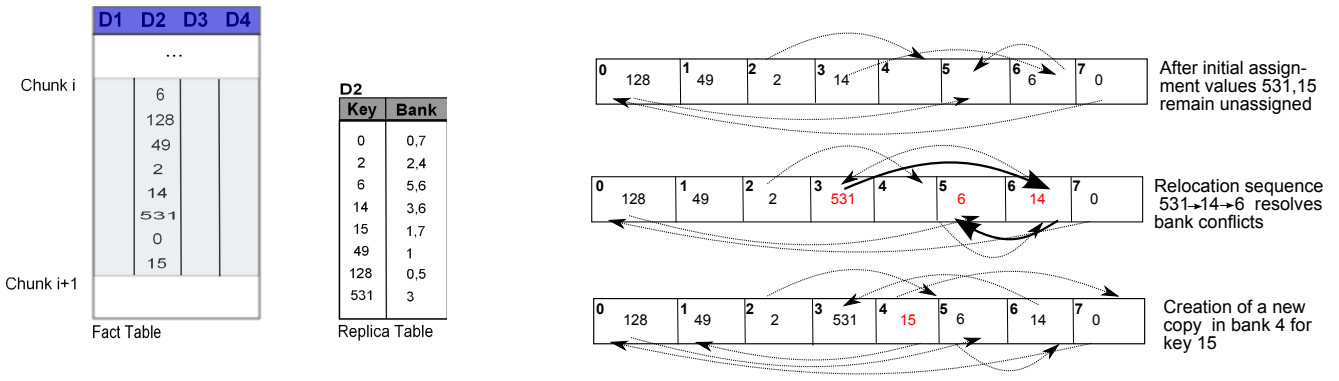


Figure 2: In this example we assume chunks of 8 elements and an equal number of shared memory banks and we show how our algorithm optimizes the second fact table column for bank conflicts. After the initial assignment, values 531 and 15 remain unplaced. A valid relocation sequence is found for value 531 (shown in bold edges). For value 15 there is no valid relocation sequence so a new copy is created in bank 4.

numbered slots, creating contention on those banks. To address this problem, we start each chunk from a different position enumeration of the copies.

Figure 2 shows how our algorithm processes a chunk of the second column of a fact table to eliminate bank conflicts. For simplicity, the chunk size in the example is 8 elements, equal to the number of threads in a warp and equal to the number of memory bank-slots. Each dashed edge links a placed value to its alternative bank locations. After the initial assignment, values 15 and 531 remain unassigned. In the next step, a relocation sequence is found for 531 that displaces 14, displacing 6 in turn. For value 15 there is no valid sequence of value movements because the only empty bank-slot 4 is unreachable, so a new copy of 15 is created in bank 4.

3.2 Read Bank Conflicts

In case of read conflicts the problem is relaxed due to the value multicasting performed in the hardware. If in a chunk there are some duplicate values, then all of those values can be assigned to the same bank without degrading the performance. This means that we can simply run the assignment algorithm for the first occurrence of the value in the chunk and put the rest of the occurrences in the same bank.

3.3 Inter-Warp Value Conflicts

Inter-warp value conflicts occur only between warps belonging to the same thread-block. We set a window size corresponding to the number of chunks prior to the current chunk to consider for value conflicts. (A window size of zero means that inter-warp value conflicts are ignored.) We shall examine the impact of window size experimentally. We extend the BFS algorithm so that copies that have previously been used within the current window are not used again in the current chunk.

In case of failure, we create a new copy as before. If we have already used the space budget we try again to place the value in the current chunk, ignoring inter-warp conflicts.

Finally, for a skewed dataset with a large window the number of copies for frequent values will also be high, increasing the search cost. To reduce this cost we consider first the copies that were least recently used, increasing the probability that a non-conflicting assignment is found early.

4. EXPERIMENTAL RESULTS

For our experiments we used synthetic data following the zipf distribution for different θ parameters. The default number of distinct values in the zipf distribution was 1024. Each column is a 4-byte integer. We used our suggested technique to resolve bank and value conflicts for different table sizes (t, up to 200M which is the maximum number fitting in GPU memory), window sizes (w) and space budgets (b).

We ran the following queries on an OLAP star-schema:

```
Q1: SELECT SUM(D1.B)          Q2: SELECT A, COUNT(*)
    FROM F, D1                    FROM F
    WHERE F.A=D1.A                GROUP BY A
```

In both cases fact table `F` was stored in the global memory. In the first query dimension tables are stored in the shared memory to perform the foreign key join, and a scalar aggregate is generated. In the second query shared memory is used to store the aggregates local to a thread block. After each thread-block computes the sums in the shared memory, it merges the results for each copy to global memory; in the end, global memory contains the correct aggregate sums. In what follows, results measuring read conflicts correspond to `Q1`, and results measuring write conflicts correspond to `Q2`.

Optimization was done on a dual-chip Intel E5620 CPU using 16 threads. GPU performance was measured on an Nvidia Tesla C2070 machine with 6GB of RAM and a nominal RAM bandwidth of 144GB/s. The GPU was configured to use 48KB of shared memory in each SM. Each thread-block processed 350K rows using 1024 threads. The number of thread blocks for a kernel was computed based on the number of table records.

Figure 3 shows the number of copies per value for different optimizations with a very lenient space budget b of just under 12 copies. Writes generate more copies than reads, because the write value conflicts create additional constraints. For similar reasons, the number of copies increases as the window size is increased. As skew increases, the number of copies decreases significantly. Many copies of a few popular items is often enough to create a conflict-free access pattern. As the number of records increases, the number of copies also increases, but the increase is fairly mild after 50M records. Figure 4 shows the average number of copies

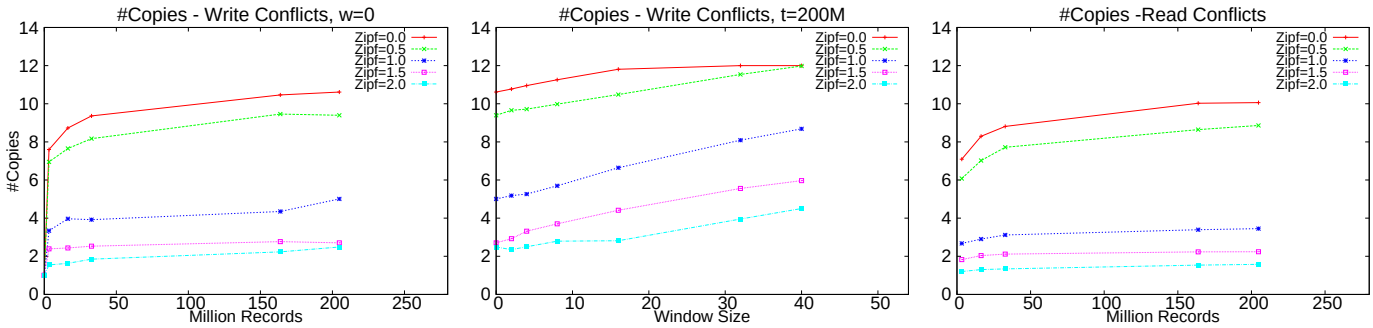


Figure 3: Number of copies per value for different table sizes, and different θ parameters

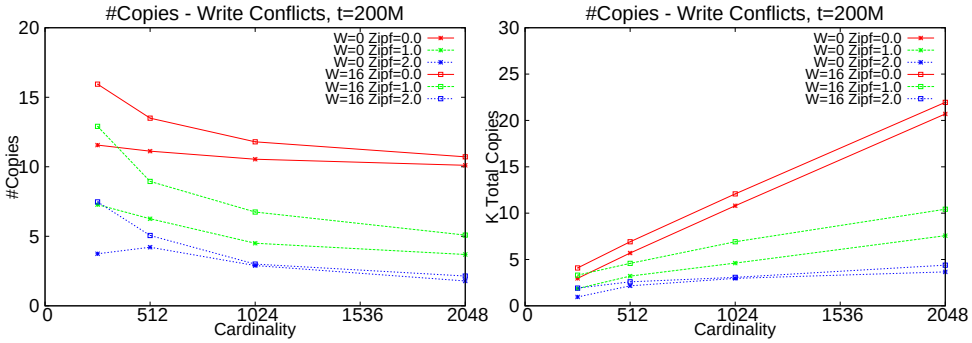


Figure 4: Number of copies per value for varying cardinality

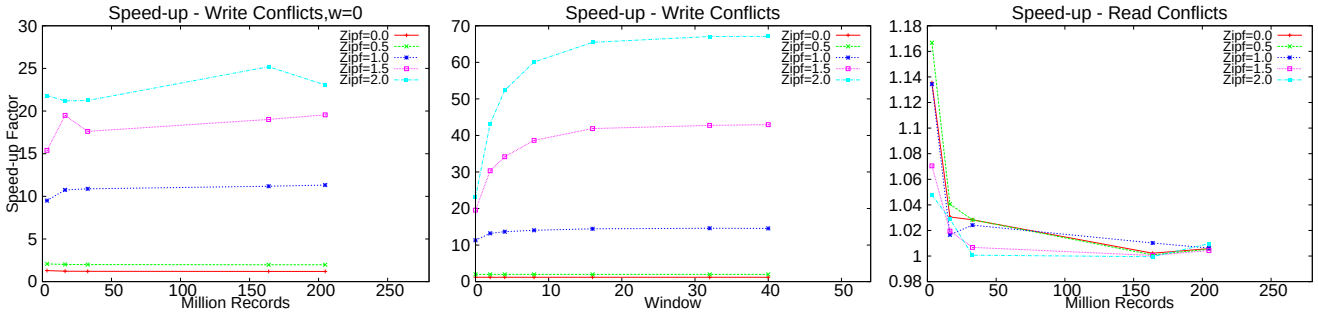


Figure 5: Speed-up for Read and Write Conflicts

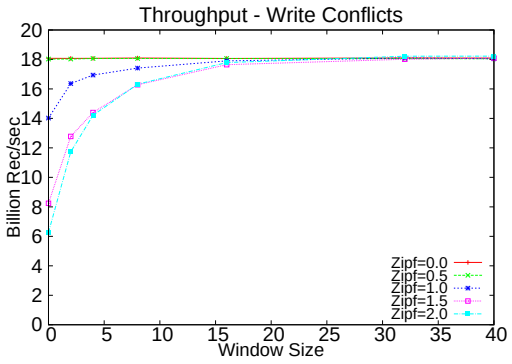


Figure 6: Throughput for different windows

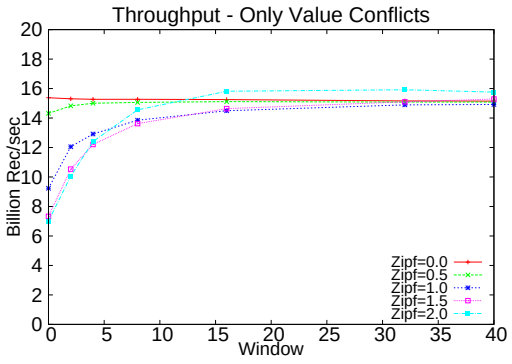


Figure 7: Optimizing for value conflicts only

and total number of replicated values for varying column cardinality. As expected, for lower cardinalities the number of copies is higher because there is an also higher probability for value conflicts and this trend is more apparent when an optimization window is set.

Figure 5 shows the speed-up of query execution on the GPU for the configurations of Figure 3. The write speed-ups are particularly dramatic at high skew, highlighting the importance of addressing conflicts when there are heavy hitters. For uniform data, the speed-up factor is about 1.2, showing that optimizing for write conflicts is still important without heavy hitters. The window size is unimportant for uniform data, but is important for skewed data. Most of the benefit of windowing occurs with a window size of 16 chunks. For reads, the speed-up is much smaller, about 2% or less once there are enough records so that thread scheduling can hide the read serialization latency. As previously noted, unlike for writes, skew helps reads because it provides more opportunities for values to be broadcast to multiple threads. In Appendix A.3 we describe a worst-case scenario for read conflicts where all threads in a warp read a different value on the same bank. We note that since read conflicts are just a sub-case of write conflicts, columns that are both read and written by various queries should be optimized for writes.

In Figure 6 we show the actual throughput for different windows, where the table size is 200M records. We can process about 18 billion records per second, i.e., about 72GB/sec. To assess the importance of optimizing writes for bank conflicts, we repeated the experiment with a modified algorithm that optimizes for value conflicts but not bank

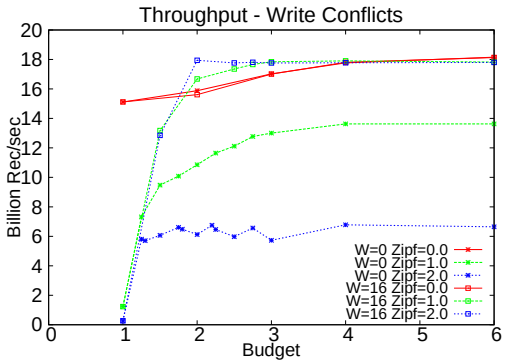


Figure 8: Throughput for different budgets

conflicts. The results in Figure 7 show that there is a 20% drop in throughput relative to Figure 6. Write bank conflicts appear to be more significant than read bank conflicts. Atomic write operations take longer, since they need a read-modify-write cycle.

For $\theta = 2.0$ in Figure 7, the performance is better than expected. At this extreme level of skew, there are two heavy hitters that occur many times in each chunk. Both of these heavy hitters have copies in every bank, so they are easy to place. The only possible bank conflicts come from the less frequent items, of which there are just a few in each chunk. The expected number of bank conflicts resembles the birthday paradox: The number of people in a group of size n having the same birthday as another member is proportional to n^2. By removing a subset of items this expectation also decreases quadratically. Thus high-skew distributions indirectly optimize for bank conflicts, even when only value conflicts are explicitly considered by the placement algorithm.

Figure 8 shows how performance depends on the space budget. For uniform data, we get close to maximum throughput at an average of 4 copies per value. For skewed data, even 2 copies per value gives good performance: our algorithm first creates copies for the frequent items that cause most of the conflicts. These results show that with a realistic (2–4X) increase in shared memory footprint, one can get most of the benefits of bank and value conflict avoidance.

In Figure 9 we see the time performance of the optimization algorithm for write conflicts. The elapsed time is just a few seconds, even for moderately large window sizes. For increasing skew, the algorithm runs faster because it is easier to arrange the records in a chunk. Our algorithm adjusts to the data distribution by creating many copies for the frequent values, so we have the freedom to place them in any bank and only have to resolve conflicts with the non-frequent values. However, for increasing window sizes, skewed data needs longer optimization time because relocations of frequent values are expensive, due to the high number of copies these items have. Reassignments of frequent items occur when infrequent values with few copies have to displace them.

5. RELATED WORK

There has been prior-work on database processing using GPU processors. Conjunctive selections and aggregations were accelerated on earlier GPU processors with a different memory architecture [9]. A subset of SQLite commands has

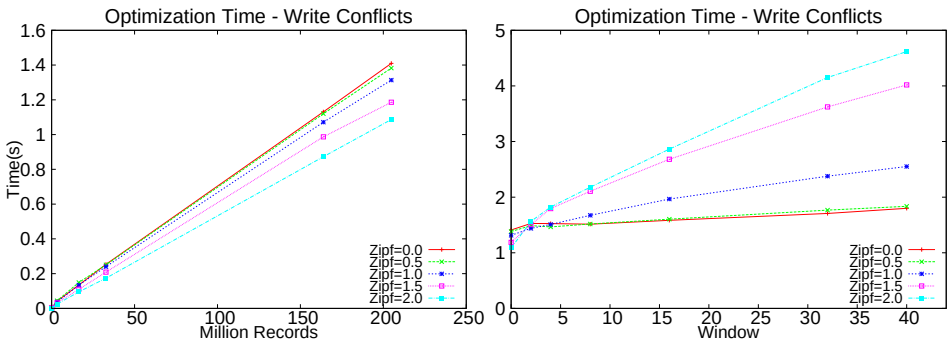


Figure 9: Optimization Time

been implemented on GPU processors resulting in significant speed-ups [1]. The power of GPU processors has also been exploited for the efficient implementation of different join algorithms [10]. Scatter and gather operations have been optimized for GPU processors to improve data locality [13]. Radix-sort was implemented with these scatter/gather operations while using shared memory to improve performance. A Map-Reduce framework has been suggested to facilitate programming of web analysis tasks on GPU processors without sacrificing performance [12].

An extended precision library has been implemented on GPUs and incorporated into a GPU-based query engine to achieve a significant performance improvement for scientific applications [16]. The bottleneck of transferring the data from CPU to GPU has been studied, optimizing the selection between GPU and CPU, suggesting the GPU as a co-processor [11, 7]. Alternative database compression algorithms have been employed to alleviate the transfer bottleneck [8]. FAST, an architecture sensitive tree index suitable for CPU and GPU processors been suggested to accelerate in-memory search [15].

Alternative aggregation strategies on chip multiprocessors have been studied to minimize thread-level contention on CPUs exhibiting different degrees of memory sharing between threads [2]. A framework for parallel data-intensive operations automatically detects and responds to contention by cloning popular items at query time [3]. This and two additional parallel aggregation strategies have been studied on a Nehalem processor [24].

Cuckoo hashing methods resolve hash collisions using multiple hash functions for each item instead of one [20, 6, 22]. During the insertion of an item, if none of the positions are vacant, the key is inserted in one of the candidate positions, selected randomly, displacing the key previously placed there. The displaced key is re-inserted in one of its alternate positions. This procedure is repeated until a vacant position is found or a maximum number of re-insertions is reached. Our method searches for the shortest relocation sequence that eliminates bank conflicts, instead of following a randomized procedure.

Data declustering techniques are used to distribute data partitions among multiple storage units, e.g., disks [14] or servers. Replication and optimal replica placement of data items has been suggested to maximize resource utilization in the Kinesis distributed storage system [17].

6. CONCLUSIONS

We defined the problem of bank conflicts and value conflicts for data-processing operators on GPU processors. We studied the impact on performance of those two contention factors for two popular OLAP operators on CUDA architecture. We suggested and evaluated a technique for resolving conflicts that can easily be configured for different memory access patterns and space budget requirements. Results indicate that columns that are written by various queries e.g., potential grouping columns, should be optimized for writes and that read conflicts should not be a high priority for bank optimization. We plan to apply the same technique on clustered tables using separate structures for fragments of the fact table to take advantage of local skew. We also plan to extend our technique for different key sizes (e.g., 2-byte and 8-byte), given that GPU processors favor 4-byte accesses.

7. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU*, 2010.

[2] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[3] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, 2010.

[4] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Conference*, 1985.

[5] N. Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, November 2011.

[6] U. Erlingsson et al. A cool and practical alternative to traditonal hash tables. In *Workshop on Distributed Data and Structures*, 2006.

[7] R. Fang et al. GPUQP: query co-processing using graphics processors. In *SIGMOD*, 2007.

[8] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3, 2010.

[9] N. K. Govindaraju et al. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[10] B. He et al. Relational joins on graphics processors. In *SIGMOD*, 2008.

[11] B. He et al. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34, 2009.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT*, 2008.

[13] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Supercomputing*, 2007.
[14] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *SIGPLAN Not.*, 1992.
[15] C. Kim et al. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.
[16] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *DaMoN*, 2010.
[17] J. MacCormick et al. Kinesis: A new approach to replica placement in distributed storage systems. *Trans. Storage*, 2009.
[18] V. Markl, F. Ramsak, and R. Bayer. Improving olap performance by multidimensional hierarchical clustering. In *IDEAS*, 1999.
[19] S. Padmanabhan et al. Multi-dimensional clustering: A new data layout scheme in DB2. In *SIGMOD*, 2003.
[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51, 2004.
[21] P. Pucheral, J.-M. Thevenin, and P. Valduriez. Efficient main memory data management using the DBGraph storage model. In *VLDB*, 1990.
[22] K. A. Ross. Efficient hash probes on modern processors. In *In Proceedings of the 23nd International Conference on Data Engineering*, 2007.
[23] K.-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus system. *ACM TODS*, 15(1), 1990.
[24] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

APPENDIX

A. CUDA ARCHITECTURE

A.1 Thread Hierarchy

All threads belonging to a thread block are executed in the same SM, and each block is executed independently from other blocks. A block has a maximum number of threads, which in current high-end processors can be up to 1536 threads. A thread-block is grouped into a set of warps, e.g., a block of 1536 threads has forty-eight 32-thread warps.

CUDA employs a Single Instruction Multiple Thread (SIMT) architecture where threads in a warp start at the same program address, but keep private register state to execute on independent data. Performance is maximized when all threads in a warp follow the same execution path. When they diverge, such as during a conditional branch, their execution is serialized.

A.2 Memory Hierarchy

To fully utilize the memory bandwidth of a GPU programmers should carefully design their memory access patterns. CUDA memory spaces are either located off-chip or on-chip. Off-chip memories are more plentiful but have higher access latency. *Global* and *local* memory are located off-chip, while *shared memory* and *registers* are located on-chip.

Off-chip Memories

All threads access the same global memory space. Global memory accesses should be *coalesced* within a warp to minimize memory transactions. The simplest example of an optimal coalesced memory access is when all threads in a warp access consecutive 4-byte addresses. Local memory is local to a thread and has similar latency to global memory.

On-chip Memories

Shared memory is available to all threads in a thread block for shared access. On the Nvidia C2070, its size is 16KB or 48KB per SM, depending on the kernel configuration. Shared memory uses the same circuits as the L1 cache, and a programmer can configure how much memory is allocated for shared memory space and how much for the L1 cache, depending on the memory access pattern of the kernel. Shared memory can be both read and written by the threads in a block.

Shared memory is divided into banks and their number is equal to the number of threads in a warp. Banks are interleaved so that consecutive 4-byte words belong to different banks. If all the memory requests of the threads in a warp fall into different banks, they can be serviced in parallel. However, if two or more threads access the same bank for different items, the access to this bank are serialized. If there are multiple conflicts in a memory request, the hardware splits this request into as many conflict-free requests as are needed. A multicast mechanism is implemented for read requests where a 4-byte word being read by multiple threads in a warp can be broadcast to all requesting threads simultaneously. A read bank conflict thus occurs only if multiple threads accessing the same bank request a *different* 4-byte word. If the keys are 8-bytes the memory request of a warp is split in two requests, one request per half-warp [5]. As a consequence, for a read-request there is a bank conflict if two or more threads in either of the half-warps access different addresses of the same bank.

A register's scope is a single thread. The access latency of a register is 0 cycles, assuming there are no read-after-write dependencies. Register memory only stores static arrays, and high register-usage limits the parallelism by limiting the number of threads in a thread block, due to the limited size of the register file.

Atomics

CUDA offers atomic arithmetic and bitwise functions on global and shared memory. Shared memory atomics are significantly faster than global atomics. Atomic functions perform read-modify-write operations on 32 or 64-bit words: when a thread performs an atomic function it is guaranteed that no other thread will interfere until this operation is finished. When multiple threads write atomically to the same address the accesses to this address are serialized.

A.3 Impact of bank and value conflicts

We used the CUDA command line profiler to count the number of bank conflicts for different degrees of conflicts. The CUDA profiler reports counters per SM. We profiled the performance of the same operators as in Section 4. We generated synthetic data causing a specified number of serialization rounds per warp. We were careful to make sure that all values are distinct within a chunk, to show the worst case scenario for read conflicts. Figure 10 shows the throughput and the number of conflicts as reported by the `l1_shared_bank_conflicts` counter of the CUDA Profiler. For the worst case of read conflicts the throughput is less

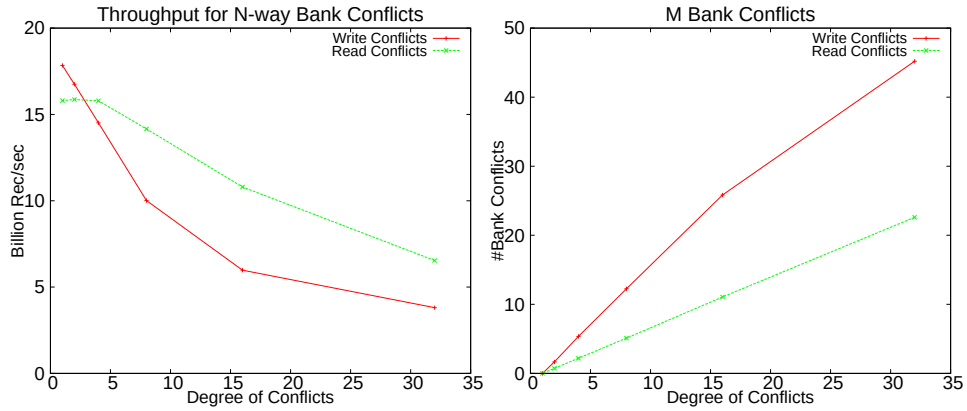


Figure 10: Throughput and Profiler Counters for varying conflict degree, t=30M

than half of the conflict-free performance. As previously noted, the effect for write conflicts was more significant.

One may wonder whether an access pattern that reads many different elements concentrated in a few banks is realistic. After all, a simple randomization of the bank location would lead to a reasonable spread of items across banks. The following example suggests that degenerate cases may indeed arise in practice.

Consider a foreign key join where the foreign key consists of two attributes, x, and y. Imagine that the dimension table represents metadata about cells in an n by m grid, so there are nm rows in total. Suppose that the dimension table is clustered by (x, y). The fact table also contains attributes x and y, but the fact table is clustered by y. As we scan through the fact table we will be repeatedly (for each y value) touching n dimension table rows separated by m rows. If d is the greatest common divisor of m and 32, then this access pattern will use only $32/d$ banks. In the worst case, m is a multiple of 32, and only one bank is accessed.

X-Device Query Processing by Bitwise Distribution

Holger Pirk	Thibault Sellam	Stefan Manegold	Martin Kersten
CWI, Amsterdam	CWI, Amsterdam	CWI, Amsterdam	CWI, Amsterdam
The Netherlands	The Netherlands	The Netherlands	The Netherlands
holger@cwi.nl	sellam@cwi.nl	manegold@cwi.nl	mk@cwi.nl

ABSTRACT

The diversity of hardware components within a single system calls for strategies for efficient cross-device data processing. For example, existing approaches to CPU/GPU co-processing distribute individual relational operators to the "most appropriate" device. While pleasantly simple, this strategy has a number of problems: it may leave the "inappropriate" devices idle while overloading the "appropriate" device and putting a high pressure on the PCI bus. To address these issues we distribute data among the devices by partially decomposing relations at the granularity of individual bits. Each of the resulting bit-partitions is stored and processed on one of the available devices. Using this strategy, we implemented a processor for spatial range queries that makes efficient use of all available devices. The performance gains achieved indicate that bitwise distribution makes a good cross-device processing strategy.

1. INTRODUCTION

Computer systems have ceased to be centralized systems in which a CPU controls dumb storage devices. Special purpose extension cards that can support the CPU, in particular General Purpose Graphics Processing Units (GPGPUs), are available at low prices. The design and use of these cards, however, is fundamentally different from the one of the CPU. Fast sequential execution based on behavior prediction (pipelining, prefetching, branch prediction, ...) is replaced by simple, yet massively parallel, execution. The internal memory of these devices is usually orders of magnitude faster but offers much smaller storage capacity than traditional memory and lacks the benefits of virtual addressing. While this can make their use difficult, it can also provide opportunities for significant gains in application performance. The high bandwidth and compute power of have aroused interest of data management researchers [2, 6, 14, 5, 15, 13, 6]. While computation-intensive applications are a natural fit for the massively parallelized architecture of GPGPUs, the, rarely computation intensive, processing of relational queries can not benefit from the available compute resources to the extend possible. However, data intensive applications like relational query processing can still benefit from the fast memory of GPUs if implemented appropriately.

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

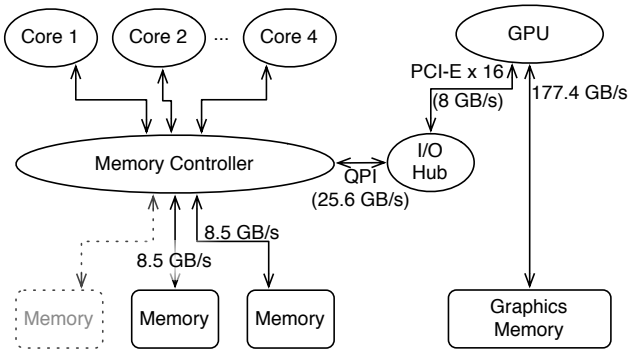
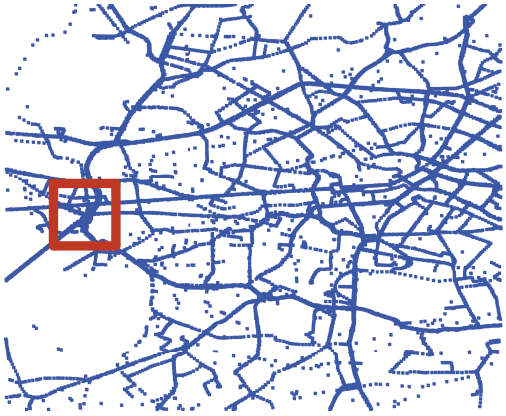


Figure 1: The Architecture of a typical GPU/CPU System

Unfortunately, the relatively small memory capacity and the lack of virtual memory management complicate the efficient management of large database on GPGPUs. The current state-of-the-art is to ship (parts of the) data to the GPU, process individual relational operations and transfer the results back to the main CPU [14]. As illustrated in Figure 1, the PCI-Bus has more than an order of magnitude lower throughput than the internal device memory. Continuous transfer of data through this bus can become a major bottleneck for CPU/GPU co-processing [11]. While the expensive cross-device transfer can be avoided in some cases, the costs for the necessary streaming of large datasets can negate the cost improvement achieved by the GPU processing. To mitigate this problem to some extend, data compression can help to reduce the data volume in the limited GPU memory [7]. However, lossless data compression may fail to achieve the necessary savings or hurt the performance due to the decompression costs. In particular the massively parallelized architecture of GPUs limits the arsenal of efficient compression methods. An alternative to reduce the data volume is the use of sampling techniques [19]. While samples are a good means to reduce the data size, they are merely useful to gain a rough overview of the available dataset. To gain precise results, sampling is not a suitable technique.

To efficiently exploit the superior bandwidth of the GPU's internal memory we propose to re-evaluate the techniques that helped reducing the footprint of main memory resident databases: the *Decomposed Storage Model (DSM)* [4] reduced the footprint to the columns that are actively used for query evaluation, i.e., the *hot* columns. On top of that, lightweight compression of the values within a column proved effective [20]. However, to achieve the necessary compression, decomposing tuples into columns of scalar values may not be enough. To limit the data volume to the capac-



(a) Traffic in Western Berlin

(b) Drill-Down to Dreieck Funkturm (A Traffic Hot Zone)

Figure 2: Spatial Drill-Down

ity of the respective device memory we propose to take data decomposition to the next level: *Bitwise Decomposition*. Each of the resulting bit-partitions is stored (non-redundantly) and processed on the most appropriate device. The data volume on each device can be controlled by varying the number of bits that are stored in the device memory. Combined with lightweight compression this promises GPU-supported data processing without the need for expensive cross device data transfers.

To this end, we make the following contributions:

- We introduce the idea of Bitwise Distribution of relational data across processing devices.
- We develop a model to determine the optimal distribution strategy.
- We evaluate Bitwise Distribution in combination with Frame Of Reference/Delta compression for range queries on real-life spatial trajectory data.

The remainder of this paper is organized as follows. In Section 2 we present our use case, and provide an overview of the benefits and challenges that come with GPGPU programming. In Section 3 we describe the data distribution strategy, its implementation and the calculation of the necessary tuning parameters. The query processing on top of the distributed data is described in Section 4. In Section 5, we evaluate our approach and conclude in Section 6.

2. BACKGROUND

To introduce the problem, we provide a brief overview of established spatial data indexing techniques, and how our work differs from these. Following that, the boundary conditions of GPGPU processing as well as the resulting opportunities and challenges will be discussed.

2.1 Spatial Data Management

The use case that motivates this work is the need for quick retrieval of historical GPS data in a traffic analysis context. A database stores several years of vehicle movements on the European road network to support applications such as traffic forecast or infrastructure monitoring. Figure 2 illustrates the monitoring of traffic in the city of Berlin. Reporting is generally focused on certain hot zones of traffic. This results in spatial window (i.e., 2 dimensional range) selection queries. The results of these queries can either be displayed directly or post-processed by more sophisticated applications. These include data mining or decision support applications. Since the data as well as the queries are two-dimensional, classical index-structures do not support the application well.

There has been substantial work on spatial data management, especially on spatial indexing methods. The general idea is to cluster the data according to their geographical proximity in order to improve I/O performance. Many data structures have been proposed. Among those, the R-Tree [12] is ubiquitous. It encloses spatial objects in bounding boxes. More precisely, a node represents the smallest possible bounding box that can cover its child nodes. The major drawback of this data structure is that the entries might overlap and contain empty space, which degrades worse case lookups performance. Many improvements were proposed. For instance, R^+ trees [12] follow similar principles, but do not allow bounding boxes to overlap. This allows faster traversals, at the cost of higher construction and maintenance costs. Another family of methods relies on spatial grids to index the data. For instance, a uniform grid is proposed in [8]. However, such a grid is not well-suited to non uniform distributions. The quad-tree is a popular alternative. Each node represent one of four quadrants of its parent node [18]. This allows adaptive data storage. Finally, the data may be adapted to traditional single dimension data structures by applying dimensionality reduction techniques such as space filling curves. For instance, Z-ordering [9] relies on bit interleaving, which preserves some spatial locality.

Our aim is to support a large volume of simultaneous queries: we target bandwidth rather than latency. Therefore, our approach to retrieving spatial data is orthogonal. It relates mainly to work carried out on bulk data processing, illustrated by systems such as MonetDB [3]. There are two main differences with the previously presented data structures. First, we assume that current hardware provides sufficient throughput to scan a complete dataset efficiently, while the branches induced by traditional index structures are ill-suited for massively parallel processing. Second, we rely on a bit-level decomposition of the data. There is to our knowledge little to no work using it for multidimensional data storage and retrieval.

2.2 GPGPU Programming

The programming of a GPU is very different from the programming of a CPU. This is largely due to the fundamentally different architecture. To achieve high compute power at low costs, GPUs rely on a parallelism paradigm called *Single Instruction Multiple Threads* (SIMT).

Single Instruction Multiple Threads

The notion of SIMT is a peculiarity of GPU hardware and the source of a common misconception of GPU programming. Even though a GPU supports many parallel threads, these are not independent as they are on a CPU. All cores of a processor execute the same instruction at any given cycle. An ATI Evergreen-class GPU, e.g, has 16 SIMT-cores which execute every instruction for at least 4 cycles. Thus, every scheduled instruction is executed at least 64 times. Each core does, however, have its own registers and usually operates on different data items than the other cores. A set of threads coupled like that is called a *Work Group*. A work group of less than 64 items underutilizes the GPU's computation units. This also has a severe impact on branching: if one branch in a Work Group diverges from the others the branches are serialized and executed on all cores. The cores that execute a branch involuntarily simply do not write the results of their operations.

The Programming Model

Programming the high number of SIMT-cores of a GPU in an imperative language with explicit multithreading is a challenging task. To simplify GPU programming, a number of competing technologies based on the kernel programming model have been introduced. The most prominent ones are: DirectCompute, CUDA [17] and OpenCL [16]. While the earlier two are proprietary technologies, the later is an open standard that is supported by many hardware vendors on all major software platforms. The supported hardware does not just include GPUs, but CPUs as well: *Intel* and *AMD* provide implementations for their CPUs, *AMD* and *NVidia* for GPUs. *Apple*, one of the driving forces behind OpenCL, ships their current OS version with an OpenCL implementation for both GPUs and CPUs. The portability does, however, come at a price: to support a variety of devices, OpenCL resorts to the least common denominator, which radically limits the programming model.

The most important concept in OpenCL is the Kernel: a function that is defined by its (OpenCL) C-code, compiled at runtime on the Host, transferred in binary representation and executed on the device. The Kernel is then scheduled with a specified problem size (essentially the number of times the kernel is run) to operate on a number of data buffers.

A Priory Fixed Problem Size

This is done by dispatching the kernel (the compiled processing function) for execution on the device with the problem size as a parameter (e.g., n). The kernel is then executed exactly n times. Each invocation has access to an id that can be used to determine which piece of the work to do. If the problem size is not known apriori, a workaround is to use a single complex thread to do all the work. Naturally, reducing the degree of parallelism on a GPU has a negative impact on performance. Another solution is to specify an upper bound on the problem size and abandon the execution of some of the kernels at runtime. Since no new kernel can be scheduled to a core until the work group has finished, this may also lead to underuse of the available compute power.

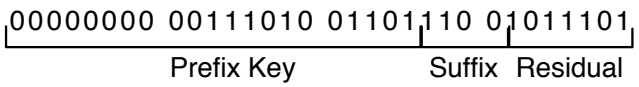


Figure 3: Bitwise Decomposition of Spatial Values

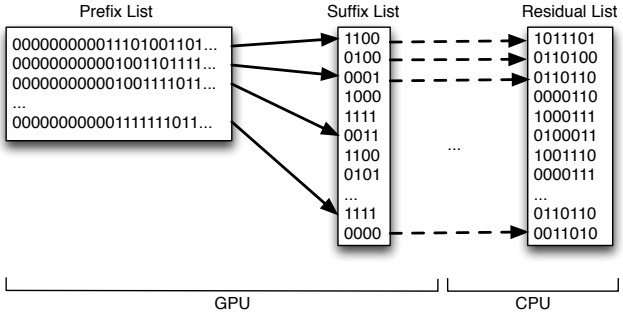


Figure 4: Bitwise Distribution of Spatial Values

Static Memory Allocation

The OpenCL programming model does not allow dynamic (re)allocation of memory. Similar to the problem size, input and output memory sizes have to be specified up front. While this is a problem for operations with a large upper bound on the output size (e.g., joins), for selection queries an overallocation of output and an overflow check at runtime mitigates this problem. Even though the lack of memory reallocation is a problem it also has a significant performance advantage. Memory can be addressed using physical addresses, which eliminates the need for costly translation from virtual to physical addresses and speeds up memory access.

3. DATA LOADING

In this section we introduce how the GIS data is prepared for processing. We introduce two splits at the bit level in order to spread data between the devices. However, determining where to split is non trivial. A model of GPGPU memory consumption is presented, as well as our clustering algorithm.

3.1 Bitwise distribution

Thanks to their high bandwidth, GPUs can evaluate queries very efficiently. However, their limited memory capacity complicates their effective use on large datasets. To resolve this problem, the GPU can be used to generate an (over)approximate answer to the query. The CPU can be used, subsequently, for result set refinement and tuple reconstruction. Figure 3 illustrates a data partitioning scheme to support such GPU/CPU co-processing. The GPU holds the most selective components of the data while the main memory holds the bits necessary to restore the original values (*residuals*). In our use case, we choose the most significant bits of the coordinates in each dimension. The data in the GPU memory is essentially a reduced-resolution representation of the original data.

To minimize the number of false positives, the query dependent selectivity of the GPU resident data has to be maximized under the given GPU memory limitations. This is achieved through a simple form of prefix compression: the GPU-resident bits are split into a prefix and a suffix. The suffixes of all values with a common prefix are physically clustered and stored in a suffix list. The prefixes, together with the respective offset into the suffix list are stored in a prefix list (Figure 4). We expect that in most spatial datasets,

Denotation	Description
p	Prefix size (bits)
s	Suffix size (bits)
$i \in \{X,Y\}$	Dimension
D_i	Size of coordinate i (bits)
D_m	Smallest value of D_i (bits)
O	Size of a pointer to a cluster (bits)
H_i, L_i	Highest, lowest value on i
MAX	Shared memory of the GPU
N	Number of items
$P_i(p)$	Clusters in dimension i
$S(p,s)$	Space required for (p,s)

Figure 5: Parameters and variables of the space usage model

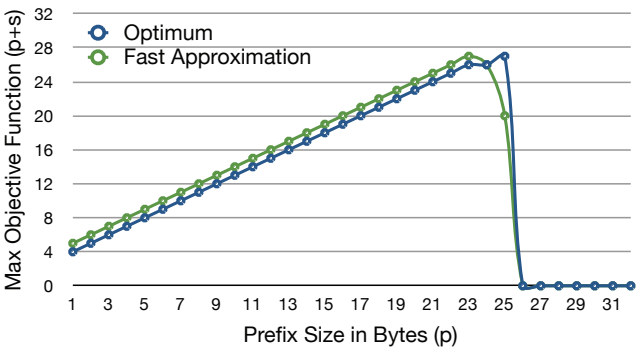


Figure 6: Optimal splits for varying k

the prefix bits of coordinates offer less variations than the suffixes. This allows efficient physical clustering and subsequent compression. For instance, the integer representation of a standard positive latitude with five decimals (up to $90x10^5$, the comma is implicit) always contains a null byte prefix.

3.2 Bitwise Decomposition

Finding a good split between *prefixes*, *suffixes* and *residuals* given a GPU memory capacity is a non-trivial problem. This section presents a model of the memory consumption for a given decomposition. We apply bounded search over the candidate solutions yielded my the model in order to find an optimal decomposition.

We apply our model to the two dimensions (longitude, latitude) of our use case, represented by $i \in \{X,Y\}$. Generalizing it to higher dimensionality is straight forward. p is the size in bits of the prefixes, s the size of the suffixes, D_i the number of bits necessary to represent one coordinate (D_i = 32 bits for an integer). For each dimension, H_i and L_i represent respectively the highest and the lowest values in the dataset. N is the total number of items. The memory capacity of the GPU is MAX. These parameters are summarized in Figure 5.

Figure 6 shows the maximal value of the objective function that can be achieved varying p with our use-case dataset and hardware. Increasing the prefix size allows better compression. Then, more bits can be stored on GPU, and less on the CPU. However, any value of p greater than 25 yields clusters that are too large to fit in GPU memory. This point is precisely what is targeted.

The range of values that can be covered by a prefix p in one dimension is 2^{D_i-p}. Therefore, the total number of prefixes necessary to represent all the values in a dimension is:

$$P_i(p) = \lfloor \frac{H_i}{2^{D_i-p}} \rfloor - \lfloor \frac{L_i}{2^{D_i-p}} \rfloor + 1$$

As a result, the total number of clusters in two dimensions given the size of the prefixes is $P_X(p)P_Y(p)$. For each cluster, the GPU stores the prefix p and a pointer to the suffix list. The size (in bits) of the pointer is O. Inside the clusters, each of the N points is represented with s bits for each dimension. Therefore, with two dimensions, the total storage footprint associated with a partition scheme (p,s) can be expressed as:

$$S(p,s) = 2(p+O)K_X(p)K_Y(p) + 2Ns$$

The objective is to store as much information about the data as possible on the GPU. Under these assumptions, the partitioning strategy research may be expressed as a constrained optimization problem:

$$\begin{aligned}
\underset{p,s}{\text{Max}} \quad & p+s && (o)\\
s.t. \quad & S(p,s) \leq MAX && (c1)\\
& 0 \leq p+s \leq L_X && (c2)\\
& 0 \leq p+s \leq L_Y && (c3)\\
& p,s \in \mathbb{N} && (c4)
\end{aligned}$$

A bounded search in the solution space is a simple and efficient way to reach an optimal solution. The algorithm in Figure 7 illustrates the procedure. The worse case complexity of the algorithm (i.e., size of the search space) is $O(\frac{D_m(D_m-1)}{2})$ with $D_m = \min_{i\in\{X,Y\}} D_i$. However, it seems reasonable to assume low minimum values (L_i) for most data sets. Also, during the exploration, if a candidate solution (p,s) violates $(c1)$ then any subsequent improvement of p or s can be discarded. This allows pruning of the search space.

An acceptable approximation of the optimum may quickly be obtained by assuming that constraint $(c1)$ is always met. This is applicable to any dataset wider than the GPU capacity. In this case, the optimization problem becomes:

$$\begin{aligned}
\underset{p,s}{\text{Max}} \quad & p+s && (o')\\
s.t. \quad & (p+O)K_X(p)K_Y(p) + Ns = MAX/2 && (c1')\\
& 0 \leq p+s \leq \min_{i\in\{X,Y\}} D_i && (c2')\\
& p \in \mathbb{N}, s \in \mathbb{R} && (c3')
\end{aligned}$$

Equation $(c1')$ yields an approximation of the optimal suffix size given a prefix:

$$\tilde{s}(p) = \lfloor \frac{MAX}{2N} - \frac{p+O}{N} K_X(k)K_Y(k) \rfloor$$

Thereby, the whole optimization system may be approximated as follows:

$$\begin{aligned}
\underset{p\in\mathbb{N}}{Max} \quad & p+\tilde{s}(p)\\
s.t. \quad & p+\tilde{s}(p) \leq \min_{i\in\{X,Y\}} D_i
\end{aligned}$$

Such result can be obtained in $O(\min_{i\in\{X,Y\}} D_i)$ by enumerating the possible values of p.

```
p_max ← 0
s_max ← 0
for p = 1 → D_m do
    for s = 1 → D_m − k do
        size ← S(p, s)
        if size > MAX then
            break
        end if
        if p + s > p_max + s_max then
            p_max ← p
            s_max ← s
        end if
    end for
end for
```

Figure 7: Optimal split lookup algorithm

3.3 Implementation

Once the optimal parameters are determined, the data can be decomposed accordingly. This is done in two phases. In the first phase, the data is scanned to build the prefix list (see Figure 3) according the number of prefix bits k. This step is similar to the determination of the size of the clusters of a radix clustering. The offsets into the suffix list are generated by prefix-summing the cluster sizes in the prefix list.

In the second phase, the data is scanned again to fill the suffix and residual list. This is equivalent to the clustering step of a radix clustering. After this phase, the data is in bitwise decomposed representation. The cluster and delta lists are transferred to the GPU memory and `free`d in the system's main memory, while the residual list is kept in the main memory. Therefore the data is distributed over the available devices.

4. QUERY PROCESSING

The decomposed distribution of the stored tuples largely determines the query processing strategy. The query is evaluated in phases with every device being responsible for one phase of the query evaluation. In each phase, the device does the best with the data it has available: narrowing down to the final result as much as possible and (partially) reconstructing the tuple values. In a (quite common) CPU/GPU co-processing setup, the processing is done in two phases: *GPU Preselection* and *CPU Refinement*. Since the result of each phase is a (potentially inaccurate) representation of tuples in the database, the two steps can be implemented like operators in a relational DBMS. Due to the high overhead when transferring data across devices, the Volcano-model [10] is not well suited to connect these operators. We, thus, implement them in the bulk processing model: in each phase the intermediates are materialized into the device's memory and copied once the processing phase has completed. When handling continuous query streams, the two evaluation phases of different queries can be interleaved, keeping all devices busy.

4.1 Phase 1: GPU Preselection

In the first phase, the GPU performs what can be considered a prefiltering of the dataset. Since the GPU memory only contains an approximate representation of the data (it misses the residuals), it cannot give an exact answer to the query. Instead it does a best-effort filtering of the tuples and returns partial results (see Figure 8). The partial result set is a superset of the exact answer to the query, but contains all the information for the CPU to narrow it down to the exact result set.

```
typedef struct {
        short queryID;
        int partialX, partialY;
        int tupleID;
} PartialResult;
```

Figure 8: Partially Reconstructed Tuple

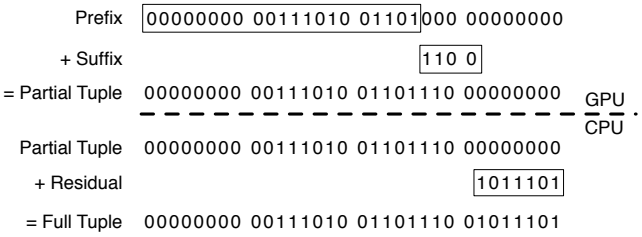


Figure 9: Cross-Device Tuple Reconstruction

Parallelization

The high degree of processing parallelism in GPUs calls for an equally high degree of parallelism in the query processor implementation. To achieve this, we parallelize the query evaluation in two dimensions: the queries and the data clusters. Every combination of query and cluster is assigned to one thread. While the number of queries is moderate (hundreds), the number of clusters (as determined by k) is usually high (in the range of tens of thousands). This results in a high degree of parallelism that can be used for efficient GPU data processing. However, the work items are grouped into work groups without optimization. Due to the SIMT processing model, the processing time spent on a cluster is determined by the size of the largest cluster in the work group. To somewhat mitigate this problem for highly skewed data, we split unusually large clusters into smaller chunks. Clustering work items into groups with similar cluster size is an optimization that we consider future work.

Runtime Pruning

When evaluating a relatively low number of queries (thousands) on the high number of clusters, it frequently happens that a cluster is hit by no query. Since the overlap of the cluster with the query can be checked by looking at it's prefix, skipping cluster scans is an easy optimization. This is a case of the workaround we discussed in Section 2.2: bounding the problem size and abandoning kernel execution at runtime.

4.2 Phase 2: CPU Refinement

In the second phase, the CPU copies the partial results from the GPU's device memory and joins them with the main memory resident residual list. Since this is an invisible/positional join [1] on the tupleID, it is cheap. The partial results are combined with the residuals to produce the final tuple values (see Figure 9). The query conditionals are evaluated again on the reconstructed values and the results, in case of a hit, copied to the output buffer.

5. EVALUATION

To evaluate the performance impact of our approach, we compare it to existing CPU-only and GPU-only approaches. As benchmark we use a set of range queries on a spatial trajectory database.

5.1 Setup

The experiments where run on a machine with two Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The used GPU is a GeForce GTX 480 with 1.5 GB device memory.

The data is a real-life dataset consisting of around 240 Million 2D spatial datapoints. The data was collected by an industry partner by tracking Navigation Devices in North-Western Europe. The queries are generated by randomly selecting a point from the dataset and constructing a rectangle around it. The size of the rectangle is random but within a maximum. The generation of the queries is according to a workload description that we received from mentioned industry partner.

5.2 Loading

The data is stored on disk in binary format. Before the query evaluation is started it is loaded into main memory and decomposed/-compressed as necessary for the presented query evaluation technique. As discussed in Section 3.3, decomposition and compression involves radix clustering the data. This is likely to increase the loading costs. To give an impression of the decomposition costs, Figure 10 shows the base costs for loading and the added costs for the clustering step. As expected, the clustering doubles the costs since it involves a second pass through the data.

5.3 Query Processing

To evaluate the query processing techniques, we varied three different parameters: 1. The number of queries evaluated, 2. the processing device (GPU vs. CPU), 3. the data representation (bitwise decomposed (BWD) vs. attribute-wise decomposed).

Since we parallelize query processing with the number of queries on the GPU, we effectively turned the evaluation of many queries into a single, parallelized nested loop (theta) join. For reference, we also report the results of a similar evaluation technique on the CPU. All processing models that evaluate multiple queries in a single run through the queries are marked with QJ=QueryJoin.

Figure 11a shows the results of our main experiment. CPU is the processing of the queries in a query-at-a-time manner on the plain data. This is the baseline for our evaluation. The state of the art for GPU processing relies on streaming the plain data to the GPU and evaluating the queries in parallel. While the performance compared to CPU-based processing is worse for a single query, the GPU benefits from larger query sets.

The QueryJoin optimization on the CPU shows worse performance than the baseline: The more complex loops seem to hurt CPU efficiency. The same holds when combining bitwise decomposition with the QueryJoin optimization. The evaluation of 2048 queries was not complete within 30 minutes at which point we aborted the query evaluation.

Bitwise decomposed storage and processing on the CPU is the best evaluated solution when evaluating a single query. For larger query sets, GPU/CPU co-processing outperforms all other approaches significantly. For 2048 parallel queries, GPU/CPU co-processing is

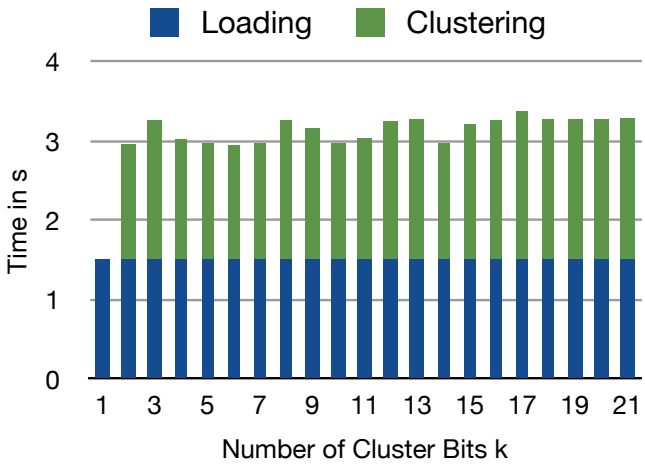


Figure 10: Data Loading+Clustering Time

about 6 times faster than bitwise decomposition on the CPU and more than two orders of magnitude faster than GPU processing on plain data. CPU processing on plain data is outperformed by more than three orders of magnitude.

In addition to the query evaluation performance, bitwise decomposition promises good load balancing over the available devices. To illustrate this, Figure 11b shows the time that is spent processing data on each device. It shows that single queries induce most of their load on the GPU, the load is almost perfectly distributed for larger query sets.

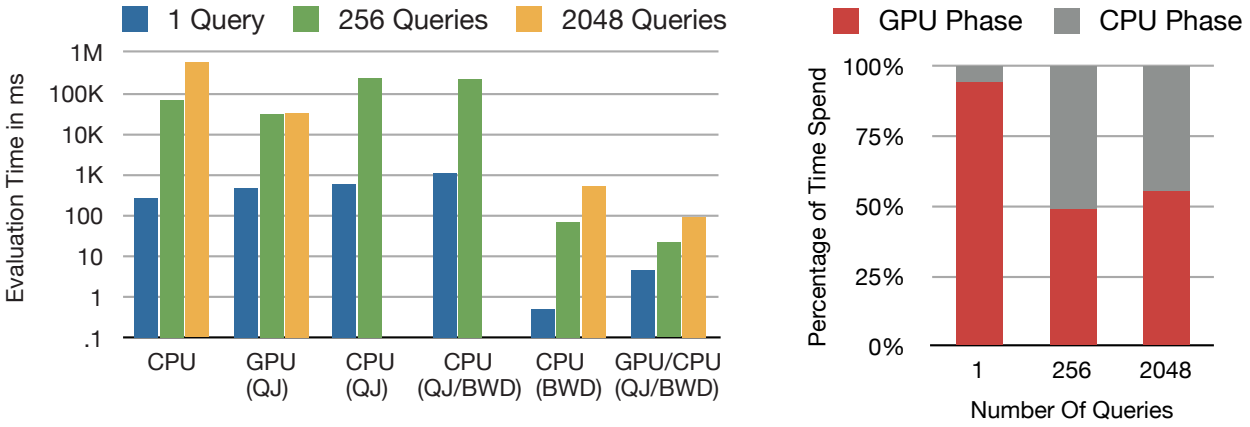
6. CONCLUSION AND FUTURE WORK

Efficient CPU/GPU processing is still an open research challenge. We presented a viable solution to this problem, tackling it by decomposing data into individual bits. Our approach outperforms current CPU/GPU co-processing strategies by more than two orders of magnitude for a spatial selection benchmark on real life data. This makes it a very attractive paradigm for relational cross device query processing.

However, we believe that there is still room for further research. Focusing on the general strategy, we deliberately abstained from rigorous optimization to the available hardware (GPU and CPU alike). We believe that with more sophisticated optimization, focusing on, e.g., GPU memory access, we may improve the performance of our implementation even further. We also believe that co-processing setups other than GPU/CPU could benefit from the approach. SSD/HDD as well as physically distributed (Client-Server) architectures are likely candidates. In addition, we also believe that a study of the approach for other applications (joins, grouping, data mining, ...) is of value.

Acknowledgments

The work reported here has partly been funded by the EU- FP7-ICT project TELEIOS. This publication was supported by the Dutch national program COMMIT.



(a) Processing Time by Evaluation Technique

(b) Load Distribution for GPU/CPU

Figure 11: Query Evaluation Performance

7. REFERENCES

[1] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[2] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[4] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[5] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th international conference on World wide web*, pages 421–430. ACM, 2009.

[6] R. Fang, B. He, M. Lu, K. Yang, N. Govindaraju, Q. Luo, and P. Sander. GPUQP: query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1061–1063. ACM, 2007.

[7] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[8] W. R. Franklin. Adaptive grids for geometric operations. In *Sixth International Symposium on Automated Cartography (Auto-Carto Six)*, pages 230–239, 1983.

[9] M. G.M. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, Ottawa, Canada: IBM Ltd., 1966.

[10] G. Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.

[11] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[13] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[14] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[15] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008.

[16] A. Munshi. OpenCL specification 1.1. *Khronos OpenCL Working Group*, 2010.

[17] C. Nvidia. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.

[18] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, 4(3):182–222, 1985.

[19] L. Sidirourgos, M. Kersten, and P. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *Proc. of the Int'l Conf. on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[20] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.

GPU Join Processing Revisited

Tim Kaldewey[§] Guy Lohman[§] Rene Mueller[§] Peter Volk[†*]
[§]IBM Almaden Research, San Jose, CA
[†]Technische Universität Dresden, Dep. of Computer Science
{tkaldew, lohmang, muellerr}@us.ibm.com peter.volk@tu-dresden.de

ABSTRACT

Until recently, the use of graphics processing units (GPUs) for query processing was limited by the amount of memory on the graphics card, a few gigabytes at best. Moreover, input tables had to be copied to GPU memory before they could be processed, and after computation was completed, query results had to be copied back to CPU memory. The newest generation of Nvidia GPUs and development tools introduces a common memory address space, which now allows the GPU to access CPU memory directly, lifting size limitations and obviating data copy operations. We confirm that this new technology can sustain 98 % of its nominal rate of 6.3 GB/sec in practice, and exploit it to process database hash joins at the same rate, i.e., the join is processed "on the fly" as the GPU reads the input tables from CPU memory at PCI-E speeds. Compared to the fastest published results for in-memory joins on the CPU, this represents more than half an order of magnitude speed-up. All of our results include the cost of result materialization (often omitted in earlier work), and we investigate the implications of changing join predicate selectivity and table size.

1. INTRODUCTION

Memory bandwidth exceeding 150 GB/s and hundreds of cores make GPUs an interesting platform for accelerating complex query processing tasks such as joins. Nvidia's *Compute Unified Device Architecture (CUDA)*, an extension of the C programming language, simplifies programming GPUs for applications other than graphics [6]. However, the use of GPUs for data-intensive operations has been limited by the amount of memory on the GPU card (≤6GB today) and the time-consuming process of copying data back and forth between CPU and GPU memory across a *PCI Express* (PCI-E) link with limited bandwidth (≤6.3 GB/s today).

Though earlier investigations exploring the use of GPUs for join processing claimed orders of magnitude speedup over CPUs [7–9], they were based on assumptions that sidestepped reality. They assumed that both the input tables *and* the join results all fit simultaneously in the GPU's limited memory. Often, they only measured the time to perform the join, omitting the non-trivial transfer times from and to CPU memory or considering them negligible, which we found not to be the case for efficient join implementations. While it was feasible to work around the memory limitation by partitioning the input tables, overlapping data copies and processing to effectively use the available PCI-E bandwidth has proven challenging. Recent work [12] on offloading hash probes to the GPU identified data copying as the dominant cost (≥ 50 %) that limited effective throughput to 50 % of the available PCI-E bandwidth.

The latest generation of Nvidia GPUs and development tools adds a common address space for the CPU and GPU called *Unified Virtual Addressing* (UVA), which allows the GPU to access the CPU-side memory directly. This not only lifts the size limitations on data sets that can be processed, but also relieves the programmer from the burden of managing two address spaces and copying data back and forth. More importantly, UVA can sustain 98 % of the nominal rate of PCI-E; we measured 6.2 GB/s in practice. UVA enables the GPU to process arbitrarily large tables in CPU memory at PCI-E speeds, without managing data transfers or multiple copies.

This paper shows how UVA can be leveraged to accelerate database join processing, achieving throughput rates of up to 6.1 GB/s, making optimal use of the available PCI-E bandwidth (Sec. 3). This amounts to a speedup of more than half an order of magnitude compared to previously published results for in-memory join operations on the CPU [10]. We provide an end-to-end performance analysis of relational joins on GPUs, i.e., including the cost of reading the input data sets from CPU memory and materializing results. Our GPU results concur with prior work on in-memory joins on the CPU [3] that a conventional (non-partitioned) hash join works best, due to its simplicity. We also analyze the sensitivity of our implementation to table size and join predicate selectivity. Our results show that the table size has marginal impact upon throughput, whereas selectivity of the join predicate significantly affects performance (Sec. 4). We propose a result cache to accelerate joins that produce a large number of results. Since the performance of database operations is known to be dominated by memory performance [1, 4], we provide a detailed evaluation of GPU memory performance exploiting UVA (Appendix A, B).

2. GPU BACKGROUND

This section provides a brief tutorial on GPU architecture

*Work done while the author was at IBM

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

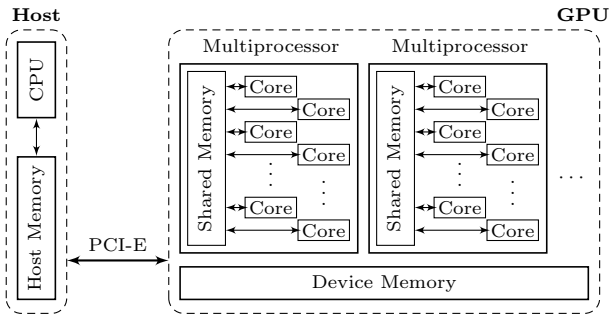


Figure 1: Architecture of an Nvidia GPU.

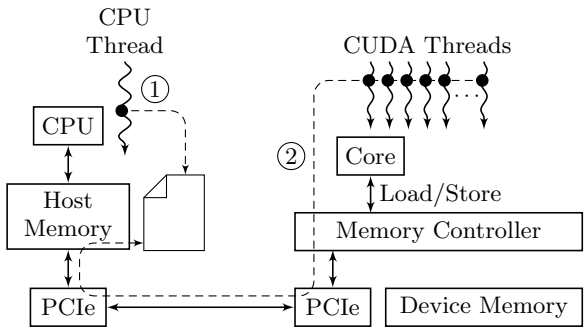


Figure 2: UVA Read access from the GPU Kernel into host memory. (1) CPU thread pins page. (2) CUDA threads execute a load or store instruction, which results in a memory read or write transaction on the PCI-E link to the pinned page.

and the new UVA enhancement, as it pertains to database joins. For a more detailed discussion of relevant memory access patterns, please refer to the Appendix.

A GPU application written for CUDA consists of two parts: a "control" thread running on the *host* (CPU) and a parallel part, called a *kernel*, running on the GPU. The kernel is organized as a number of *thread blocks*, with each block running all its threads on the *CUDA cores* of one *streaming multiprocessor* (SM), see Figure 1. CUDA cores within the same SM execute the same instruction in lockstep, in *Single Instruction-Multiple Data* (SIMD) mode. Threads within the same thread block can access a low-latency on-chip *shared memory* on an SM. For example, the GTX 580 GPU we use in our experiments has 16 SMs, and each SM has 32 CUDA cores and 48 kB of shared memory.

Host memory access. Data is transferred between the host memory and the GPU over a PCI-E link, which puts an upper bound on the throughput of data-intensive operations. Before UVA was introduced with Nvidia's Fermi architecture and CUDA 4.0, this had to be done through explicit host-initiated `cudaMemcpy()` calls, hereafeter referred to as `memcpy`. Now, UVA allows a compute kernel to directly access host memory, which is fetched over the PCI-E link as needed. The concept of UVA is illustrated in Figure 2. In step (1), the CPU thread of a CUDA application pins a page in host to obtain a real address pointer to this page from the operating system. This real address pointer is then passed to the GPU during the kernel invocation. In step (2), the CUDA threads can now issue load and store instructions using this real address pointer. The memory controller on the GPU distinguishes this access from requests to the device memory and performs the corresponding memory read or write transaction over the PCI-E link. The *PCI Express Root Complex* on the host side then issues an access to the host memory. The 16× PCI-E 2.x connection of our GPU has a nominal bandwidth of 6.3 GB/s. We measured an effective bandwidth of 6.2 GB/s for 64-bit read and write accesses to host memory through UVA. Our measurements show that at least 1,024 threads and 16 blocks are required (Appendix B), which places a lower bound on the parallelism required for an efficient hash join implementation.

Device memory access. Hash joins are known to produce irregular memory access patterns during parallel hash table creation and probing. In particular, compare-and-swap for inserting tuples into the hash table and quasi-random, data dependent reads for hash table probes are problematic. On our GTX 580 GPU we measured 7.7 GB/s for 64-bit random read accesses to 512 MB of device memory. For 64-bit compare-and-swap accesses to random locations we measured 4.6 GB/s. To achieve these bandwidths the compute kernel needs to be launched with at least 16 thread blocks and at least 8 threads/block (Appendix A).

3. IMPLEMENTATION APPROACHES

To determine the optimal join algorithm for GPUs, we ported the best known in-memory join algorithms from recent studies of CPU joins: a traditional hash join [3] and a partitioned variant of hash join. The latter algorithm partitions the hashed values of the input keys in such a way that each partition fits into the processor cache, to avoid random accesses to memory when joining each partition [10]. We then evaluated both hash algorithms, first explicitly copying data from and to the CPU, and secondly exploiting UVA. Our GPU implementation of a conventional (parallelized) hash join using UVA outperformed all other approaches and achieved half an order of magnitude speed-up over the partitioned CPU hash join.

Implementation Details. Our parallel join implementations on the GPU do not differ fundamentally from their CPU counterparts. The core algorithms are identical, but the parallelization, data placement, and intermediate data structures are GPU-specific. For example, in the partitioned hash join, the GPU's 48 kB of shared memory takes over the role of the CPU 's caches. The remaining parameters are the same as in prior work [3, 10] to allow performance comparisons. For example, we use the least significant bits (LSB) of the input key as the hash function, and an open addressing hash table two times the size of the input table.

Similar to a conventional CPU hash join implementation, we create a hash table from the *build table* and then probe it with the entries from the *probe table*. As in earlier work [2,9], we build the hash table in the device memory of the GPU. As discussed in Section 2, achieving high throughput requires a high degree of thread parallelism. For example, accessing the input tables located in host memory requires at least 1 k threads to be efficient. Obviously, this requires coordinating write access to shared data structures. We use atomic operations to efficiently coordinate multiple threads concurrently inserting data into the hash table, e.g., compare-and-swap, and into the result set, e.g., an index that is incremented atomically.

Using pre-UVA techniques that perform `memcpy` opera-

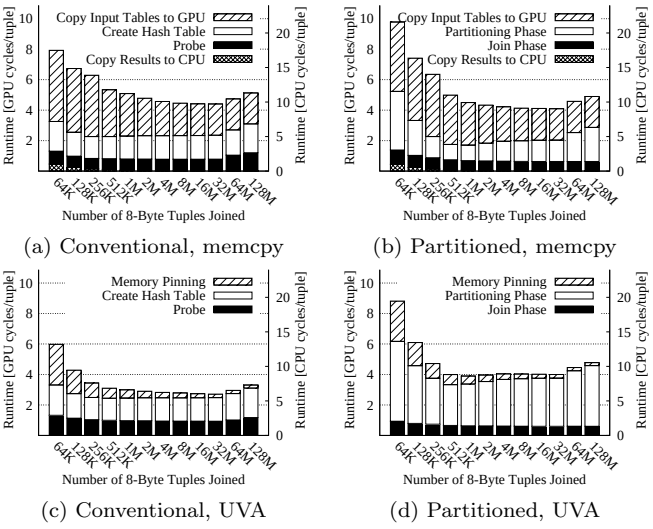
tions, a conventional hash join algorithm copies the build table from host to device memory, creates a hash table in device memory, deletes the build table from device memory, copies the probe table to device memory, probes the hash table, stores the results in device memory, and finally copies them back to host memory. This approach is limited by the combined size of its working data, i.e., the build table (or horizontal partition thereof) and its hash table during build, and during probe the hash table, probe table (or horizontal partition thereof), and results all must simultaneously fit into device memory. It also requires passing control of the program execution back and forth between the GPU and CPU, as `memcpy` operations are controlled by the CPU.

Using UVA, the GPU can read the build table directly from host memory while it creates the hash table in device memory, so it never has to store the build table in the device memory. The probe phase reads the rows of the probe table directly from host memory, probes the hash table stored in device memory, and again stores the results directly in host memory. The only limiting factor is the size of the hash table, which has to fit into device memory ($\leq$6 GB). Just as a CPU join may have to spill portions of its hash table to disk if it exceeds memory, the GPU may have to spill its hash table to host memory using UVA. Implementing support for larger hash tables and evaluating the impact of spilling on performance is future work.

The partitioned hash join described by Kim et al. [10] can also be ported to the GPU with relatively few modifications. The histogram(s) that are used to determine the target location of data in the partitioned data set can be created in shared memory to avoid the performance penalties of frequent device memory accesses. The size of shared memory (48 kB) limits the number of partitions that can be processed in one pass to 12 k (48 kB / 4 bytes per histogram entry) and the partition size to 16 kB (assuming a conservative 50% load factor for the hash table). Therefore, tables larger than 192 MB require multi-phase partitioning.

Data Sets. To compare our results with prior work [3, 10], we join two equally sized tables, with tuples comprised of a 32-bit key and a 32-bit row identifier. Choosing two tables of equal size represents the worst case scenario for a hash join, as the hash table is usually created from the smaller of the two tables, to optimize memory consumption and performance. We use a uniformly distributed, randomly generated data set, again the worst case scenario, as there will be no locality of reference.

We control the result size by varying the *match rate* of the join predicate, which is defined as the percentage of keys in the probe table that have matching key(s) in the build table. [1] Low match rates correspond to joining domains that barely intersect, whereas referential integrity would guarantee a 100 % match rate. We generate the data set such that for each tuple in the probe table, we either randomly select a tuple from the build table or generate a random value that is outside of the key space, according to the match rate. If both tables were simply filled with randomly-generated 32-bit integers, as done by some papers, the match rate could not be controlled, and would be unrealistically low, based upon the probability of generating the same random 32-bit integer in both tables. Other than one experiment that explicitly examines the impact of varying the match rate on performance, our evaluation uses a match rate fixed at 3 %, to be compatible with prior work [10], though we believe it to be unrealistic.



(a) Conventional, memcpy (b) Partitioned, memcpy

(c) Conventional, UVA (d) Partitioned, UVA

Figure 3: Comparing the runtime of conventional and partitioned hash join implementations, using data copies vs. a uniform address space.

Conventional vs. Partitioned Hash Join. Using the traditional GPU programming approach that requires data copies, conventional and partitioned hash joins achieve nearly identical performance. Figure 3(a) and (b) show the execution time per tuple in GPU cycles (left y-axis) and CPU cycles (right y-axis)[2] for increasing number of tuples joined, i.e., the combined size of the build and probe tables. Though performance is comparable (4–5 GPU compute cycles per tuple) for larger data sets, for smaller data sets (<512 K tuples), the conventional hash join achieves slightly better performance (6–8 GPU cycles/tuple) than the partitioned one (7–10 GPU cycles/tuple). Obviously, the cost of data copies, labeled "Copy Input Tables to GPU" for transferring the input tables and "Copy Results to CPU" for transferring the results, are identical for both algorithms. For data sets larger than about 8 M tuples, the cost of initiating input data transfers is dominated by the PCI-E bandwidth, which we measured at 2 GPU cycles/tuple or 6.1 GB/s. The same applies to copying join results back to host memory, except that it occurs earlier, for data sets larger than 512 k tuples, because we assumed highly selective (3 %) joins. Aside from the cost of copying the input tables, the execution time of a conventional hash join depends on both hash table creation (labeled "Create Hash Table") and probing (labeled "Probe"). The rate of hash table creation is governed by the efficiency of compare-and-swap operations, while the speed of probes is limited by random reads from the device memory (Appendix A). The time to read the input tables from device memory is negligible, as these reads are coalesced and achieve rates above 150 GB/s (Appendix A). The perfor-

[1] Though other papers have called this "selectivity", it is *very* different from selectivity as used in the query optimization literature. We use it only so we can compare our results to prior work.

[2] We include the latter metric to enable a direct comparison with prior work on CPU joins, since there is a more than 2× difference between GPU (1.5 GHz) and CPU (3.4 GHz) clock frequency.

mance of probes is directly correlated with that of random read accesses to device memory, which decreases for data set sizes of 256 MB or more (see Appendix A).

The partitioned hash join described by Kim et al. [10] requires three passes over the input tables, one to create a histogram to compute the partition boundaries, one to move the input data to the target partitions, and one to join the subtables. The first two are part of the "Partitioning Phase" and the third is required for the "Join Phase". As with conventional hash join, the input data can be read from device memory at up to 150 GB/s. However, moving data to the partitions requires random writes, which is the dominating cost of the partitioning phase. On the other hand, all random accesses required by hash table creation and probing during the join phase are to fast shared memory, as partitioning is done so that a partition fits into shared memory. Therefore, the overall cost of the join phase is relatively small (< 1GPU cycle/tuple). The increases in execution time for the partitioned hash join on data sets of 64 M tuples (256 MB per table) and larger are the result of 2-phase partitioning, which requires double the scans(4), but more importantly two (random) rewrites of the data set.

Data Copy vs. UVA. Using UVA's uniform address space obviates data copies, since the input tables remain in host memory and results are written directly to host memory. UVA requires the input tables to be pinned in physical memory. While the (per tuple) cost of pinning the input tables in host memory amortizes over larger data sets (Fig. 3(c),(d)), reading the input tables through UVA directly from host memory puts an upper limit on overall throughput (≤ PCI-E bandwidth).

For a conventional hash join, we observe from Figure 3(c) that using UVA to access the input tables has no impact on hash table creation and negligible impact on probing. Compare-and-swap operations are slower than the PCI-E link, and so remain the limiting factor of hash table creation. However, random accesses to device memory perform slightly better than PCI-E bandwidth, so probing is limited by the PCI-E bandwidth.

The partitioned hash join implementation cannot take advantage of UVA (see Fig. 3(d)), as it requires several passes over the input tables, which with UVA remain stored in host memory. Host memory access is more than an order of magnitude slower than device memory access—6 GB/s vs. 150 GB/s. The overall performance is almost exactly the same as using explicit data copies, with the time previously spent on copying the input data now included in the partitioning phase. We conclude that, in the presence of UVA, partitioning input tables does not provide an advantage over a simple hash join implementation, an observation that conforms with prior results comparing CPU implementations of these algorithms [3].

GPU vs. CPU. Our "conventional" GPU hash join using UVA achieves an end-to-end throughput of roughly 3 compute cycles per tuple (Fig. 3(c)), which is about an order of magnitude faster than results reported from the best known parallel CPU implementation [10] (32 CPU cycles/tuple). However, using cycles per tuple as a metric does not permit an "apples-to-apples" comparison, as clock frequencies differ significantly, e.g., our GPU is clocked at 1.5 GHz, versus our CPU at 3.4 GHz and the 3.2 GHz CPU used in [10]. Allowing for the difference in clock frequency between our GPU and CPU by comparing the vertical scale on the right of Figure 3 (measured in CPU clock cycles per tuple), our GPU implementation is still more than half an order of magnitude faster than the best parallel CPU implementation. [3]

4. EXPERIMENTAL EVALUATION

Having determined the best implementation for our GPU join to be a conventional hash join using UVA to directly access the host memory, we now evaluate the efficiency of that implementation, and how sensitive its performance is to table sizes and the selectivity of the join predicate. We also evaluate the benefit of a result cache to reduce contention when materializing results. To identify inefficiencies or bottlenecks independent of a specific machine's architecture or performance characteristics, for the remainder of this paper we measure performance and compare it to the nominal hardware capabilities in terms of throughput, i.e., the number of input bytes processed per second.

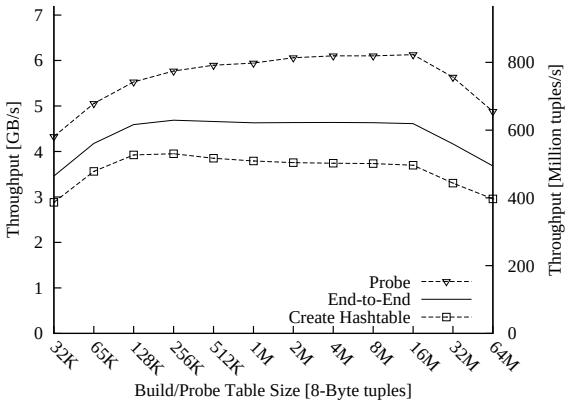


Figure 4: Throughput of GPU hash join with respect to input table size(s), with equally sized input tables accessed via UVA, and match rate fixed at 3 %.

Efficiency. Based on the same experiment as Figure 3(c), Figure 4 depicts the throughput of build phase, probe phase, and overall, end-to-end query execution as a function of the size of the input tables. The end-to-end throughput is the geometric mean of the two stages of our hash join — hash table build and probe. We observe an overall performance of up to 4.6 GB/s, and little sensitivity to the table sizes. Hash table probes can be executed at up to 6.1 GB/s, the maximum rate at which the probe table can be accessed across the PCI-E link (Appendix B), while hash table creation is significantly slower, reaching at most 3.9 GB/s. The limiting factor for parallel hash table creation is the locking required to prevent parallel threads from overwriting existing hash table entries. Our implementation uses compare-and-swap to manage concurrent hash table access (Sec. 3). The 18 % performance difference between the resulting hash table build rate (3.9 GB/sec) and compare-and-swap throughput (4.6 GB/s) can be attributed to the time necessary to initialize the hash table and to resolve hash collisions. In fact, excluding the time it takes to initialize the hash table, we measured hash table creation rates up to 4.1 GB/s, and

[3]Running the partitioned CPU hash join on our quad-core i7 Sandy Bridge CPU, making use of all of its eight hardware threads, we observe a 20% performance improvement over the previous CPU generation [10].

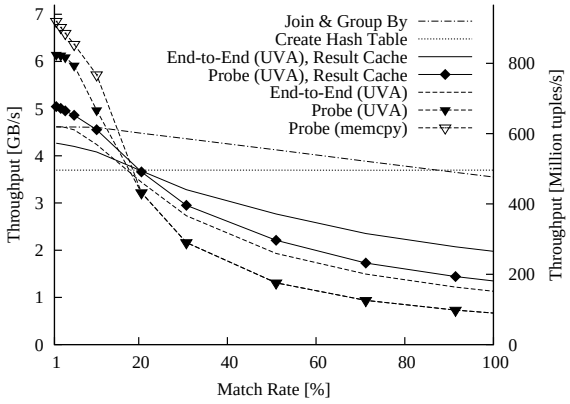


Figure 5: Throughput of GPU hash join as a function of percent of rows finding matches, with equally sized input tables of 16M tuples accessed via UVA

with a perfect hash that produces no collisions[4] we observe rates up to 4.5 GB/s, close to optimal.

Match Rate. The previous experiments used a match rate of 3 %, which made the time to write these few results back to host memory negligible, compared to the time required to read the probe table from memory. Other work on in-memory joins on the CPU suggests that the time required to materialize join results can be neglected, as it only adds a constant overhead of a few compute cycles per result [3, 10, 11]. However, given that our best GPU join implementation requires only 3 cycles per tuple (Fig. 3(c)), a few extra cycles can add substantial overhead when more probes return results.

Figure 5 confirms that join predicates that have a higher percentage of matches significantly reduce GPU join throughput. We ran a conventional hash join, labeled "End-to-End (UVA)", for data sets of 16 M tuples per table accessed via UVA, and varied the match rate from 1 % to 100 %. Though earlier papers assumed a very optimistic match rate of 3 %, the more common situation of a join between a foreign key and a primary key would result in a 100 % match rate. [5] For join predicates having match rates less than 5 %, throughput is high, but beyond this point the throughput drops steeply from 4.6 GB/s to 1 GB/s.

When the join predicate has a low match rate, overall throughput was limited by hash table creation, labeled "Create Hash Table" in Figure 5, but as the match rate increases, throughput becomes dominated by the probe phase, labeled "Probe (UVA) ". Obviously, the join predicate's match rate does not impact hash table creation, so we will limit further investigation to the performance of hash table probes.

Although PCI-E is bidirectional, the GPU we used in our test system can only transfer data in one direction at a time, as it only has one DMA engine. [6] To determine the performance impact of simultaneously probing the hash table and transferring results across the PCI-E link, we repeated the experiment above with input tables stored in device memory and results written to device memory as well. To our surprise, the curve labeled "Probe (memcpy)" in Figure 5 shows that using device memory exclusively provides only marginal improvements over UVA, which indicates that UVA is not a bottleneck here.

Our measurements in Appendix B show that we need to launch at least 1 k threads to make efficient use of the available PCI-E bandwidth. The less selective the join predicate is, the more of these 1 k GPU threads that will find a match and will need to write its result contiguously to host memory at the same time, inevitably creating contention for the write coordinator. Although we coordinate writes as efficiently as possible, using an index on an array that is incremented atomically, as join match rates approach 100 %, 1 k threads access this index simultaneously, creating a bottleneck.

Result Cache. Reducing the number of threads attempting to write results to the same data structure will reduce this contention. To accomplish this, we implemented a result cache that uses on-chip shared memory (cf. Fig. 1) to stage results before they are written to host memory. With this cache, we only need to coordinate the writes of threads within a single block, as only they have access to the same shared memory (on the same SM). Moreover, once the result cache fills up, we use all of those threads to write (flush) the cached results in parallel, which allows for efficient coalesced writes. On the downside, every time we flush the result cache, we need to synchronize all threads within that block before we can write its content back to host memory. Since shared memory and therefore our cache is limited to 48 kB, for joins with high match rates we have to flush the cache more frequently, which again requires an atomic operation on the UVA-accessible memory. Nevertheless, our result cache reduces the pressure on the atomic insert into the result table by more than three orders of magnitude, as we can cache up to 6 k results.

Our result cache doubled the throughputs of probe and overall, labeled "Probe (UVA), Result Cache" and "End-to-End (UVA), Result Cache", respectively, in Figure 5. Although the result cache marginally reduces overall throughput for joins with match rates < 10 %, the result cache improves throughput thereafter, with more gradual decreases from > 4 GB/s for 10 % match rate to 3 GB/s for 50 % match rate. Even for 100 % match rates, our implementation still achieves 2 GB/s end-to-end throughput, which is nearly three times faster than the fastest reported in-memory CPU join [10][7], which assumed a very optimistic 3 % match rate and did not include the cost of result materialization in its performance evaluation.

Operator Pipelining. Prior work on GPU joins [12] suggests that a join is often followed by a group by (and aggregate) operator. Pipelining the join results into the group by operator so that the same set of threads will handle the aggregation as well, avoids materializing the results in host memory and the locking associated with it. The curve labeled "Join & Group By" in Figure 5 depicts a scenario in which each thread computes a local aggregate at its end and is added to a global aggregate located in host memory using an atomic add. For this scenario, overall throughput grad-

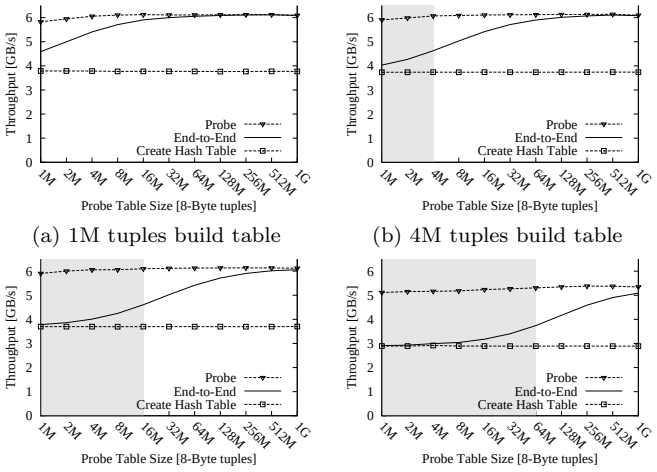
[4]For the LSB hash we are using for our experiments, creating a build table with consecutive even or odd integers as keys satisfies this condition.

[5]We are only considering the effect of the join predicate itself, and assume that any predicates local to individual tables have already been applied.

[6]Professional video cards like Nvidia's Quadro series or specialized GPU compute cards like the Tesla series have two DMA engines enabling bidirectional use of the PCI-E link.

[7]32 cycles/8-byte tuple at 3.2 GHz are the equivalent of 0.75 GB/s.

ually decreases from 4.6 GB/s for match rates of 1–5 % to 3.6GB/s for 100 %.



(a) 1M tuples build table (b) 4M tuples build table

(c) 16M tuples build table (d) 64M tuples build table

Figure 6: Throughput of GPU hash join for increasing input table sizes, tables accessed via UVA and match rate fixed at 3 %.

Table dimensions. While our prior experiments focused on the worst case scenario for (hash) joins, i.e., building and probing tables of equal size, we now take a look at the common case in which the build table is smaller than the probe table [3]. Figure 6 depicts the performance of our GPU join implementation for build tables of size 1, 4, 16, and 64 million tuples, with increasing probe table size from one million to one billion tuples, and a constant match rate of 3%. The rates of hash table creation (labeled "Create Hash Table") and probe (labeled "Probe") are virtually constant at 3.9 GB/s for building hash tables for 1 M, 4 M, 16 M tuples, and 6 GB/s for probes, independent of the probe table size. The overall throughput (labeled "End-to-End") is the geometric mean of the two, weighted by the number of tuples. Hence, with increasing probe table size, performance becomes dominated by probing.

Figures 6(c) and (d) leave the impression that for smaller probe tables, performance is dominated by hash table creation. This is an artifact of using the same range for probe table sizes across all experiments (a–d), and always creating the hash table based on the table identified as the build table, regardless of size. Typically, an optimizer would choose the smaller table to build the hash table, both to save memory and because the hash table creation rate is slower than the probe rate. For example, building a hash table on a 16 M-tuple table and probing it with a table with 1 M tuples reduces overall throughput to the rate of hash table creation, 3.9 GB.s, as shown in Fig. 6(c). Conversely, building a hash table on the 1 M-tuple table and probing it with the table with 16 M tuples yields far better performance of 6 GB/s, as shown in Fig. 6(a). The grayed-out areas in Figure 6 mark table-size combinations that an optimizer would reject as sub-optimal, in favor of exchanging the build and probe tables to get better throughput.

5. CONCLUSIONS

In this paper we have shown how to efficiently offload large-scale relational join operations to the GPU, achieving a sustained overall throughput of 2 GB/s to 6.1 GB/s depending upon the size of the result set. This corresponds to 3–8× performance improvement over the fastest reported implementation on CPUs [10]. Adjusting for the current CPU generation still leaves us with more than half an order of magnitude speed-up. UVA allows us to achieve throughput rates close to the hardware capabilities, without requiring any hand tuning. Moreover, the input tables remain in host memory, obviating the need to manage data copies. In the typical scenario for a hash join, i.e., when the probe table is larger than the build table, our GPU hash join implementation was able to achieve 96 % utilization of the PCI-E generation 2 bandwidth (6.3 GB/s). PCI-E generation 3 doubles that bandwidth, and we expect the next generation GPUs to be able to take advantage of it. Future work includes support for handling larger data sets, e.g., spilling large hash tables to CPU memory and reading input tables from external storage, and processing more complex queries.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB'99*.

[2] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Ameta. *GPU Computing Gems: Jade Edition*, chapter 4, pages 39–53. Morgan Kaufmann, 2012.

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD'11*.

[4] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99*.

[5] R. Budruck, D. Anderson, and T. Shanley. *PCI Express System Architecture*. Addison-Wesley, 2003.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4).

[7] N. K. Govindaraju and D. Manocha. Efficient relational database management using graphics processors. In *DaMoN'05*.

[8] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), Dec. 2009.

[9] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD'08*.

[10] 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. *Proc. VLDB Endow.*, 2(2), Aug. 2009.

[11] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowledge and Data Engineering*, 14.

[12] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS'11*.

Table 1: Measured Bandwidth to Device Memory on the GTX 580 (512 MB accessed by 1,024 blocks with 1,024 threads/block).

		Bandwidth GB/s		
		coalesced	non-coalesced	random
READ (data-dep.)	32 bit	122	4.8	3.8
	64 bit	151	10	7.7
READ (data-indep.)	32 bit	119	4.8	3.8
	64 bit	152	10	7.7
WRITE	32-bit	144	5.4	3.8
	64 bit	155	11	7.7
CAS	32-bit	8.6	2.4	2.3
	64 bit	11	4.9	4.6

APPENDIX

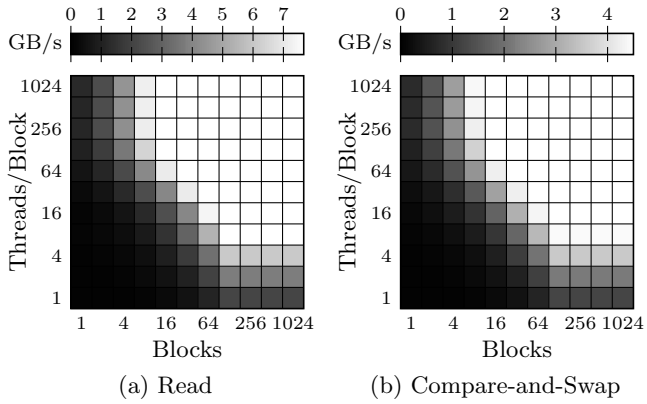
System Configuration. Throughout our experiments, we use a high-end consumer graphics card, a GeForce GTX 580. Its GF110 core is clocked at 1.54 GHz and encompasses 512 CUDA cores in 16 streaming multiprocessors with 32 CUDA cores each, and has access to 3 GB of GDDR5 device memory. The card was installed in a system with an Intel Core i7-2600 Sandy Bridge CPU clocked at 3.4 GHz and 16 GB of DDR3-1333 memory. We ran Suse Enterprise Linux 11.1 with a 2.6.32 kernel Nvidia graphics driver 285.33, and CUDA toolkit 4.1 installed. The system was dedicated to the experiments, i.e., with no other users or applications active. The video output on the GTX 580 was disabled and system's video output was handled by another graphics card.

A. DEVICE MEMORY ACCESS

Table 1 shows the measured bandwidth to the 3,072 MB device memory on the GTX 580 for different access patterns. The table reflects the well-known property of GPU memory; the effective bandwidth highly depends on the pattern and the type of memory access. In coalesced memory access, adjacent threads are accessing adjacent memory locations, e.g., thread t references memory location m, thread $t+1$ location $m+1$, etc. Non-coalesced accesses, e.g., where a single thread t references consecutive locations $m, m+1, \ldots, m+k-1$ should be avoided since they result in 15–25× lower memory throughput.

In this setting, we are considering read, write, as well as atomic *compare-and-swap* (CAS, `atomicCAS()`) operations on 32- and 64-bit data. Reads are further divided in data-dependent and data-independent accesses. In the former, the next memory location accessed depends on the content of the currently accessed location. Such a pattern is exhibited, for example, when traversing a linked list.

For our join implementation random access operations, that occur when accessing hash table entries, are of particular interest. Random CAS is used to coordinate the insert operations done concurrently by hundreds of threads during the create phase of the hash table. The random read happens during the probe phase of the hash join when the key and payload entries are looked up in the hash table. For 64-bit random accesses we note a peak bandwidth for reads of 7.7 GB/s and 4.6 GB/s for CAS.[8]



(a) Read (b) Compare-and-Swap

Figure 7: Measured bandwidth of 64-bit accesses to random locations within 512 MB of device memory on the GTX 580.

Figures 7(a) and 7(b) show the measured aggregate read and CAS bandwidth that can be obtained for different GPU thread configurations. In CUDA, a compute kernel running on the GPU is always executed in a grid of thread blocks. In particular, a thread block is mapped to one streaming multi-processor, and therefore, all threads of that blocks are executed by the CUDA cores on this multi-processor. The figures show that for both read and CAS a minimum number of thread blocks and threads/block is needed to achieve maximum performance. For reads and CAS at least 16 blocks have to be scheduled to achieve a peak bandwidth of 7.7 GB/s and 4.6 GB/s. This requirement is quite obvious; starting with 16 blocks all 16 multi-processor are used. Inside of a thread block we need at least 8 threads on this GPU to reach peak bandwidth for reads and CASs.

Similar to SMT in traditional CPUs, memory latency can be hidden by mapping more parallel activity on a core. In this case, while one thread (warp of threads) is waiting for data another thread (warp of threads) can be executed. Latency can be hidden (and throughput improved) when another thread is executed in the meantime. This interleaving can be applied all the way down to the actual DRAM access, where a column access in a bank can be interleaved with opening a new row in another currently idle bank.

The diagonals in Figures 7(a) and 7(b) correspond to a total of 1,024 threads. Hence, having accesses of at least 1,024 threads in-flight is necessary to keep all queues in the memory controller full. We cannot observe any degradation as we move towards the upper right corner in the figures by increasing the total number of threads. For this reason we can further increase the number of threads/block and block count in our join implementation as needed. In more complex GPU kernels than those used in these benchmarks, other resource constraints limit the number of blocks and threads/block that can be executed concurrently due to the register usage of a thread and the shared memory requirement of a block.

In Figure 8 we show the measured bandwidth for differ-

[8]For comparison to a CPU, the quad-core Sandy Bridge i7-2600 with DDR3-1333 memory in our experiments has an aggregate bandwidth of 700 MB/s for random 64-bit reads and 380 MB/s for 64-bit CAS using 8 threads.

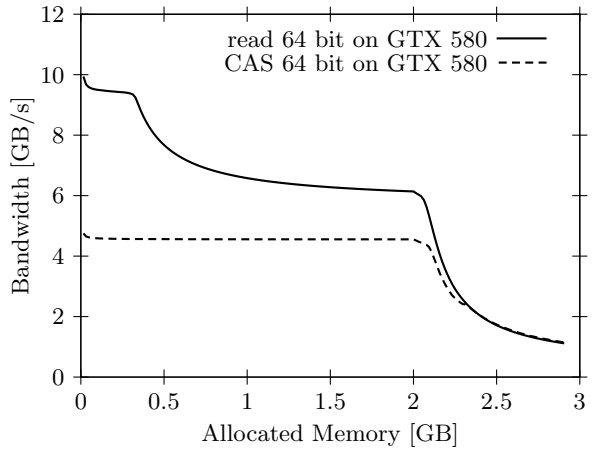


Figure 8: Measured bandwidth of accesses to random locations in device memory on the GTX 580.

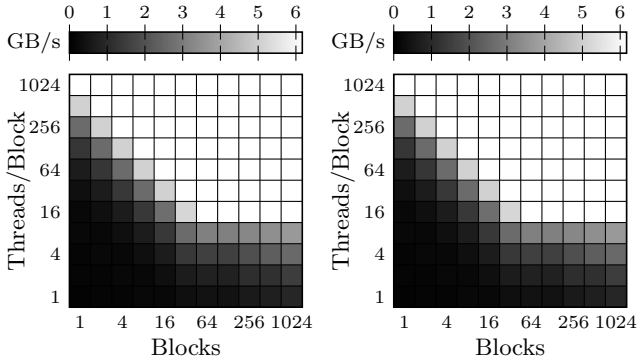


Figure 9: Measured bandwidth of 64-bit coalesced accesses to host memory using UVA for different CUDA grid and block configurations.

ent random access patterns for different amounts of allocated device memory. An interesting observation on our 3 GB GTX 580 is the bandwidth drop for random reads at 256 MB and a further sharp drop for all patterns when more than 2 GB/s are randomly accessed. We are unable to explain this behavior using publicly available data. However, we believe that the following provides an explanation for the performance drop in Figure 4 for table sizes ≥ 32 M tuple: If the GPU uses a common queue for both device memory and UVA accesses, the bandwidth drop for device memory accesses could slow down the UVA access and reduce the overall throughput even though the actual device memory bandwidth is still well above the PCI-E limit.

B. HOST MEMORY ACCESS

For the hash join algorithm UVA-based memory accesses is used in both the create and the probe phase to read the input tuples. The join results are written back to host memory via UVA as well. In both cases, memory accesses are coalesced. Figures 9(a) and 9(b) show the memory bandwidth for 64-bit read and write access to host memory through UVA for different thread configurations. We are able to obtain a sustained throughput of 6.17 GB/s for both access types. The figures show that in order to achieve peak bandwidth, at least 16 threads/block need to be used. This can be explained by the DMA transfer sizes supported by the GPU. The PCI-E endpoint of the GTX 580 supports a maximum payload size of 128 bytes in PCI-E Transaction Level Packets[9]. Using 16 threads, 16×64-bit requests translate into full 128 bytes payloads in the PCI-E packets. The diagonals (cf. Fig. 9) correspond to total of 1,024 threads that are required to achieve peak bandwidth. In other words, having accesses of at least 1,024 threads in-flight is required to keep the memory queues filled. Figures 9(a) and 9(b) show that we cannot observe any degradation as we move towards the upper right corner, and the total number of threads increases.

The measured bandwidth is close to the theoretical peak bandwidth for a 16-lane PCI-E generation 2 device as the following simple calculation shows: A single PCI-E generation 2 lane operates at 5 GT/s. Given the 8b/10b symbol encoding used in the PCI-E physical link, the nominal bandwidth thus is 500×10^6 B/s for a single lane and 8×10^9 B/s for all 16 lanes. With a 24 byte packet overhead,[10] the packet sent for a single 128-byte request is 152 bytes in size [5], and hence, the theoretical peak-bandwidth $8 \times 10^9 \text{ B/s} \times {}^{128}/_{152} \approx 6.27$ GB/s. The results in Figures 9(a) and 9(b) show that the measured bandwidth is 98 % of the theoretical peak value.

[9]The maximum payload size in a transaction level packet supported by a PCI-E device can be read from the *Device Capability Register*, for example, using `lspci -vv`.

[10]For a 24 byte packet overhead we assume that 64-bit host addresses are used without the optional end-to-end CRC field.

GiST Scan Acceleration using Coprocessors

Felix Beier*
Ilmenau University of Technology
felix.beier@tu-ilmenau.de

Torsten Kilias†
Ilmenau University of Technology
torsten.kilias@tu-ilmenau.de

Kai-Uwe Sattler
Ilmenau University of Technology
kus@tu-ilmenau.de

ABSTRACT

Efficient lookups in huge, possibly multi-dimensional datasets are crucial for the performance of numerous use cases that generate multiple search operations at the same time, like point queries in ray tracing or spatial joins in collision detection of interactive 3D applications. These applications greatly benefit from index structures that quickly filter relevant candidates for further processing. Since different lookup operations are independent from each other, they might be processed in parallel on modern hardware like multi-core CPUs or GPUs. But implementing efficient algorithms for all kinds of indexes on various hardware platforms is a challenging task. In this paper, we present a new approach that extends the existing GiST index framework with an abstraction layer for the hardware where index operations are executed. Furthermore, we provide first performance evaluations for the scan execution on CPUs and an Nvidia Tesla GPU.

1. INTRODUCTION

Efficient search operations in huge, possibly multi-dimensional, datasets are crucial for the performance of numerous use cases in a wide range of applications. For example in computer graphics 2D images have to be created from 3D models, consisting of billions of triangles as most common primitive, to display them on a screen. Ray tracing [17] is a well-known technique for this rendering process. For each pixel the path of light is simulated with rays that are shot into the scene to calculate its color. Each ray requires a search operation (point query) to determine the intersection point with visible objects and light sources. To obtain photo-realistic results, this process can be repeated recursively but leads to an exponential growth in the number of search operations.

Another use case is collision detection [13] which is required, e.g., in physical simulations. The task is to determine if two or more 3D models do collide, i.e., if parts of them intersect or not. This is an example for a (spatial) join operation where, naively, the intersection test has to be performed for each triangle of the first model against all other triangles of the second one, requiring a lookup operation each.

Those, and other examples massively benefit from well-known index structures like R-trees [6] or B-trees [3] to speed up lookups. Furthermore, they generate many independent search operations at the same time which can be processed in parallel. Hence, they might benefit from recent trends in hardware development like multi-core CPUs, GPUs, FPGAs, etc. which provide the possibility to process many computations in parallel. To efficiently utilize these hardware components, a fine grained parallelism is required and several characteristics like memory hierarchies, explicit memory transfers, or different processing models have to be considered.

Therefore, the efficient implementation of numerous index structures is a challenging task, requiring a lot of development resources. Especially in scenarios where the actual structure and properties of the data is unknown in advance, e.g., in scientific contexts [1], it is a problem to decide which index type shall be implemented. It might be necessary to prototype and evaluate several to find the best one for each case. Frameworks like GiST [9] support the rapid development of new custom index structures. In the following, we present a novel approach to extend GiST with an abstraction layer for the hardware where tree operations are actually executed. We focus on scan operations in scenarios where numerous search operations have to be processed simultaneously and provide first performance evaluations for their execution on CPUs and an Nvidia Tesla GPU.

2. BACKGROUND & RELATED WORK

2.1 Index Frameworks

The approach presented in this paper is based on the GiST index framework [9] which eases the development of height-balanced index tree structures. Nodes in a generalized search tree (GiST) consist of an array of (key, ptr)-entries where *key* denotes the search key and *ptr* a pointer to the indexed

*This work is partially funded by the TMBWK ProExzellenz initiative, Graduate School on Image Processing and Image Interpretation.

†Torsten Kilias receives funding from the Federal Ministry of Economics and Technology (BMWi) under grant agreement "01MD11014A, 'MIA-Marktplatz für Informationen und Analysen' (MIA)". Research was conducted while Torsten Kilias was working with University of Technology Ilmenau.

DaMoN'12 May 21 2012, Scottsdale, AZ, USA

data or subtree nodes. To develop a new index structure, the key data structure has to be specified as well as some key-dependent methods. The most important one for the present approach is the *consistent(E,q)* predicate which returns false for an entry E if the given value that is searched with a query q can not be found in E's subtree. If it *might be* found, the predicate returns true. Other complex index-invariant methods like a deletion, insertion, or height-balancing the tree are implemented by the framework. For GiST details, please refer to [9].

The GiST idea to hide the complexity of tree management operations in order to speedup the development of new index structures with similar properties but different key and/or query predicates, was adopted for other scenarios than height-balanced search trees. SP-GiST [2] implements a similar approach for space-partitioning trees like octrees or k-d trees. For executing tree methods efficiently on modern hardware platforms, CC-GiST [11] extends GiST with pointer and key compression techniques to become cache conscious.

2.2 Index Operations on the GPU

Several approaches exist for processing index structures on special hardware like GPUs (which we also refer to as "device" in the following). Indexes are either embedded in other GPU algorithms like joins [8] where the index scan results are directly used without any data transfer back to the CPU (which we also refer to as "host"). Another approach is to use a coprocessor accelerated index as a separate building block which offloads queries to the device and transfers results back to the host. [19] implements such an approach for R-trees. It requires that the entire tree data structure - an array containing child node IDs for each tree node and another one for the coordinates of corresponding minimal bounding rectangles (MBRs) - is stored on the device's main memory. Scans are executed iteratively for each tree layer, starting at the root. The query predicate is evaluated in parallel for all child nodes of the current layer and matching children are marked in a boolean result array. After each iteration these results are logically ANDed with the array of the previous iteration. After processing the leaf layer, the result array contains the final query results, i.e., all matching leaf nodes.

To reduce response time of a single query, [16] implements speculative query processing for prefix trees indexing string data. For one query, nodes of deeper layers than the one currently scanned are processed in parallel on separate cores. The intent is to reduce the number of iterations required for processinging a single query. Results of these deep path scans are merged after each iteration to filter false positive results. A speedup is achieved if the computational overhead for scanning unnecessary nodes can be compensated with saving additional iterations.

To maximize performance of both, single query response time as well as throughput in batch query processing, the Fast Architecture Sensitive Tree (FAST) [10] implements a blocking strategy in an in-memory binary search tree. Nodes of (sub)trees are grouped together and stored contiguously in memory to maximize locality of data access and therefore cache efficiency. Special hardware functions like SIMD capabilities are considered as well. For this special tree structure, scan and bulk load operations were parallelized and implemented for a CPU and GPU system.

2.3 Contributions of this Paper

When analyzing the approaches in sections 2.1 and 2.2, several challenges can be identified. Existing GiST frameworks support the development of new index structures. But they lack the capability to efficiently utilize modern hardware. Furthermore, only single queries are processed which is not optimal for throughput-oriented use cases as mentioned in section 1. Existing index implementations for GPUs require that the entire index data is stored in device memory. Hence, transfer times which can be critical for algorithms using coprocessors [5,12] were ignored in performance evaluations. For large-scale applications this assumption is unrealistic. Some implementations do not efficiently utilize the hardware, e.g., [19] uses only one CUDA block (cf. section 3.5.1) for scanning and therefore utilizes only one of the available cores. Furthermore, it is not computationally efficient. All index nodes must be scanned in order to process one query. The filtering idea of an index tree is therefore ignored.

The intent of our approach is to extend the GiST framework to hide the complexity of implementing and tuning general index tree algorithms for special hardware like multicore CPUs or GPUs. Since at the current state of development only a prototype was implemented, our approach is subject to some restrictions:

- Unlike FAST [10] which implements scan and creation operations, we focus on scans only.
- Generally, the presented algorithms could be specialized for arbitrary parallel coprocessors. Currently we provide a sample implementation for a GPU.

3. EXTENDING GIST WITH A HARDWARE ABSTRACTION LAYER

3.1 Requirements

Several aspects have to be considered when designing the new extension for the existing GiST framework:

Fine-grained parallelism: To maximize the benefit achievable through parallel execution of a large set of index scan operations, the tasks have to be partitioned into independent subtasks that can be efficiently executed on the available processing units.

Exploiting hardware capabilities: Since different types of processing hardware have special characteristics like caches, SIMD capabilities, support of asynchronous operations etc., the operations have to be tailored to each specific platform for maximizing the achievable performance. Especially coprocessors like GPUs require the explicit transfer of the data to the device which introduces an overhead that has to be compensated somehow. Otherwise, a possible speedup in comparison to algorithms carefully tuned for CPUs as processors can easily be nullified [5, 12] - or even worse a slowdown is experienced.

Out-of-core implementation: To be scalable in terms of size of indexed data, the new index framework must not rely on the entire data to be stored in the available memory of the respective execution hardware. Usually the available main memory of coprocessors is much smaller than the available host main memory. Existing GPU approaches (cf. section 2.2) assume that all data is stored or generated in the

GPU's main memory and therefore transfer times can be neglected. Since coprocessors are shared resources for multiple tasks, we consider this assumption as unrealistic.

Work efficiency: For scalability in terms of computations per input data, each additional input element must only add a (nearly) constant overhead. Otherwise, a linearly increasing workload can not be handled with adding a linear amount of (co)processors, resulting in increasing hardware costs to handle it.

3.2 Scan Processing Model

Our GiST extension was developed to maximize index query throughput. Therefore, a synchronous processing model [18] was implemented which is illustrated in figure 1. The application requests the scan of a batch of index queries that shall be processed simultaneously. After scanning the tree, all matching leaf nodes per query are returned. Generally, each query might have multiple result leaf nodes. Therefore, a set of iterators (one per query) is returned. Depending on the application's use case, they can be processed in the same order like in the input query batch - or any order, i.e., the next available result. Depending on the internal scheduling not all queries are processed equally fast, hence the former access operation may block even if results of following queries are already available while the latter isn't.

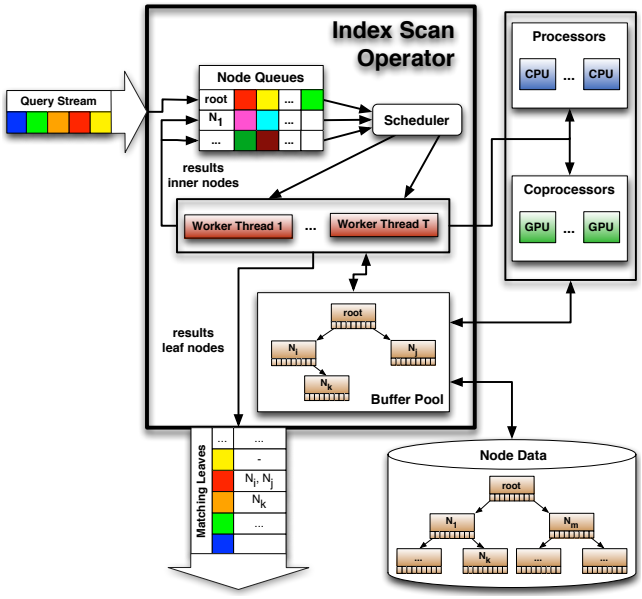


Figure 1: Scan Operator Overview

Query batches are managed in queues per tree node. New batches start at the root. Nodes are scanned iteratively like in [19], i.e., all queries in the batch of a node are processed in a single step. The scan results - matching (query, child node)-pairs - are either added to

a) a queue for the child node in case the scanned node was an inner node and matching children have to be processed in the next iteration

b) the result list of the query if the scanned node was a leaf.

The decision which processing unit of the available hardware executes a node scan is implemented in a scheduler component and can be adapted for each supported hardware platform, depending on its characteristics. Node data is transferred to the (co)processors and, if necessary, can be stored on disks. To avoid data transfers, caching strategies are implemented. The iterations are synchronized by the framework with a single host thread.

3.3 Scan Parallelization

To achieve maximal benefit from massively parallel hardware, a fine-grained parallelization for the scan operation has to be found. The processing scheme is illustrated in figure 2. In our model, a set of queries is executed simultaneously for a number of tree nodes which are independent from each other. For each query, all child node entries (referred to as slots in the following) have to be tested with the GiST key predicate (cf. section 2.1). For the slot keys, two cases exist:

1. **No strict order is defined between the keys:** The data ranges represented by the corresponding subtrees may overlap, i.e., even for point queries multiple child nodes may match. In this case, all child nodes can potentially belong to the result set. Therefore, the key predicate test is required for each of them. An example for this property is the R-Tree where bounding hypercuboids are stored as keys which may intersect within a node.

2. **Keys are strictly ordered:** The represented data ranges do not overlap. Due to this, a binary search can be used to find matching child nodes for a single point query (or multiple binary searches for range queries). The most common example for this is the B-Tree.

In our implementation we focus on the first case since it is the most general one. The second case can be regarded as optimization to speed up the scan within a node.

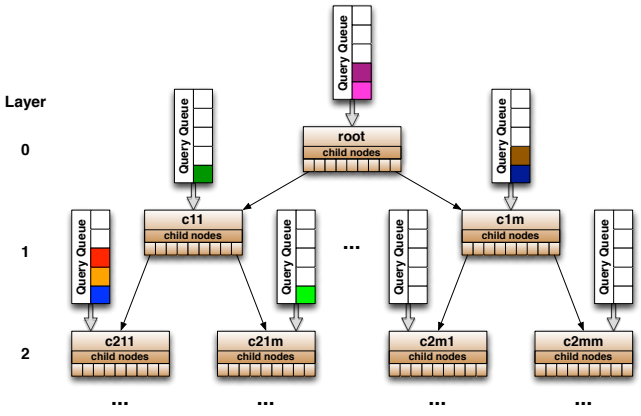


Figure 2: Index Tree Scan

3.3.1 *Intra-Node Parallelization*

Within a node the tests of all (query, slot)-pairs are independent from each other. Therefore, they can be executed in parallel with SIMD operations similar to [10]. We obtain an $(n \times m)$-test matrix for a query batch of size n that shall be scanned on a node with m slots. Let x denote the SIMD capability of the execution hardware (usually $x << n,\ x << m$). For inner nodes one slot is scanned for x queries (matrix column) and for leaf nodes one query

is scanned for x slots (matrix row). Using this scheme allows sequential writes of the respective result lists (cf. section 3.2) which is, due to hardware-dependent memory transfer transactions, usually faster than scattered write operations.

3.3.2 Inter-Node Parallelization

Since all nodes to be processed in an iteration are independent from each other, their scans can be parallelized too. All nodes are scheduled to the available (co)processors and therefore the approach scales linearly with the number of cores. In contrast to existing GPU approaches where a query (batch) has to be finished before the next one can be processed, this independent node scheduling allows the implementation of a pipeline, i.e., within each iteration new queries might become ready to be scanned by the index, which are considered in future iterations. They could be scheduled to the root node by the application or become ready if they were waiting in the previous iteration for a data transfer to complete after a cache miss.

3.4 Parameter Determination

The main index parameters that are relevant for the node scan are the number of slots s per node and the number of queries q within a node batch. To be cache efficient, both, the node and the batch have to fit into the (co)processor cache. In this case, all matrix tests can be executed without cache miss penalties. Therefore, the hardware-dependent cache size as well as the SIMD capability x (cf. section 3.3.1) determine efficient values for s and q. If heterogenous hardware is used - e.g., with a different cache size - the other parameter can be adjusted accordingly and considered during the scheduling.

When indexing very large datasets, it may happen that the index does not completely fit into the available main memory. In this case, like in any disk-based DBMS, nodes can be stored on pages which are fetched from disk if necessary. Usually only a small, cacheable part of all nodes is required for processing the application's lookup operations. For processing multiple query batches one after another, the upper tree layers are required very often. Furthermore, locality can be exploited for multiple search operations. For example when rendering a camera movement within a 3D-scene, the same (or neighboring) spatial areas need to be scanned again in consecutive frames [4].

3.5 GPU Implementation

As use case for a special co-processing hardware we implemented our framework for a GPU besides the straightforward CPU implementation. We decided to use the CUDA programming model [15] to abstract from the specific underlying GPU. Other programming models like OpenCL [14] were also possible, but we chose CUDA since we used an Nvidia GPU for our experiments.

3.5.1 Mapping to CUDA Processing Model

In CUDA, a parallel algorithm (kernel) of independent subtasks (blocks) is processed by a GPU consisting of multiple independent cores on separate dies, called streaming multiprocessors (SMs). Each SM can, depending on memory requirements and hardware capabilities, execute one or more blocks consisting of multiple threads. Within a SM there are w thread processors which work in lockstep mode, i.e., all of them execute the same instruction on different parts of input data - or idle in case code branches differ. For current GPUs, those thread groups (warps) usually have sizes of 16 or 32. The threads of a block can be explicitly synchronized to work on shared data. In our framework, we consider (node, query batch)-pairs to be scanned as separate tasks and therefore map them to CUDA blocks. The elements of the key predicate test matrix are mapped to CUDA threads.

Besides this processing hierarchy, CUDA defines a memory hierarchy which is depicted in figure 3. The global main memory (VRAM) on the GPU device (up to several GB in size) can be directly accessed by the host system. Data that shall be processed has to be transferred explicitly via the connecting PCIe bus. In our environment we achieve transfer rates of ≈5.5 GB/s up and down stream. For access via the SMs, the VRAM achieves transfer rates of ≈75 GB/s. To reduce transfers, we use the VRAM to cache node and query data between different iterations. Furthermore, it is used to store result data which can be preallocated since the maximum size of the matrices is known in advance. On a SM's die, there is a small (some KB in size) shared memory buffer (SHM) which achieves access rates ≈2 orders of magnitude higher than the VRAM. We use it to cache node and query data for a single scan.

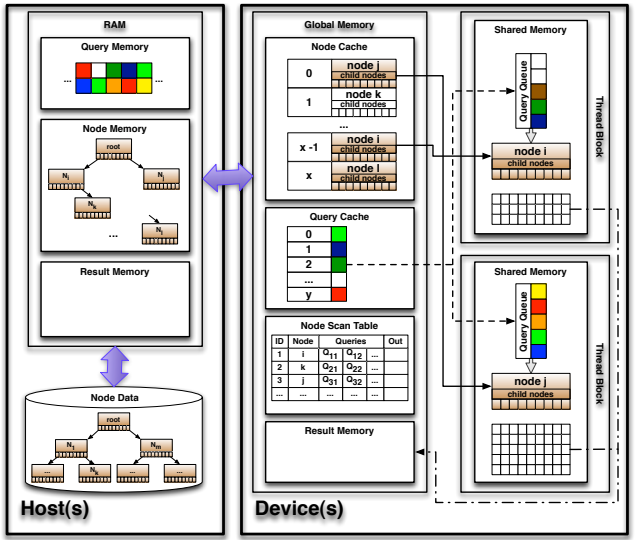


Figure 3: Index Scan - CUDA Implementation

3.5.2 Scan Preparation

To determine the parameters for each block, a preparation kernel is required. It fills a Node Scan Table (cf. figure 3) with the starting positions of each node data and query batch. Therefore, all cached node and query entries are scanned in parallel for each task and a pointer to the starting address is inserted into the table which is later used to load the data into the SHM cache. Additionally, the position of the results in the output data buffer is computed.

3.5.3 Scan Execution

After the scan has been prepared, the actual matrix test kernel is launched. First, all node slots and queries to be scanned are loaded in parallel into the SHM of the active SM. To assure that all required data has been stored, all threads are synchronized after this step. Second, the scan

is executed. For each (query, slot)-pair the index-dependent key predicate is evaluated as explained in section 3.3.1. The predicate has to be provided by the index developer as well as the actual key types. Note, that for storing the key data no special method needs to be provided by the developer since the framework loads raw byte arrays of fixed size for all nodes. Predicate results are directly written into the global result memory buffer since the size of the matrix grows quadratically in the order of the number of slots and queries.

3.5.4 Result Creation

Since - depending on a key predicate's selectivity - the rows or columns of the result matrix are usually sparsely populated, we implemented an optional result creation kernel as final phase to compress them with removing gaps between the (queryID, slotID)-entries. Therefore we store the rows/columns one after another in the SHM (which always fit) and execute a parallel prefix scan/stream compaction primitive [7] on the array which removes all NULL entries.

4. EVALUATION

We used our framework to implement a 3-dimensional R-tree with 64-bit values as coordinates and conducted some experiments on our test server having two Intel Xeon X5690 6-core CPUs and two Nvidia Tesla C2050 GPUs which were connected to the host system via PCIe 2.0 16x bus.

Experiments were executed to analyze the impact of the index parameters on the CPU and GPU scan algorithms and profiled the efficiency of our GPU implementation. Parameters that are relevant for our framework are:

- **slots:** the number of child entries per node which impacts the total node size
- **queries per task:** the number of queries that are scanned per node
- **tasks:** the number of nodes scanned in parallel in each iteration, impacting the utilization of available cores
- **selectivity:** the number of matching child nodes according to the range of a query. All nodes were artificially generated with disjoint rectangles of equal area as child entires. Due to this, we were able to generate queries with certain selectivities to simulate different result sizes.

4.1 Impact of Tree Parameters

The first question at hand is if executing a node scan on the GPU is beneficial, i.e., if it is faster (including the overhead) than the same scan on the CPU. We compared total CPU and GPU scan execution times with the varying tree parameters *slots* and *queries per slot* in steps of 32 (warp size as SIMD unit on the GPU cf. section 3.3.1). To reduce the number of measurements, we used a constant number of $tasks = 128$ which is much larger than the number of cores on the GPU to keep it fully utilized. Queries were generated with $selectivity = 0.25$ to include result processing times. As a measure, we define:

$$speedup = \frac{CPU\ time}{GPU\ time} \quad (1)$$

for the execution on the device vs. one thread on the host. The results are illustrated in figure 4. The GPU algorithm

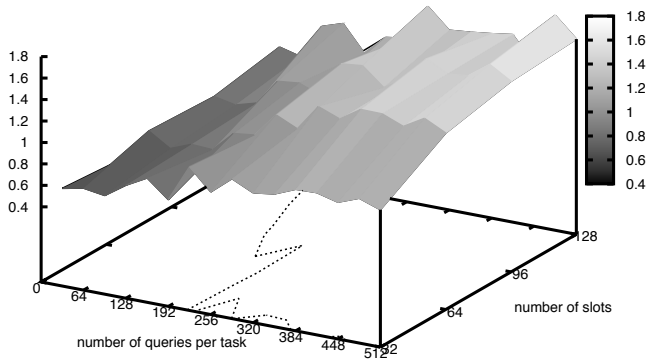


Figure 4: GPU Speedup

is up to 1.8 times faster than its CPU counterpart for scanning large query batches (448-512) on nodes with 96-128 slots. But there are several parameter combinations where the GPU algorithm is up to 2.5 times slower. This slowdown is experienced when scanning short query lists on small nodes since the GPU can not exploit its full potential through highly parallel processing for such small inputs.

So, how do we chose the parameters? Finding an optimal value for the number of slots is quite difficult. In contrast to index implementations where the size of disk pages determines the sizes of index nodes, it now becomes important to know where a node shall be processed - which, besides the installed hardware, depends on the workload. A fixed node size seems not to be useful. For high-loaded nodes like the first tree layers where most queries pass through, a high fanout would be beneficial since the GPU provides best performance here. For low-loaded nodes like those in layers near the leaves, a CPU scan execution on smaller nodes would be faster since query batches are expected to be shorter there.

To expect CPU and GPU runtimes for different parameter values (especially for different query batch sizes, since node sizes won't change at runtime) is important for the framework to perform scheduling decisions. Wrong expectations could lead to a slowdown. Therefore, the model of figure 4 has to be implemented in the scheduler component. The dotted line on the x-y projection marks the break-even parameters where both implementations have approximately the same runtime. All scans on the left side shall be executed on the host while the others are faster with co-processing on the device.

Implementing such a decision model that is general enough for use in a framework is non-trivial. It depends on the actual (co)processors that are used, and the actual index implementations provided by the developer. The keys and predicates for the currently implemented R-tree are relatively simple. A predicate evaluation translates just to 2n coordinate comparison operations where n denotes the dimensionality of the tree. For more complex predicates like nearest neighbor searches a GPU implementation may be more beneficial due to the increased computational complexity. To solve this scheduling problem we suggest executing some calibration scans for estimating the model. We are currently working on a self-learning scheduler that approximates expected runtimes with polynomials and adjusts its parameters according to actually measured execution times. But further details on that are out of the scope of this paper.

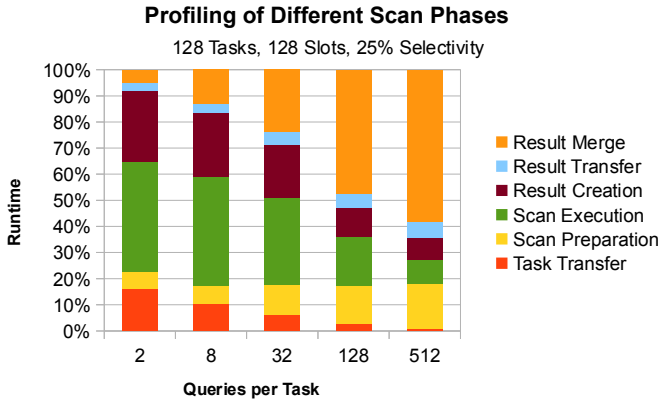


Figure 5: Scan Phases - Percental Runtimes

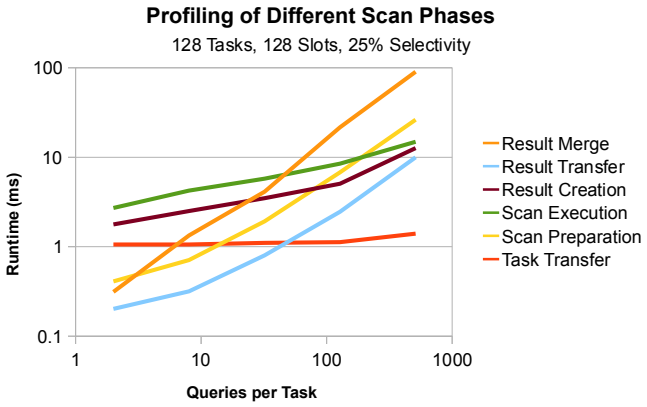


Figure 6: Scan Phases - Runtimes

One may discuss if comparing runtimes of an entire GPU to just a single host thread is valid. In our model, that is exactly the decision the scheduler has to do. Each processing unit that can be separately controlled is managed by a worker thread (cf. figure 1). For host processors, each CPU core (12 in our scenario) can execute an independent thread, or even more when hyperthreading is enabled. GPU cores on the device are not working independently but are executing the same kernel. From the scheduler point of view, it has to be decided which subsets of all nodes in the current iteration have to be distributed to which worker thread to achieve a maximum overall throughput. This can be achieved with the hybrid system approach where many independent CPU cores can scan many low-loaded nodes while attached GPU devices process a smaller number of high-loaded nodes in the meantime. Moreover, for processing the final results after the queries we streamed through the index, additional cores will be required.

4.2 Scan Profiling

Finally, we analyze the efficiency of our GPU framework implementation. We omit the host counterpart here since its algorithm is quite simple and we don't have explicit control on the usage of caches as they are implemented by the hardware. During the first predicate evaluation in a test matrix row / column, the data elements are stored into the executing CPU's L2 cache (which is larger than the GPU's SHM) and are reused for all further evaluations.

To analyze the GPU algorithm, where the entire caching strategy is managed by the framework, we profiled the different scan phases (cf. section 3) with varying queries per task because this parameter impacts the complexity of all scan phases. The number of tasks was fixed at 128 for guaranteeing a full GPU utilization. A selectivity of 0.25 was chosen to include the result processing phases on the host and the device. As shown in figures 5 and 6 the impact of input transfers decreases with increasing queries per tasks since the computational complexity grows quadratically with respect to the input size. For result transfers, as expected, the impact increases since the result size is in the same order as the computation (matrix). What we did not expect is a large overhead (>50%) for merging GPU results with the batches maintained on the host which is the main bottleneck of our prototype. This overhead grows dramatically with the size of the result sets depending on the selectivity (not shown here). Detailed profiling revealed that maintaining the shared data structures used for storing query batches requires expensive synchronizations on the host side. This can be avoided by merging results locally and executing a small synchronizing phase afterwards. Large performance improvements are expected here. Further tuning can be done with utilizing Nvidia hardware characteristics like coalesced reads, avoiding bank conflicts, or overlapping kernel execution with copy operations [15].

5. CONCLUSIONS & OUTLOOK

Recent trends in hardware development such as multi-core CPUs and GPUs provide great opportunities for improving the performance of index operations by fine-grained parallelization. However, particularly co-processing units like GPUs require a signifiant effort for an efficient implementation of the index structures.

In this paper, we have introduced our approach of bringing the GiST framework to these platforms. In contrast to other works, which rely on the assumption of keeping the entire index structure in the GPU memory, our approach implements a hybrid out-of-core strategy hiding the complexity of efficient data transfer between host and GPU memory. Furthermore, we focus on a multi-query processing model to mitigate the transfer time.

The experimental evaluation of an R-tree implementation shows that for large tree nodes and query batch sizes that can be processed in parallel, a scan on the GPU can be nearly twice as fast as the same scan on the host. Those speedups are expected to be larger after further tuning the prototype. Nevertheless, careful scheduling is required because, for small nodes and/or a small number of queries, a GPU-based processing may otherwise lead to a slowdown of 2.5. For future work we therefore plan to:

1. implement a learning scheduler component for automating this task
2. support index load and update operations
3. investigate further index implementations to show the benefit of a GiST-based framework to speed up the development
4. integrate support for other coprocessors and models

6. REFERENCES

[1] A. Ailamaki. Managing scientific data: lessons, challenges, and opportunities. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1045–1046, New York, NY, USA, 2011. ACM.

[2] W. G. Aref and I. F. Ilyas. Sp-gist: An extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17:215–240, 2001. 10.1023/A:1012809914301.

[3] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[4] W. T. Correa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society.

[5] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134 –144, april 2011.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[9] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[10] 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 international conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[11] W.-S. Kim, W.-K. Loh, and W.-S. Han. Cc-gist: Cache conscious-generalized search tree for supporting various fast intelligent applications. In S. Mehrotra, D. Zeng, H. Chen, B. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics*, volume 3975 of *Lecture Notes in Computer Science*, pages 657–658. Springer Berlin / Heidelberg, 2006.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[13] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 289–298, New York, NY, USA, 1988. ACM.

[14] A. Munshi. The OpenCL Specification, 2011.

[15] C. Nvidia. Nvidia cuda. *Changes*, 7(6(36)):187, 2011.

[16] P. B. Volk, D. Habich, and W. Lehner. Gpu-based speculative query processing for database operations. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[17] T. Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[18] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[19] X. Xiao, T. Shi, P. Vaidya, and J. J. Lee. R-tree: A hardware implementation. In *CDES'08*, pages 3–9, 2008.