Seventh International Workshop on Data Management on New Hardware (DaMoN 2011)

June 13, 2011

Athens, Greece

In conjunction with ACM SIGMOD/PODS Conference

Stavros Harizopoulos and Qiong Luo
(Editors)

Industrial Sponsor

FOREWARD

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. CPUs with superscalar out-of-order execution, simultaneous multi-threading, multi-level memory hierarchies, and future storage hardware (such as flash drives) 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

Stavros Harizopoulos, HP Labs (stavros@hp.com)
Qiong Luo (Hong Kong University of Science and Technology, luo@cse.ust.hk)

Program Committee

Peter Boncz	(CWI Amsterdam)
Shimin Chen	(Intel Research)
Goetz Graefe	(HP Labs)
Ryan Johnson	(University of Toronto)
Christian Lang	(Acelot Inc.)
Kenneth A. Ross	(Columbia University)
Jens Teubner	(ETH Zurich)
Dimitris Tsirogiannis	(Microsoft Research)

TABLE OF CONTENTS

Scalable Aggregation on Multicore Processors

Yang Ye, Kenneth A. Ross*, Norases Vesdapunt
Department of Computer Science, Columbia University, New York NY
(yeyang,kar)@cs.columbia.edu, nv2157@columbia.edu

ABSTRACT

In data-intensive and multi-threaded programming, the performance bottleneck has shifted from I/O bandwidth to main memory bandwidth. The availability, size, and other properties of on-chip cache strongly influence performance. A key question is whether to allow different threads to work independently, or whether to coordinate the shared workload among the threads. The independent approach avoids synchronization overhead, but requires resources proportional to the number of threads and thus is not scalable. On the other hand, the shared method suffers from coordination overhead and potential contention.

In this paper, we aim to provide a solution to performing in-memory parallel aggregation on the Intel Nehalem architecture. We consider several previously proposed techniques that were evaluated on other architectures, including a hybrid independent/shared method and a method that clones data items automatically when contention is detected. We also propose two algorithms: partition-and-aggregate and PLAT. The PLAT and hybrid methods perform best overall, utilizing the computational power of multiple threads without needing memory proportional to the number of threads, and avoiding much of the coordination overhead and contention apparent in the shared table method.

1. INTRODUCTION

The number of transistors in microprocessors continues to increase exponentially. Power considerations mean that chip designers cannot increase clock frequencies. Instead, chip designers have shifted the design paradigm to multiple cores in a single processor chip. Application developers thus face the challenge of efficiently utilizing the parallel resources provided by these multi-core processors.

We consider data intensive computations such as aggregation, a central operation in database systems. The essential question is whether such computations should be organized on multi-core processors as disjoint independent computations, or as coordinated shared computations. Independent computations avoid coordination overhead and contention, but require resources proportional to the number of threads. Compared with shared computations, independent computations on n threads have effective use of only $1/n$th of the cache.

Shared computations allow multiple threads to use a common data structure, meaning that larger data sets can be handled with the same amount of RAM or cache. This enhanced scalability comes with the burden of making sure that threads do not interfere with each other, protecting data using locks or atomic instructions. This coordination has overheads, and can lead to contention hot-spots that serialize execution.

1.1 Prior Work

Adaptive parallel aggregation has been investigated in the context of shared-nothing parallelism [13]. On chip multiprocessors, previous papers have proposed a variety of methods for aggregating a large memory-resident data set [7, 4]. These papers focused on the Sun Niagara T1 and T2 architectures because they offered a particularly high degree of parallelism on a single chip, 32 threads for the T1 and 64 threads for the T2. The high degree of parallelism made issues such as contention particularly important. We examine four previous hash-based algorithms. These proposals are:

Independent. Perform an independent hash-based aggregation on disjoint subsets of the input, and combine data from the tables at the end. Each active thread has its own full-sized hash table.

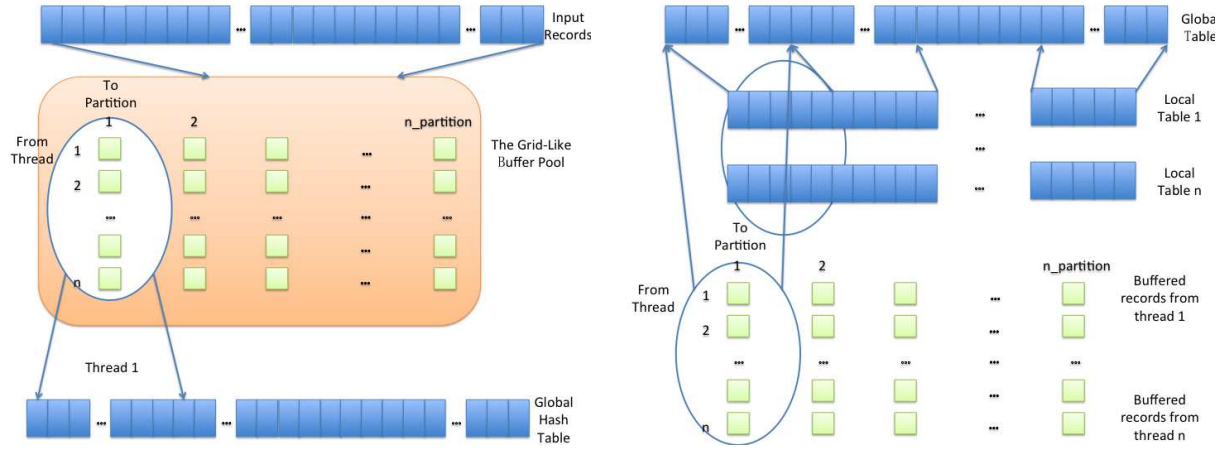
Shared. Use a single common hash table for all threads, and protect access to data elements by using atomic instructions to update hash cell values.

Hybrid. In addition to a global hash table, each thread has a small private local table that is consulted first. If there is a match in the local table, then the update happens there without atomic instructions. A miss in the local table leads to an eviction of some partial-aggregate element from the local table into the global table, and the creation of a new aggregate for the record in the local table.

Contention Detection. The programmer specifies a set of basic operations to handle the initialization and combination of aggregate values. The system automatically detects contention on frequently accessed data

*This work was supported by the National Science Foundation under awards IIS-0915956 and IIS-1049898.

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.



(a) Partition-then-Aggregation Method (b) PLAT

Figure 1: Illustration of Partition-and-Aggregate method and PLAT

items, and clones them so that fewer threads access each copy, reducing contention. Infrequently accessed items are not cloned. A "local" variant of this scheme allows threads to make their own local clones when facing contention, while a "global" variant shares clones across a balanced number of threads.

The contention detection method counts the number of compare-and-swap attempts needed to update an item. Appendix A gives the code for an atomic add instruction that also returns this number of attempts. If this number is greater than a threshold (set to 1 by default), cloning is triggered. Note that the contention detection method allows computations more general than aggregation, and that this generality comes with a performance overhead [4].

1.2 Contributions

In an effort to broaden prior results, we aim in this paper to study aggregation on a more conventional architecture, the Intel Nehalem processor. Compared with the Niagara processors, the Nehalem processor has a higher effective clock speed per thread, and fewer threads (8 per chip, 16 in a dual-chip system).

We ported the code for the methods discussed above from the Niagara to the Nehalem architecture and quantified their performance. Some of the performance results are expected. The independent method performs well, but cannot scale to the largest data sets. The shared method performs poorly for small group-by cardinalities. As on the Niagara machines, contention for a small number of hash cells severely limits performance.

One unexpected result is that the contention detection mechanism failed to ameliorate the contention for small group-by cardinalities. Because there are fewer threads than on the Niagara, and because each thread operates much faster, observable contention (via a failed compare-and-swap) becomes relatively rare. As a result, clones are not generated at a sufficiently rapid rate. We also measure several indirect indicators of contention, including cache coherence overhead and MOMC events.

In light of these observations, we consider two additional aggregation methods that minimize thread sharing and partition work without creating a global hash table for each thread. These methods are illustrated in Figure 1.

Partition-and-Aggregate. First partition the input into fragments based on the group-by value, and aggregate each fragment on a single thread. Different fragments can be processed in parallel.[1] To avoid contention when writing the partition output, the partition buffers are arranged in a grid-like manner.

PLAT. Extending the Partition-and-Aggregate method, each thread is given a private local table in which to perform aggregation. Once this table fills up, input records that miss in the local table are partitioned as before. The local table is particularly useful when there is a small number of frequent group-by values. PLAT is an acronym for "Partition with a Local Aggregation Table."

The best methods overall are the hybrid method and PLAT. Both methods generally match the performance of the independent tables method for small group-by cardinalities, while being able to scale to larger data sets. For some distributions with group-by cardinalities somewhat beyond the L1 cache size, the hybrid method outperforms PLAT, because its local table adapts over time using dynamic eviction. PLAT does not evict data from the local table. On the other hand, PLAT outperforms hybrid when data cannot fit in the L3 cache because the partitioning enhances the locality in the second phase.

Our results show that on the Nehalem architecture, one can achieve the performance of independent computation without duplicating large hash tables across threads. As more threads become available in future generations of machines, the importance of such scalability increases. The use of a cache resident local table is a key element of achieving scalability while retaining high performance.

Our results also show that, despite efforts to hide architectural details, architecture matters. The contention detection method that worked well on the Niagara T2 machine does not work as well on the Nehalem processor because directly

[1] A version of the Partition-and-Aggregate method was previously evaluated on the Niagara architecture [12], but it did not outperform direct aggregation using a large hash table.

Platform	Sun T2	Intel Nehalem Xeon E5620
Operating System	Solaris 10	Ubuntu Linux 2.6.32.25-server
Processors	1	2
Cores/processor (Threads/core)	8 (8)	4 (2)
RAM	32GB	48 GB
L1 Data Cache	8KB per core	32 KB per core
L1 Inst. Cache	16KB per core	32 KB per core
L2 Cache	4MB, 12-way Shared by 8 cores	256 KB per core
L3 Cache	N.A.	12MB, 16-way Shared by 4 cores

Table 1: Experimental Platforms

measurable contention (via a failed compare-and-swap) is rare. We show that a lock-based implementation of the contention detection method does work well on the Nehalem architecture because the lock attempt coincides with the time-consuming cache load, making contention more easily observable.

The remainder of the paper is organized as follow. Section 2 presents an overview of the architecture and its influence on aggregation algorithms. Section 3 experimentally evaluates the algorithms. Section 4 concludes this paper.

2. ARCHITECTURAL OVERVIEW

With large main memories, main memory access has taken the role of I/O bandwidth as the new bottleneck in many data-intensive applications, including database systems [1, 2]. Caches are designed to speed up memory access by retaining frequently used data in smaller but faster memory units. There are multiple cache levels, each successively larger but slower. The higher level caches are typically private to a core, and the lowest level caches are typically shared between cores on a chip. The specific cache configurations of the Sun Niagara T2 and the Intel Nehalem processors are given in Table 1.

On a multi-chip machine, the caches of the various chips are kept consistent via a cache coherence protocol. Within a chip, the private caches also communicate to make sure that they store consistent results for items accessed by multiple threads.

On the Intel Nehalem machine, if one thread is updating a cache line and another thread wants to access the same cache line, there is a potential memory order hazard. To avoid potential out-of-order execution, the machine has to flush the pipeline and restart it. This event is called a "Memory Order Machine Clear" (MOMC) event. When an MOMC event is triggered, it can induce more MOMC events because of the delay caused by the first event. MOMC events can be quite expensive, and can effectively serialize execution. Previous research also discovered the influence of MOMC events on Intel's Pentium 4 SMT processor [3].

The L3 cache on the Nehalem, and the L2 cache on the Niagara T2 are inclusive, meaning that data in the higher level caches also reside in these caches. Thus the lowest-level cache is the natural place to resolve accesses within a chip. Accessing cache memory in another processor in multi-processor system is more complicated. The Nehalem uses the QuickPath Interconnect (QPI) to control the request to and from the memory of another processor. The Nehalem processor implements the MESIF (Modified, Exclusive, Shared, Invalid and Forwarding) protocol [5] to manage cache coherency of caches on the same chip and on other chips via the QPI. Because of the different path lengths to different kinds of memory, the Nehalem exhibits non-uniform memory access (NUMA) time. Table 2 summarizes the memory latencies of the Nehalem 5500 processor; this data is taken from [10]. (Our experimental machine uses 5620-series processors, but the latencies are likely to be similar.) Note that accessing L3-resident data becomes more expensive if the data has been modified by threads on a sibling processor.

Since the unit of transfer between levels of the memory hierarchy is the cache line, it is possible for false sharing to impact performance. False sharing occurs when two threads access disjoint items that happen to reside in the same cache line. Because of the performance pitfalls of modifying shared cache lines, the Intel optimization manuals recommend that sharing of data between threads be minimized [11]. Nevertheless, our goal is to take advantage of shared data to better utilize memory, and so some degree of sharing may be necessary.

3. EXPERIMENTAL EVALUATION

3.1 Experiment Setup

We consider an aggregation workload (count, sum, and sum squared) similar to that in [4, 7]. The query is

```
Q1: Select G, count(*), sum(V), sum(V*V)
    From R
    Group By G
```

where `R` is a two-column table consisting of a 64-bit group-by integer value `G` and a 64-bit integer value `V` for aggregation. The input size is $2^{28} \approx 270$ million records which fits in 4GB of RAM. We measure the throughput of aggregation as our performance metric. We consider a variety of input key distributions and vary the number of distinct group-by keys in each distribution. We use the synthetic data generation code from [7] (based on the work of Gray et al. [8]). The distributions are: (1) uniform, (2) sorted, (3) heavy hitter, (4) repeated runs, (5) Zipf, (6) self-similar, and (7) moving cluster.

3.2 Independent Table Method

The performance of the independent table method is particularly good for small group-by cardinalities: over 1,000 million tuples per second as shown in Figure 2 for 8 concurrent threads. The downward steps in performance occur as the group-by cardinality causes the active part of the hash table to exceed successive cache sizes.

The independent table method does not scale well because for 8 threads it needs 8 times the memory to accommodate hash tables for all threads. For instance, if records with the same key appear in the inputs of all threads, the threads all need to store this key in their individual tables and merge

Access Type	Exclusive			Modified			Shared			RAM
	L1	L2	L3	L1	L2	L3	L1	L2	L3	
Local	4	10	38	4	10	38	4	10	38	195
On Chip	65			83	75	38	38			
Across Chip	186			300			170			300

Table 2: Read Cache Latency for Different Access Types on the Nehalem 5500 Processor

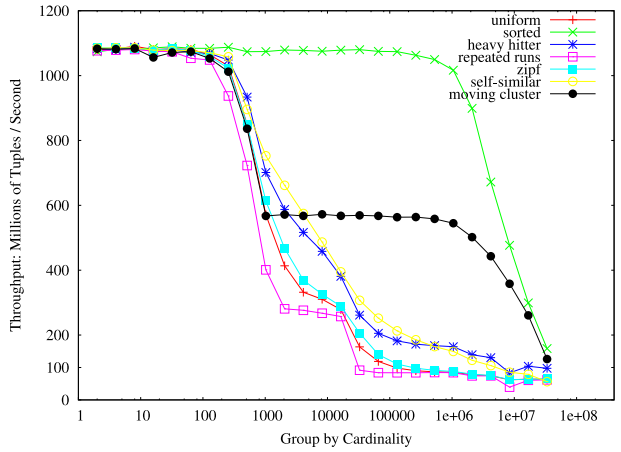


Figure 2: Independent Tables on all Distributions, 8 Threads

the results at the end. In our experiment, when the group-by cardinality is 2^{26}, the independent table method fails because the memory required is 64GB, more than the 48GB available on the machine.

3.3 Contention Detection and Shared Table

Figure 3(a) shows the performance of the contention detection aggregation method using uniform input on the T2 platform with all 64 threads. The No-Detection curve is a simple shared table with no cloning, which performs badly (due to contention) for small group by cardinalities. On the T2, both the local and global cloning methods are effective at eliminating the poor performance at small group-by cardinalities.

Figure 3(b) shows the performance of the contention detection aggregation on the Nehalem platform with 16 threads enabled. (The results were similar for 4 or 8 threads, even when all four threads were mapped to a single chip.) Contrary to our initial expectations, the contention detection method does not achieve similar benefits on the Nehalem platform as on the T2 platform. We kept track of the number of clones created. On the Nehalem processor, the cloning methods create relatively few clones, often just 2 or 3 even for very small group-by cardinalities with high levels of contention.

To understand why cloning is rarely triggered, we created a micro-benchmark in which a single shared variable is updated using an atomic compare-and-swap operation by all threads in parallel on both the T2 and Nehalem architectures. On the T2, the average number of failed compare-and-swaps was about 53, which seems reasonable given that there are 64 threads. On the Nehalem, however, this number was 0.1. This sharp difference is due to the smaller number of much faster threads on the Nehalem platform. (We measured the single threaded performance of the Nehalem to be seven times the single-threaded T2 speed for a simple scalar aggregation.)

Figure 12 in Appendix B shows the performance counters for the number of MOMC events per record, and the number of snoops hitting another cache in modified states (HITM events) per record as measured using the Vtune performance analysis tool [9]. The global method has higher event numbers than the local method.

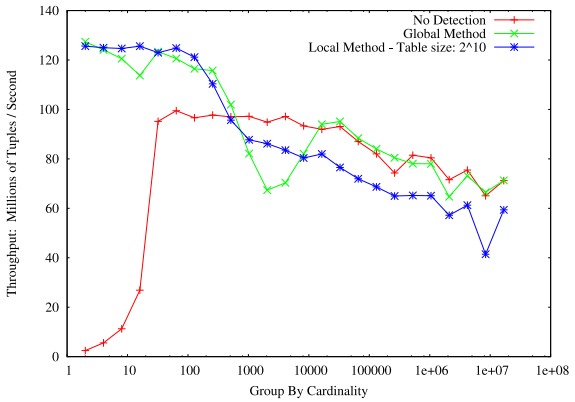
To understand whether clone triggering issues are the only performance issues, we re-ran the contention detection methods for small group-by cardinalities with the contention threshold set to zero, so that clones are always created. The performance of these methods is also shown in Figure 3(b). While the local method performance mirrored that of the T2, the global method performs even worse than without always cloning. After analyzing the code, we realized that the global method allocates clones in contiguous memory. Since the aggregate state is 24 bytes, smaller than a cache line, false sharing was occurring between clones. When we padded the aggregate state to be exactly 64 cache-line aligned bytes, the global performance improved, as shown in Figure 3(b).

3.3.1 *Locking vs. Atomic Operations*

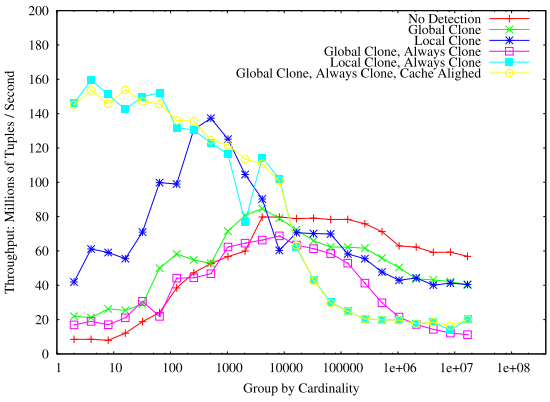
Cieslewicz et al. [4] suggested (but did not evaluate) an alternative contention detection framework using locks rather than atomic operations. A single mutex can protect all aggregate operations for a given group. A failed attempt to obtain a lock would indicate contention, analogously to a failed compare-and-swap. We implemented this alternative version of contention detection by using the `try_lock` system call that returns control to the calling context if a lock attempt is unsuccessful. Each failed `try_lock` adds to the contention count.

On the Niagara architecture, locks are more expensive than atomic operations; it takes six or more atomic operations before the lock-based implementation performs better [7]. This observation remains true in our contention detection method: On the T2, the atomic operation based method performs better (data not shown).

Figure 4 shows that for the Nehalem machine, the lock-based methods were able to detect contention and perform well even in cases where the original implementation failed to detect contention. The critical difference between the two methods is that the `try_lock` method induces a time-consuming coherency-related cache-miss to load the cache line containing the mutex and aggregate values. This delay makes lock contention much more likely to be observed. In contrast, the atomic operations decouple the initial cache miss of the aggregate values from the compare-and-swap, leading to the absence of observable contention as discussed above.



(a) T2, 64 threads



(b) Nehalem, 16 threads

Figure 3: Performance of Contention Detection on the T2 and Nehalem Processors, Uniform Distribution

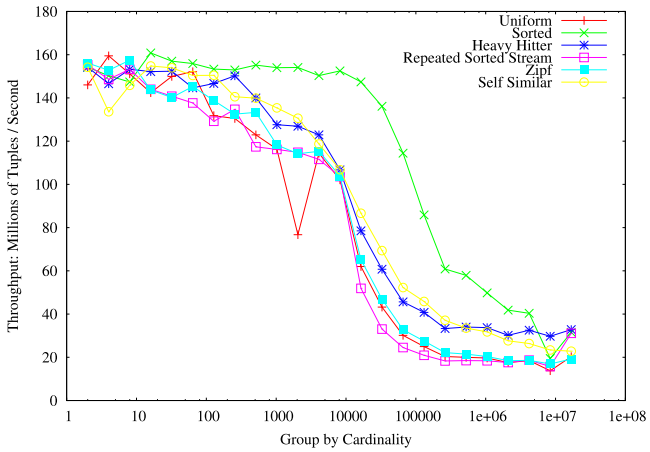


Figure 4: Contention Detection Method using `try_lock`

3.4 Partition-and-Aggregate

We first vary the fanout from 2^3 to 2^9-way partitioning on all distributions to empirically determine the optimal fanout. As shown in Figure 6, the performance difference is noticeable when the group-by cardinality is 2^{20} or larger: the throughput difference is up to 50 million records per second. Fewer partitions suffer from large partition size that needs more hash buckets for each thread and thus yields worse performance due to cache misses.

In our implementation, the size of the pointer is 8 bytes and the size of the buffer head each pointer points to is 64 bytes. Therefore, when the fanout is 256, the size of all buffer headers together is 16 KB (fanout times the header size). The pointers also need space and thus to fit into the L1 cache (32KB), 256-way partitioning is the largest fanout. Furthermore, the Nehalem architecture in our experimental setup has 2 levels of translation look-aside buffers for each core [6]. The L1 DTLB has 64 entries and the L2 TLB (Data and Instruction together) has 512 entries and thus 256-way partitioning does not thrash the TLB. But 512-way (2^9) partitioning suffers from both L1 cache misses and TLB misses. Beyond 512-way partitioning, larger partitioning fanout will lead to further performance degradation.

Therefore, we use 256-way partitioning, which is the optimal fanout for our experimental setup: the partitioning buffers fit into the L1 cache and the group-by cardinality in each partition is reduced to enhance data locality. The performance of the Partition-and-Aggregate method is stable at 120-140 million tuples per second for all distributions, as is shown in Figure 11 in Appendix B.

Figure 8 shows that the partition-and-aggregate method outperforms the independent table method when the group-by cardinality is larger than 2^{15}, which corresponds to the L3 cache limit in the independent table method.

3.5 The PLAT Method

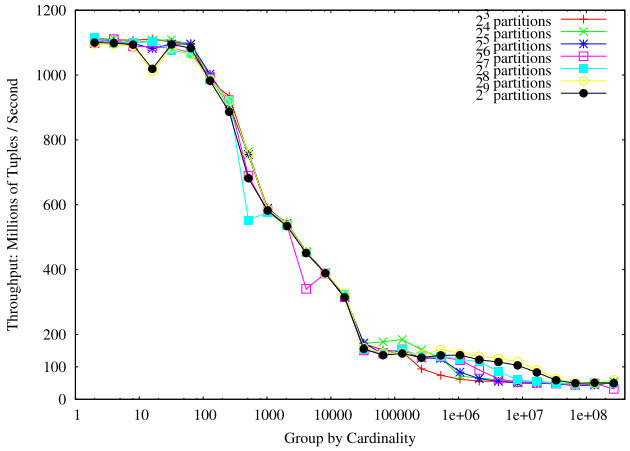
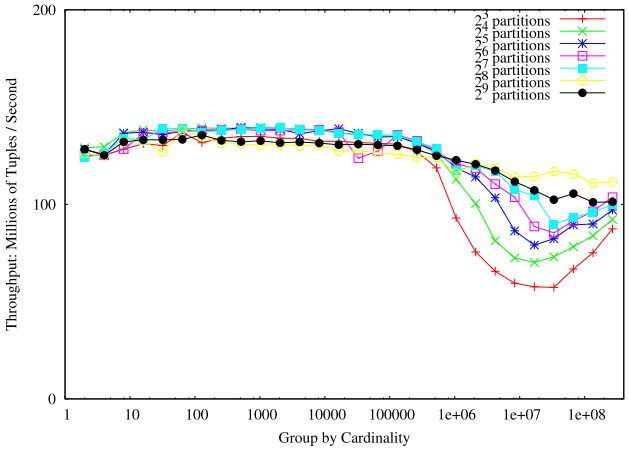
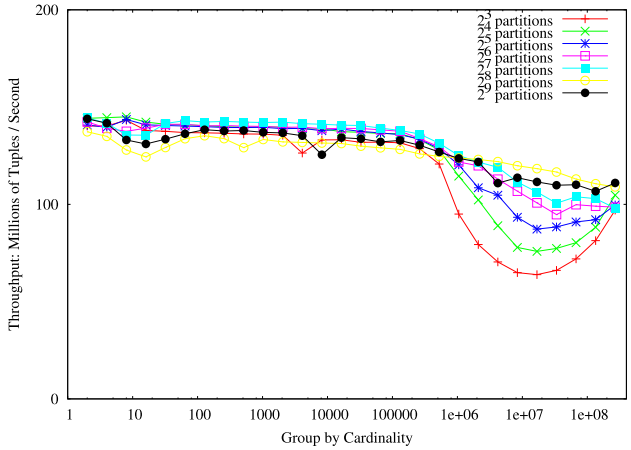


Figure 5: Varying the Fanout on PLAT, 8 Threads, Uniform Distribution

We run PLAT on different number of partitions (fanout) from 2^3 to 2^9. As shown in Figure 5, the results agree with the results from the partition-and-aggregate algorithm that the best fanout is 256. The performance difference is noticeable when the group-by cardinality is around 2^{20} or larger: the throughput difference is up to about 80 million records per second. When the group-by cardinality is very large (larger than 2^{24}) there is no discernible difference in performance with different fanouts. 512-way partitioning



(a) Uniform



(b) Zipf

Figure 6: Varying the Fanout on Partition-and-Aggregate Method, 8 Threads

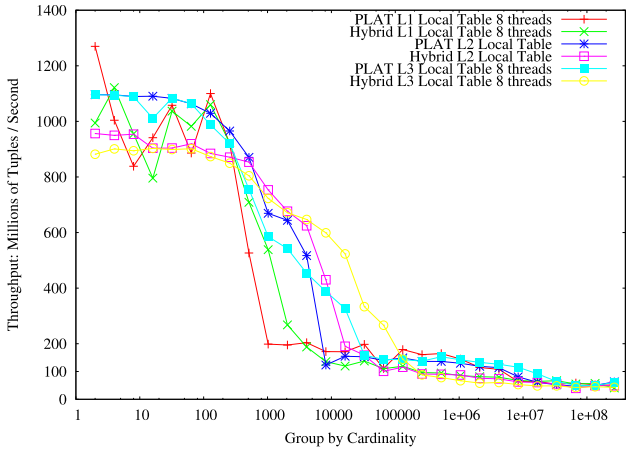


Figure 7: Impact of Local Table Size on PLAT and Hybrid Methods, Uniform Distribution, 8 Threads

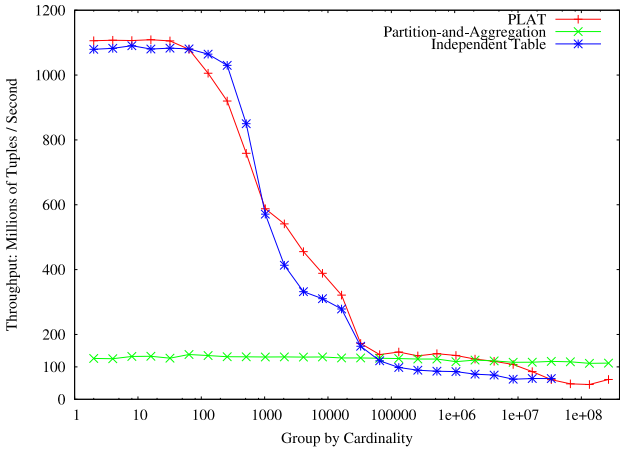


Figure 8: Comparison of Independent Table, Partition-and-Aggregate, and PLAT methods on Uniform Distribution, 8 Threads

performs slightly worse than 256-way partitioning.

The size of the local table in PLAT was set at 2^{15} to ensure L3-cache residence. Figure 7 shows that a smaller (L1-resident or L2-resident) local table performs slightly better for small group-by cardinality, but worse for cardinalities between 8000 and 30000.

Figure 10 shows the performance of the PLAT aggregation method. Figure 8 shows that, like partition-and-aggregate, PLAT outperforms the independent method on large group-by cardinalities by roughly a factor of 2, and scales to larger group-by cardinalities due to lower memory consumption.

One potential optimization we considered was turning off the local table if the hit rate to the local table is sufficiently low. However, even when the hit rate is very low the local table processing overhead was sufficiently small that no performance gain was apparent.

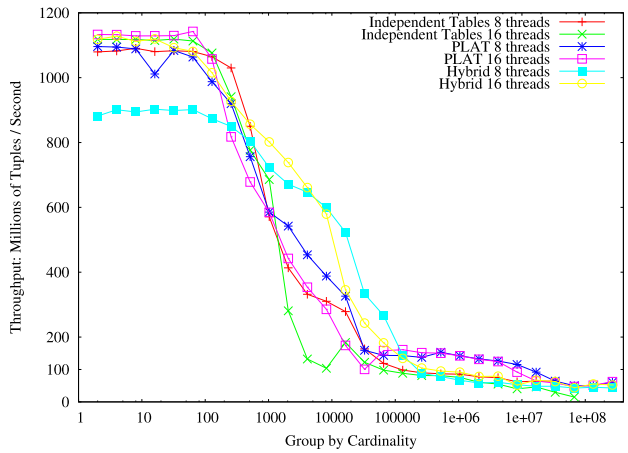
When the data is skewed, PLAT is able to aggregate the most frequent items in the local table in the first phase. Without such pruning, the partition-and-aggregate method will send all records with a frequent common key to one thread; this thread will become the performance bottleneck. As shown in Figure 11 and Figure 10, the partition-and-aggregate method performs worst on the heavy-hitter distribution whereas PLAT performs relatively well.

3.6 Hybrid Method

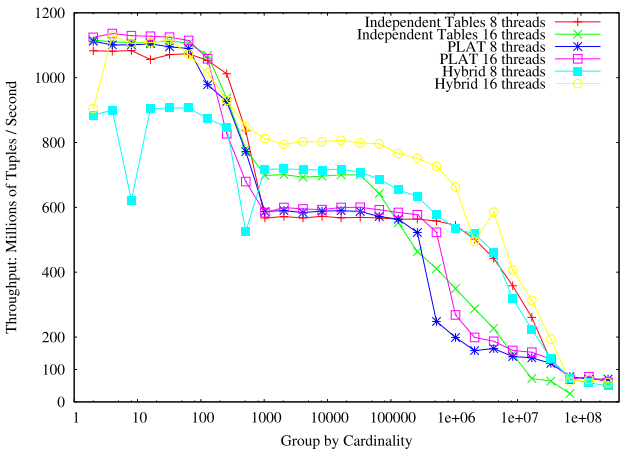
The Hybrid method [7] reduces sharing by making use of a small local table for each thread. The table also adapts to the input because it spills the oldest entry to the global table when the bucket is full. Although it does not eliminate sharing completely, it effectively reduces sharing so that it has minimal impact on performance.

Figure 7 shows the impact of local table size on the performance of the hybrid method using the uniform distribution. An L1-resident local table can accommodate 2^9 entries, and an L3-resident local table can accommodate 2^{15} entries. The trade-off is similar to that of PLAT, and we choose to size the local table based on the L3-cache size.

We show the performance comparison of the hybrid method, independent method and PLAT in Figure 9 for uniform and moving-cluster distributions. (Comparisons for other distri-



(a) Uniform



(b) Moving Cluster

Figure 9: Comparing Hybrid with PLAT and Independent methods

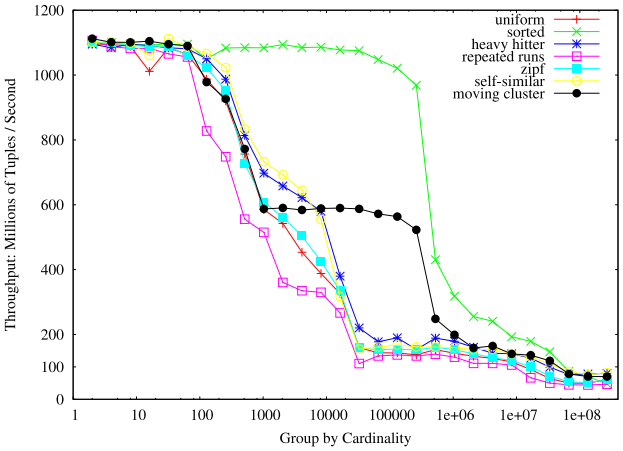


Figure 10: PLAT Method on all Distributions, 8 Threads

butions are shown in Figure 13 in Appendix B.)

The hybrid method does well at intermediate cardinalities relative to PLAT for two reasons. First, PLAT needs to store both a local table and partitioning data in cache, whereas the Hybrid method can use all of the cache just for the local table. Second, as is apparent for distributions with changing locality such as the moving cluster distribution, the local table in the hybrid method is able to adapt to the changing reference pattern. PLAT is much more sensitive to the initial data values, because it does not evict items from the local table.

On the other hand, PLAT does better at high cardinalities for distributions without locality. The investment of cache in the first phase for partitioning pays off at the second phase when aggregation has better locality.

3.7 Tuple Width

One subtle difference between the hybrid and PLAT caching schemes is that PLAT passes down an input record when there's a miss, whereas the hybrid method passes down a partially aggregated result. In the experiments above, an input record is 16 bytes wide, while a partially aggregated result is 32 bytes wide. In situations with poor locality, the hybrid method will be copying twice as many bytes. In this particular example, both kinds of record fit in one cache line, and so the difference between the two kinds of copying is unlikely to be significant. However, one can imagine more extreme examples in which (a) many aggregates are computed from a few columns, or (b) a few aggregates are computed based on many columns. Case (a) would favor PLAT, whereas case (b) would favor the hybrid method.

4. CONCLUSIONS

We have investigated the performance of various aggregation methods on the Nehalem processor. Unlike the Niagara processors previously studied, the Nehalem performance is sensitive to data sharing. This difference is fundamentally due to different memory models. The Niagara processor does not enforce memory access order: it is the responsibility of the programmer to insert suitable `fence` instructions if a particular order is required. For computations like aggregation the order of operations is not important. In contrast, the Nehalem processor includes hardware to detect and resolve potential out-of-order execution. For general purpose computing such out-of-order events are rare, as long as threads do not modify each others' data. For aggregation, such considerations make sharing of data more expensive than alternatives that do not share data.

We also showed that detecting contention can be challenging on the Nehalem processor. Explicit contention detection via failed compare-and-swap operations is not sufficient. An alternative implementation based on the `try_lock` primitive was effective, illustrating that different architectures call for different implementation primitives for optimal performance.

It may be possible to further improve the performance of the methods presented here. For example, using a staged computation, prefetching can overlap the latency of multiple cache misses [14].

As the number of cores continues to increase in future processors, techniques such as the ones studied here will become even more important for efficient machine utilization.

5. REFERENCES

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory Access. In *VLDB*, 1999.

[2] A. Ailamaki, D. J. DeWitt, M. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[3] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

[4] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic Contention Detection and Amelioration for Data-Intensive Operations. In *SIGMOD*, 2010.

[5] D. Kanter. The Common System Interface: Intel's Future Interconnect. http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032&p=5

[6] D. Kanter. Inside Nehalem: Intel's Future Processor and System. http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719&p=7

[7] J. Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, 2007.

[8] J. Gray et al. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.

[9] D.d Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. http:/software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[10] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[11] Intel 64 and IA-32 Architecture Optimization Reference Manual. www.intel.com/Assets/ja_JP/PDF/manual/248966.pdf.

[12] J. Cieslewicz, and K. A. Ross. Data Partitioning on Chip Multiprocessor In *DaMoN*, 2008.

[13] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, 1995.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash join performance through prefetching. In *Proc. Int. Conf. Data Eng.,* 2004

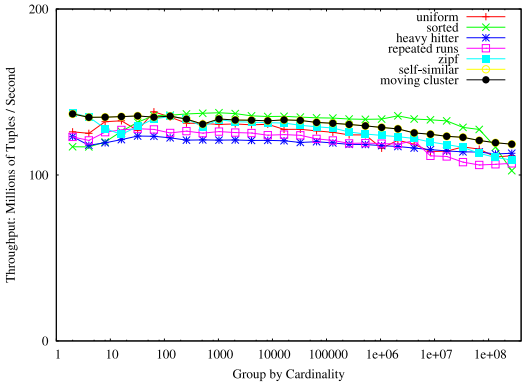


Figure 11: Partition-and-Aggregate Method on all Distributions, 8 Threads, 256-way partitioning

APPENDIX

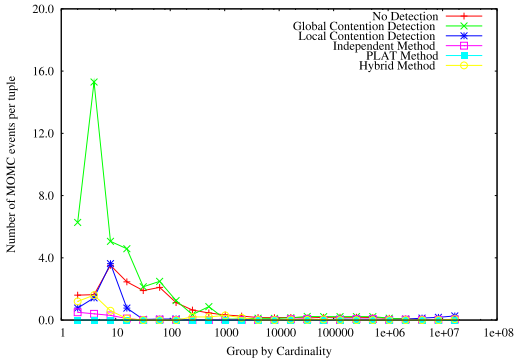
A. CONTENTION DETECTION DETAILS

The code for an atomic 64-bit add using the compare-and-swap primitive is given below as a `C` intrinsic using x86 assembly language. The `cmpxchg` opcode is the compare-and-swap instruction, and the `lock` prefix requires the instruction to be executed atomically.

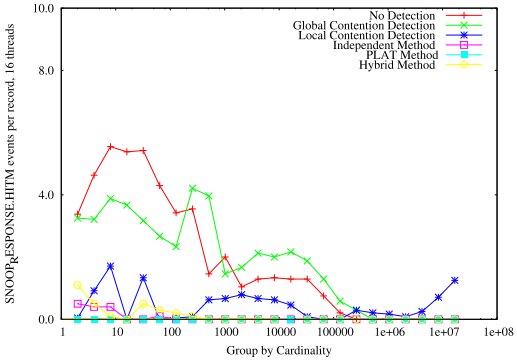
```
inline void atomic_add_64(volatile uint64_t *des,
                          int64_t src)
{
 __asm__ __volatile__("spi1:\tmovq %0, %%rax\n\t"
           "movq %%rax, %%rdx\n\t"
           "addq %1, %%rdx\n\t"
           "lock\n\t"
           "cmpxchg %%rdx, %0\n\t"
           "jnz spi1\n\t"
           :"=m"(*des)
           :"r"(src),"m"(*des)
           :"memory","%rax", "%rdx"
           );
}
```

B. ADDITIONAL EXPERIMENTS

Figure 11 shows the performance of the partition-and-aggregate method with 256-way partitioning. Figure 12 shows MOMC and HITM performance counter measurements for all methods. Figure 13 shows the performance of the various methods on a variety of distributions.

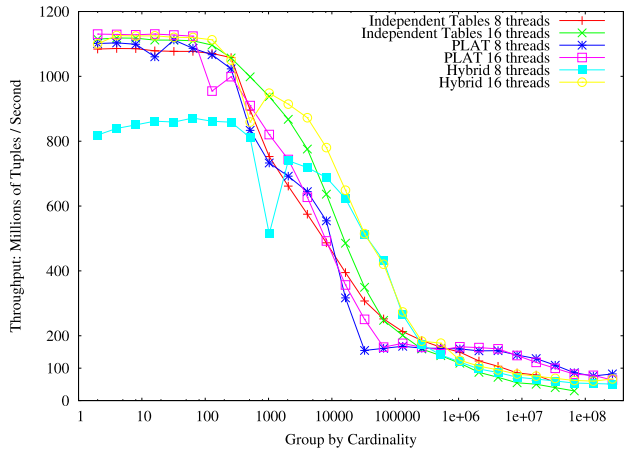


(a) MOMC events

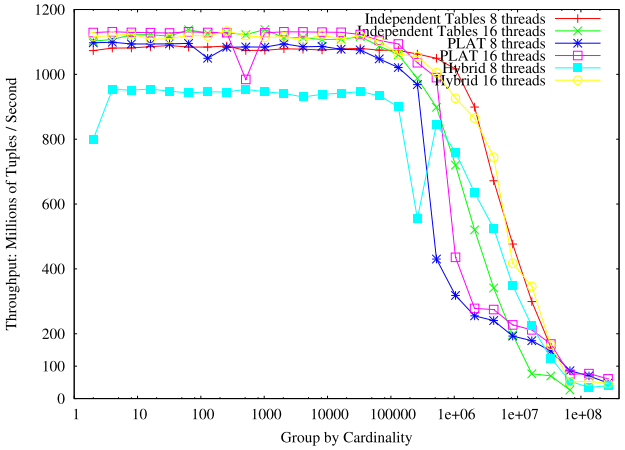


(b) Cache HITM events

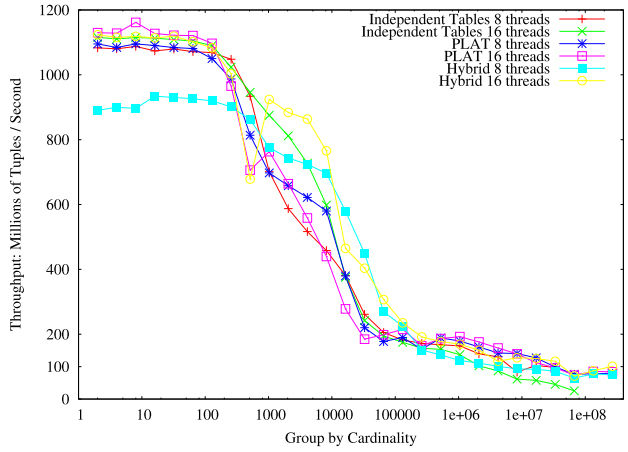
Figure 12: Performance counter events for all methods, 16 threads



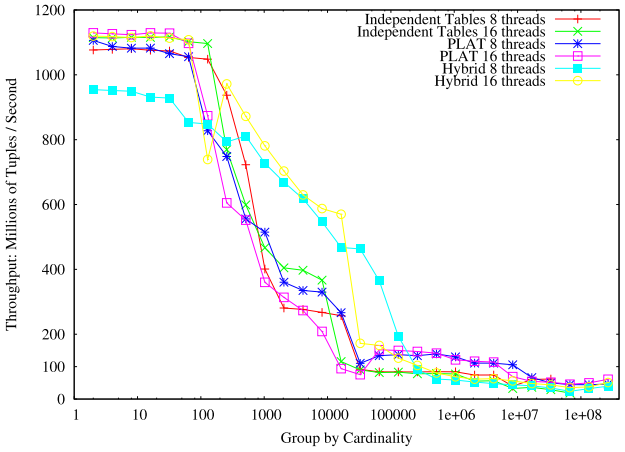
(a) Self-similar



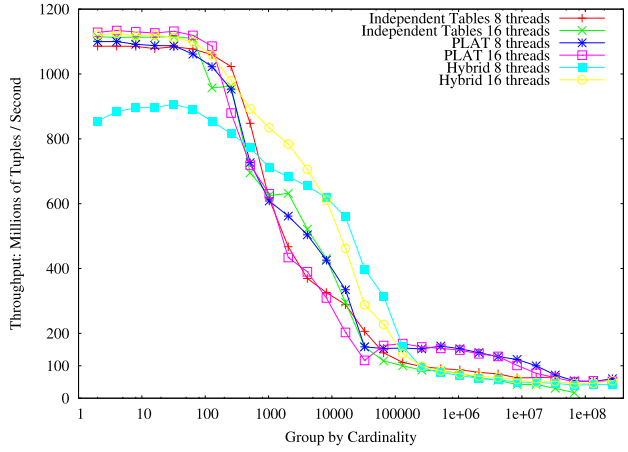
(b) Sorted



(c) Heavy Hitter



(d) Repeated Runs



(e) Zipf

Figure 13: Independent Tables, PLAT, and Hybrid, on various data distributions.

Enhancing Recovery Using an SSD Buffer Pool Extension

Bishwaranjan Bhattacharjee
IBM T.J.Watson Research Center
bhatta@us.ibm.com

Christian Lang*
Acelot Inc.
clang@acelot.com

George A Mihaila*
Google Inc.
gam@google.com

Kenneth A Ross
IBM T.J. Watson Research Center and
Columbia University
kar@cs.columbia.edu

Mohammad Banikazemi
IBM T.J.Watson Research Center
mb@us.ibm.com

ABSTRACT

Recent advances in solid state technology have led to the introduction of solid state drives (SSDs). Today's SSDs store data persistently using NAND flash memory and support good random IO performance. Current work in exploiting flash in database systems has primarily focused on using its random IO capability for second level bufferpools below main memory. There has not been much emphasis on exploiting its persistence.

In this paper, we describe a mechanism extending our previous work on a SSD Bufferpool on a DB2 LUW prototype, to exploit the SSD persistence for recovery and normal restart. We demonstrate significantly shorter recovery times, and improved performance immediately after recovery completes. We quantify the overhead of supporting recovery and show that the overhead is minimal.

General Terms

Measurement, Performance, Design, Experimentation.

Keywords

Persistence, Solid State Storage, Database Engines, Recovery

1. INTRODUCTION

Workloads that require substantial random I/O are challenging for database systems. Magnetic disk drives have high capacity, but mechanical delays associated with head movement limit the random I/O throughput that can be achieved. Newer persistent memory technologies such as NAND flash [1] and Phase Change Memory [2] remove those mechanical delays, enabling substantially higher random I/O performance. Nevertheless, devices based on these new technologies are more expensive than magnetic disks when measured per gigabyte [15]. It is therefore important for a system designer to create a balanced system in which a relatively small amount of solid state storage can be used to ameliorate a relatively large fraction of the random I/O latency.

In recent years, flash has been exploited as a storage medium for a second-level cache in a DBMS below main memory [3][7][13]. For example, multi-level caching using flash is discussed in [5]. There, various page flow schemes (inclusive, exclusive and lazy) between the main memory bufferpool and the flash bufferpool are compared both theoretically, using a cost model, and experimentally.

Flash SSDs have also been used as a write-back cache and for converting random writes to the HDD into sequential ones, like in the HeteroDrive system [6]. Here, the SSD is used primarily for buffering dirty pages on their way to the HDD, and only secondarily as a read cache, which allows blocks to be written sequentially to the SSD as well.

In the industrial space, Oracle's Exadata Smart Flash Cache [7] takes advantage of flash storage for caching frequently accessed pages. The Flash Cache replacement policy avoids caching pages read by certain operations such as scans, redos and backups, as they are unlikely to be followed by reads of the same data. Still in the industrial space, Sun's ZFS enterprise file system uses a flash resident second-level cache managed by the L2ARC algorithm [1].

In recent work [3] [8], we have prototyped a system based on IBM's DB2 LUW [4] database product that incorporates a solid state disk (SSD) as an intermediate level in the memory hierarchy between RAM and magnetic disk. This uses a temperature aware cache replacement algorithm. Here, the SSD can be thought of as an extension of the in-memory buffer pool that allows more data to be cached. Performance improves because (a) retrieving a random page from the SSD is much faster than retrieving it from the disk, and (b) there exists some locality of reference at a scale larger than the RAM-resident buffer pool. The SSD buffer pool extension is a write-through cache, meaning that it is always consistent with the state of the disk. Figure 1 shows the system overview of the SSD Bufferpools.

While most of the current work exploits the random access characteristics of flash, there has not been much emphasis on

* Work done while authors were working at the IBM T.J. Watson Research Center

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.

exploitation of its persistency. For example, in [3], the issue of crash recovery was not considered. In fact, the system described there provides limited benefits during crash recovery because the SSD directory was kept in RAM. Upon a crash, that directory would be lost, and the system would need to recover without the benefit of the data on the SSD. As recovery progresses, and the SSD bufferpool is populated, it would assist recovery by providing fast access to frequently used pages. However, all the data that was there in the SSD bufferpool before the crash would not be tapped directly despite the fact that the SSD is persistent. One could imagine a similar situation during a normal database shutdown and restart where a warm bufferpool restart, based on what is available in the SSD Bufferpool, would be useful.

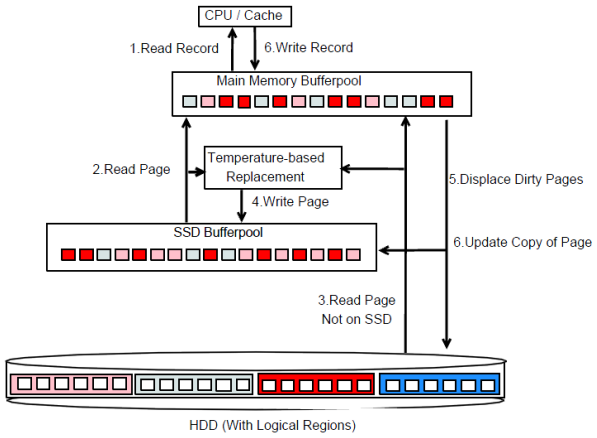


Figure 1: SSD Bufferpools without persistence exploitation

The present work extends [3] by supporting the use of the persistence of the SSD buffer pool during and after recovery as well as after a normal shutdown and restart. All further discussions will be on the recovery aspect but is equally applicable during normal shutdown and restart.

The use of the SSD persistence during recovery is motivated by the observation that recovery is itself a random I/O intensive process. The pages that need to be read and written during recovery may be scattered over various parts of the disk. With a valid SSD buffer pool, many reads can be satisfied without magnetic disk I/O. Further, the recovery process usually affects the most recently accessed pages (at least since the most recent checkpoint). These pages are likely to be in the SSD buffer pool at the time of the crash. We demonstrate significantly shorter recovery times, and improved performance immediately after recovery completes. We quantify the overhead of supporting recovery and show that the overhead is minimal.

2. HOW RECOVERY WORKS

Crash recovery entails reading the logs and acting on their contents to bring the system to a consistent state. Crash recovery has two phases of operations, namely, the *redo* phase and the *undo* phase. The *redo* phase focuses on work that is committed and needs to be reconstructed. Here, the logs are read in time order till end of log. The start point is the minimum of the oldest active transaction that is not commited (*lowtran*) and the oldest log sequence number not written to disk *(minbuff)*. If *lowtran* is greater than the *minbuff*, then we will start to read from *minbuff* to end of log and the log records will be played. If *lowtran* is less than *minbuff*, then between *lowtran* and *minbuff*, the log records are read and the transaction table is reconstructed. Under this condition, most of the log records are skipped and not played. In the *undo* phase, log records are read in reverse order and uncommitted changes are rolled back.

The work done in *redo* and *undo* would entail reading and processing data and index pages referred there. While the log record reads would be sequential, the reads of the data and indexes could be random since they would originate from different disconnected transactions and would mirror what happens in the normal usage scenario. Thus any persistent bufferpool mechanism which can preserve state beyond a crash would be of use in reducing this random IO just like in a normal usage scenario.

In database systems, although crashes are relatively rare, when they happen the time until the system recovers is a critical period. System unavailability can have a dramatic impact on an organization in many scenarios. Database systems have thus gone to length to try to reduce the time it takes to recover after a crash. Another important consideration is the time taken to reach a stable level of performance after a crash. A database system might have many service level agreements in place with its users and meeting these service guarantees is important.

3. IMPLEMENTING RECOVERY

We address crash recovery with two goals in mind. First, we want to preserve the state of the SSD buffer pool so that it can be used during crash recovery, potentially speeding up the recovery process. Second, once the system recovers, it can operate with a warm cache. Thus, it may be possible to reach a stable level of performance sooner if the SSD contents are preserved.

In order to preserve the state of the SSD, we store some SSD Bufferpool metadata on the persistent SSD storage. In particular, the slot table that maps disk pages to slots in the SSD buffer pool resides persistently on the SSD device. In the case of a crash, the slot table allows the system to locate pages that were SSD-resident at the time of the crash.

The slot directory is preserved by memory mapping it into a flash resident file. This file is initialized and of fixed length. All operations into the file are done using O_DIRECT which circumvented the OS caching. In addition after a write, a sync request is sent in to ensure all writes went through to the persistent storage. Given that the slot directory size was dictated by the number of slots in the SSD Bufferpool, the amount of memory needed for it was static. It does not change during the workload execution.

When an SSD-resident page is updated, such as when a dirty page is evicted from the RAM-resident buffer pool, no modifications of the slot table are required. On the other hand, when a new page is admitted to the SSD buffer pool and an old page is evicted, the slot table must be updated. Since these updates require additional SSD I/O relative to the base system,

they represent a potential source of performance overhead for supporting fast recovery. We quantify this overhead and demonstrate that it is minimal. We also show that compared to the size of the SSD bufferpool, the metadata table is very small and could be placed in a storage area which is faster but costlier, like a Phase Change Memory card.

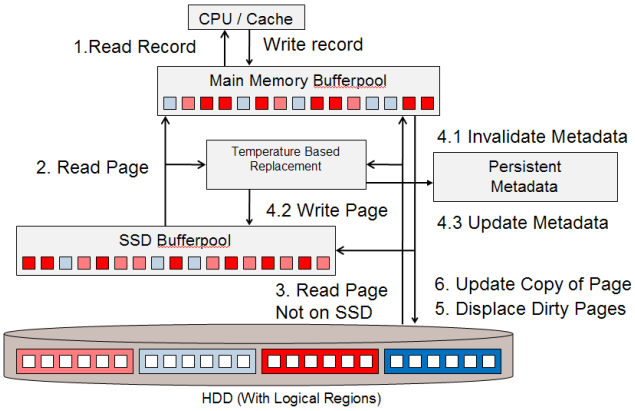


Figure 2: SSD Bufferpools with persistence exploitation

Figure 2 shows the system overview of the SSD Bufferpools with the exploitation of persistence. The slot table updates must be done carefully to ensure recoverability. In particular, the slot must initially be invalidated on persistent storage. This is shown in step 4.1 in Figure 2. After the data is written to both SSD and disk, the slot is then updated with the new page-id. This is shown in step 4.3 in Figure 2. Invalidation ensures that a subsequent partial write to the slot won't be mistaken for valid data. More subtly, invalidation allows the system to maintain the invariant that the SSD cache is always consistent with the disk, which can be critical for the correctness of recovery, as discussed in Section 4.

In this implementation, a philosophy adopted was to try to ensure we could recover a large chunk of the pages persistent in the SSD bufferpool correctly. It was not felt necessary to try to recovery all the pages that may be available there. The invalidation and validation mechanism for metadata described above would make it possible to have a valid page in the SSD Bufferpool which may be marked as invalid for a short duration of time. If the system were to crash during that period, the page will not be recovered during recovery. However it was felt that the chances of this happening was small and its overall impact on the system would be small. Thus, this mechanism puts correctness above complete recovery of all the pages in the SSD Bufferpool.

In addition to the slot directory, we also store the region temperature persistently, so that the SSD buffer pool can function normally after a restart. Without the region temperature information, all pages in the SSD bufferpool will look similar for page replacement after a crash. By storing the region temperature information and using it during recovery and beyond, we are able to differentiate between pages and victimize only the colder pages during page replacement.

4. CORRECTNESS

By explicitly writing slot array updates to persistent storage, we avoid situations in which the SSD and hard disk contain different versions of the data. It is tempting to avoid these steps in order to reduce the overhead of normal processing. For example, one possible scheme would involve skipping persistent slot array updates altogether. At recovery time, the entire SSD could be scanned, with the page-ids of the pages in each slot used to rebuild the slot array. The overhead of such a scan would be a few minutes for an entire 80GB FusionIO device, which may seem like a reasonable cost to pay for reduced I/O during normal processing.

However, this scheme has vulnerabilities. The rebuilt slot array may point to SSD pages that are out of sync with the hard disk. Consider a page write that goes to both the SSD and hard disk. If the SSD write completes before the failure while the corresponding hard disk write does not, then the SSD will contain a more recent version of the page. The converse order can happen too, in which case the SSD page may be older than the page on the hard disk. Similarly, an SSD page that had been marked for eviction but not actually overwritten with new data when the system crashed may be older than the corresponding page on the hard disk.

It may seem that having a few out-of-sync pages is only a minor inconvenience, since the recovery process can bring older page versions up to date with just a few extra I/Os. To see an example of what could go wrong, imagine an Aries-style [9] recovery mechanism in which the SSD is consulted for reads, without hard disk I/O. Suppose that the SSD has a newer version of page P than the hard disk. During recovery, the system reads P from the SSD, and sees a relatively recent log sequence number, meaning that updates from early in the log do not need to be applied. Recovery proceeds, during which time the SSD decides to evict page P to make way for warmer pages. Because the SSD is designed assuming that resident pages reflect what is on disk, no disk write of P happens on eviction. Recovery proceeds, and when the recovery manager now gets to an update on P it will see an old version of the page that is missing several updates from earlier in the log.

The converse situation in which the SSD has an earlier version of a page can potentially cause correctness violations as well. Consider a page P that was written to the hard disk (but not the SSD) just before the failure. The recovery process may read the older page from the SSD and may proceed with recovery based on the data in the older version of P, writing compensation log records to the log, but keeping the dirty page in the RAM buffer pool. Suppose that P is evicted from the SSD buffer pool, and that there is then another crash. During the second round of recovery, the system will attempt to reapply the compensation-logged updates to an incorrect version of P.

5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the proposed technique, several experiments are conducted using the industry standard TPC-C [10] benchmark. TPC-C is a popular benchmark for comparing online transaction processing (OLTP) performance on various hardware and software configurations. TPC-C simulates a complete computing environment where multiple users execute

transactions against a database. The benchmark is centered on the principal activities (transactions) of an order entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. The transactions do update, insert, delete, and abort operations and numerous primary and secondary key accesses.

While we have used the TPC-C benchmark workloads to evaluate the effectiveness of the proposed technique, the results presented are not audited or official results. The results are shown for the sole purpose of providing relative comparisons within the context of this paper and hence the tpmC scale is not indicated in the figures provided.

Before discussing the main experiment results, we will describe the hardware and software specifications used.

5.1 Hardware Specifications

The experiments were done on an IBM System x3650 Model 7979 machine [14] with a dual core AMD processor of 3GHz running Fedora Linux OS. The machine had 8GB of DRAM. In addition, it had a 80GB Fusionio SLC and a 320GB Fusionio MLC PCI-e bus based card [11]. Table 1 shows the specs of these cards.

The main hard disk based storage was provided by a DS4700 [12] with 16 SATA HDDs of 1TB each. This was connected to the server via Fiber Channel connections. The hardware experimental setup is depicted in Figure 3.

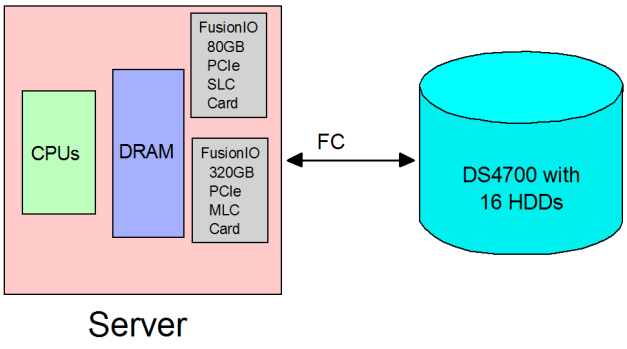


Figure 3: Hardware Experimental Setup

	80GB	320GB
NAND Type	SLC	MLC
Write Bandwidth	500 MB/Sec (32K size)	490 MB/Sec (64K size)
Read Bandwidth	750 MB/Sec (32K size)	700 MB/Sec (64K size)
IOPS (75/25 R/W mix @ 4K packet)	89549	67659
Read Access Latency	50 microsecs	80 microsecs
Wear Leveling (@ 5TB write erase per day)	24 years	16 years
Usage in experiments	SSD Bufferpool, Metadata (optionally)	Logging, Metadata (optionally)

Table 1: Hardware specification of the SSDs

5.2 Software Specifications

In our experiments with TPC-C, the scaling factor was set to 500 Warehouses. With this scaling factor the database occupied a total of 48 GB of disk space. In contrast, in the benchmark [17], 184K warehouses with 3.2TB of database space was used. So the experiment used a comparatively small database. The database was created on a tablespace striped across 16 disks of the DS4700 with the logging done on the 320GB fusionIO card. A standard TPC-C mixed workload with 16 clients was run on it. The workload consisted of the TPC-C transaction types New Orders 45%, Payment 43%, Order Status 4%, Delivery 4% and Stock Level 4%.

The main memory DB2 bufferpool was kept at 2.0% of the database size. This resulted in a main memory bufferpool hit ratio in the range of typical customer scenarios. The SSD bufferpool was created on the 80GB FusionIO card. Its size could be varied as a multiple of the main memory bufferpool size. For these experiments it was put at 3X of the main memory bufferpool.

5.3 Impact of metadata writes

In the first experiment, we varied the location of the metadata being persisted to determine the impact of the writing overhead. We used the metadata being written to DRAM via a ramdisk as the baseline. This was then compared against the metadata being written to the 80GB SLC flash card and then the 320GB MLC flash card.

The metadata file, at 23MB, was very small in comparison to the 1.2GB of the DRAM Bufferpool and the 3.6GB potential size of the SSD Bufferpool. The size of the metadata is dictated by the size of the SSD Bufferpool and the size of the database. The size does not change during the running of the workload.

As figure 4 shows, there was no noticeable degradation in the tpmC rate when the metadata was being written to flash in comparison to the ramdisk. So the performance advantages of persisting the metadata out weighs the processing and storage costs.

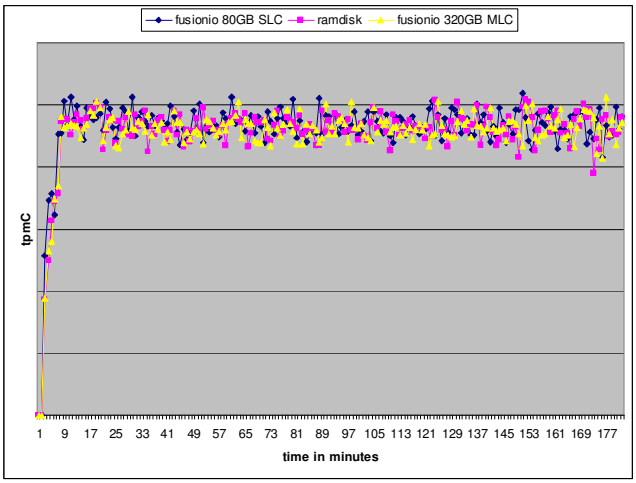


Figure 4: Impact of metadata writes on tpmC for various storage mediums

It should be noted that in this experimental setup, the workload was IO intensive with the CPU having an IO wait component

associated with it. This is typical of customer scenarios. The impact of the IO from the metadata is very small compared to the IO and associated overhead from the disks and the SSD Bufferpool. Thus, even though the random write pattern is not an ideal one for SSDs [16], the volume of such I/O requests is sufficiently small that the burden is hardly noticed.

5.4 Impact on logging

Since recovery time is so inherently tied to the logging process, we studied the impact of the SSD Bufferpool on the amount of logging that is done. There are two parameters which are important. These are the number of log records and the number of log pages that are written.

Figure 5 shows the number of log pages that were written for three configurations in an experiment. These are the base DB2 LUW and our SSD Bufferpool prototype with and without metadata. For a given time window of operation, the SSD Bufferpool prototype executes a much higher number of transactions compared to base. However, the number of log pages that are written are significantly higher for the base in comparison to the prototype. This is because the base which is running a lower transaction rate has to flush the log buffer at a lower fill factor. Thus the total number of log pages it wrote was higher. For a given total number of transactions, the number of log records would be the same although the number of log pages could vary.

During recovery, the log needs to be read. Thus a larger size of the log for base in comparison to the prototypes would mean more IO. And it would impact the recovery time. However the amount of time taken for the extra page reads does not account for all the enhancements for recovery time that we are going to demonstrate in the subsequent subsections. The improvement in recovery time comes from the exploitation of the SSD Bufferpools during recovery both explicitly by preserving the state via persistent metadata and implicitly via use of the SSD Bufferpool during recovery for caching hot pages.

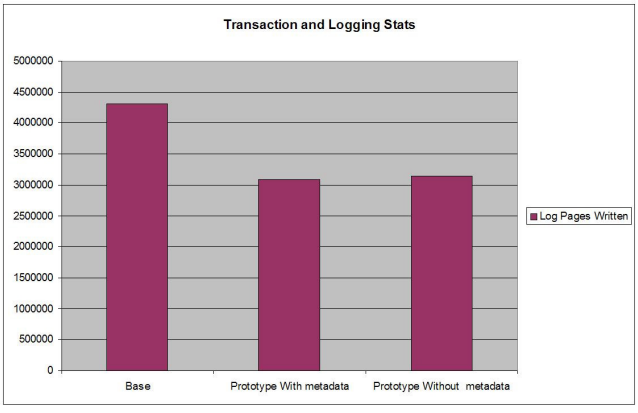


Figure 5: Amount of logging

5.5 Crash Recovery performance

To determine the effect of the SSD Bufferpool on the crash recovery time, we ran the TPC-C workload on the prototype and killed the database engine with a kill -9 after a predetermined time interval. When the database engine was restarted and the first connection to the database made, crash recovery kicked in. During this time no transactions successfully executed. This was reflected in the tpmC figure produced by the TPC-C scripts. After crash recovery completed, the tpmC figures slowly picked up and stabilized.

This experiment was done with and without the SSD Bufferpool. For the case when the SSD Bufferpool was involved in crash recovery, it was run with and without a warm restart by utilizing the persisted metadata to jumpstart the SSD Bufferpool.

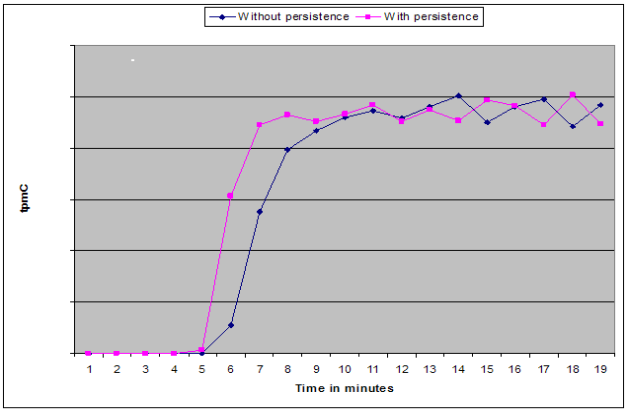


Figure 6: tpmC after crash with prototype

Figure 6 shows the tpmC rate after crash and includes recovery. With the exploitation of persistence, after 4 minutes, the recovery finishes and the transaction rate starts climbing up. By the 8th minute the transaction rate has stabilized. Without the exploitation of persistence, recovery takes 5 minutes and the transaction rate stabilizes after 10 minutes. Thus recovery and transaction stability is 20% faster. It should be noted that in other experiments, the base DB2 recovery was about 20% slower than the prototype without persistent metadata exploitation.

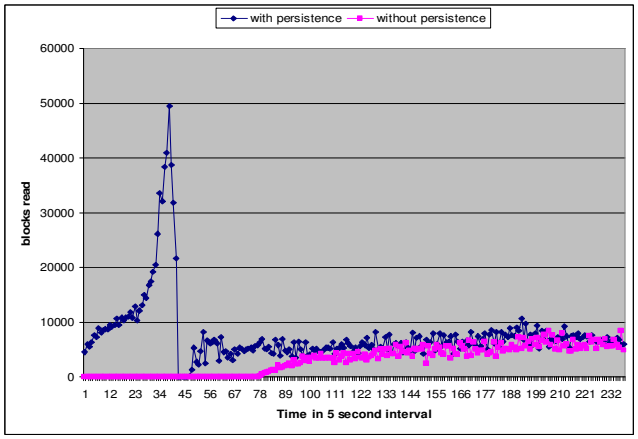


Figure 7: Reads from SSD Bufferpool after crash with prototype

Figure 7 shows the reads that are happening from the SSD Bufferpool during crash recovery and beyond. The crash recovery for the persistence case is happening in the initial 50 reading. Subsequent reads are from normal transaction processing. For the non persistence case we see no reads from the SSD

Bufferpool for the first 75 readings. Then the read rate picks up till it stabilizes to the level we see with persistence. The net benefit of exploiting persistence is not only for crash recovery but even beyond. We see the read rate peaking much faster than without persistence.

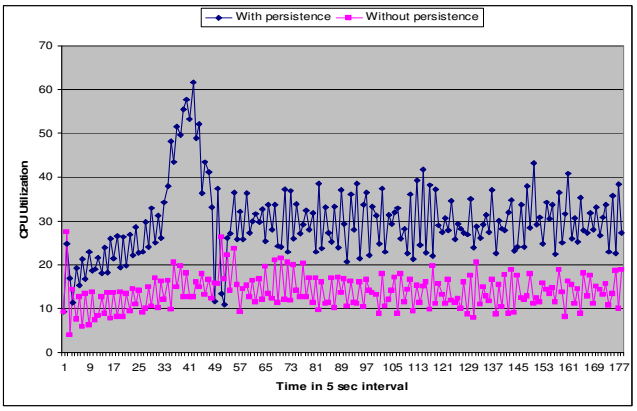


Figure 8: CPU utilization during crash recovery for prototype

The ability to feed data faster during crash recovery and beyond has a positive effect on CPU utilization as shown in Figure 8. We see that for the persistence case, CPU usage peaks during recovery and then stays higher than the non persistence case as data is fed faster from the SSD Bufferpool.

5.6 Restart performance

We now evaluate the impact of persistence on normal database shutdown and restart. In this case, we ran TPC-C for a fixed amount of time and then shutdown the database engine. When the engine was restarted, it was run with and without a warm bufferpool restart from the persistent bufferpool which was on a ramdisk.

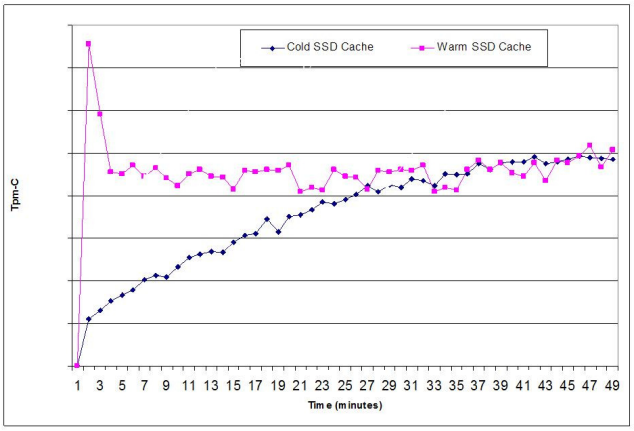


Figure 9: tpmC comparison for restart on prototype

Figure 9 shows a comparison between the two cases. For warm restart, we see the tpmC initially peaking and then falling down to a stability level. The peak is for the period when the DRAM Bufferpool is filling up from the SSD Bufferpool and there is no dirty page cleaning to the HDDs. The subsequent drop is attributed to the fact that the SSD Bufferpool is a write through cache. Thus pages need to be updated to the HDD. In the cold SSD cache case, the tpmC slowly climbs up and ultimately reaches the rate of the warm SSD Bufferpool.

6. CONCLUSION

We presented our work on the exploitation of the persistence of Solid State Disks in the context of enhancing recovery and restart in a database engine. Most current work focuses on the exploitation of the random access capability of Solid State Disks. This includes our previous work on SSD Bufferpools. In this paper we have described our extensions to that previous work for supports the use of the persistence of the SSD bufferpool during and after recovery and normal restart. We demonstrate significantly shorter recovery times, and improved performance immediately after recovery completes. We quantify the overhead of supporting recovery and show that the overhead is minimal.

In future work we plan to look at other recovery mechanisms. Our performance was limited by the write through nature of our cache: cold reads and all writes still need to go to the hard disk. A write back cache could potentially perform better in this respect since the SSD cache is persistent and has additional I/O capacity. A write back cache would need to carefully address questions of consistency and recoverability. Some very recent work [18] has examined such policies in the context of Microsoft SQL Server 2008 R2, but recovery and restart times are not explicitly described. Since recovery methods differ between these two commercial systems (DB2 uses a fuzzy checkpoint, whereas SQL Server uses a sharp checkpoint that flushes all dirty pages to disk), the best way to employ SSDs may be different.

7. REFERENCES

[1] A. Leventhal. Flash storage memory. Communications of the ACM, 51(7):47–51, 2008.

[2] R. F. Freitas and W.W.Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4-5):439.448, 2008

[3] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435.1446, 2010.

[4] DB2 for Linux, UNIX and Windows. http://www-01.ibm.com/software/data/db2/linux-unix-windows

[5] I. Koltsidas and S. Viglas. The case for flash-aware multi level caching. Internet Publication, 2009. http://homepages.inf.ed.ac.uk/s0679010/mfcache-TR.pdf.

[6] S.-H. Kim, D. Jung, J.-S. Kim, , and S. Maeng. HeteroDrive: Re-shaping the storage access pattern of oltp workload using ssd. In Proceedings of 4th International Workshop on Software Support for Portable Storage (IWSSPS 2009), pages 13–17, October 2009.

[7] A technical overview of the Sun Oracle Exadata storage server and database machine. Internet Publication, September 2009. http://www.oracle.com/technology/products/bi/db/exadata/pdf/Exadata_Smart_Flash_Cache_TWP_v5.pdf.

[8] B. Bhattacharjee, M. Canim, C. Lang, G. Mihaila and K. Ross. Storage Class Memory Aware Data Management, IEEE Data Engineering Bulletin, 2010

[9] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB 1990:

[10] TPC-C, On-Line Transaction Processing Benchmark, http://www.tpc.org/tpcc/.

[11] Fusionio drive specifications. http://www.fusionio.com/load/media-docsProduct/kcb62o/Fusion_Specsheet.pdf

[12] DS4700 specifications : http://www-03.ibm.com/systems/storage/disk/ds4000/ds4700/

[13] B. Khessib, Using Solid State Drives As a Mid-Tier Cache In Enterprise Database OLTP Applications, TPCTC 2010

[14] http://www.redbooks.ibm.com/xref/usxref.pdf

[15] J. Handy. Flash vs DRAM price projections - for SSD buyers. www.storagesearch.com/ssd-ram-flash%20pricing.html.

[16] R. Stoica, M. Athanassoulis, R. Johnson, A. Ailamaki. Evaluating and Repairing Write Performance on Flash Devices, DaMoN 2009.

[17] http://www.tpc.org/results/FDR/TPCC/IBM-x3850X5-DB2-Linux-111610-TPCC-FDR

[18] J. Do et al. Turbocharging DBMS Buffer Pool Using SSDs, SIGMOD 2011.

How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled?

Henrik Mühe
TU München
Boltzmannstr. 3
85748 Garching, Germany
muehe@in.tum.de

Alfons Kemper
TU München
Boltzmannstr. 3
85748 Garching, Germany
kemper@in.tum.de

Thomas Neumann
TU München
Boltzmannstr. 3
85748 Garching, Germany
neumann@in.tum.de

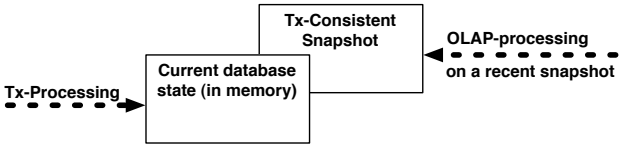
ABSTRACT

The quest for real-time business intelligence requires executing mixed transaction and query processing workloads on the same current database state. However, as Harizopoulos et al. [6] showed for transactional processing, co-execution using classical concurrency control techniques will not yield the necessary performance – even in re-emerging main memory database systems. Therefore, we designed an in-memory database system that separates transaction processing from OLAP query processing via periodically refreshed snapshots. Thus, OLAP queries can be executed without any synchronization and OLTP transaction processing follows the lock-free, mostly serial processing paradigm of H-Store [8]. In this paper, we analyze different snapshot mechanisms: Hardware-supported Page Shadowing, which lazily copies memory pages when changed by transactions, software controlled Tuple Shadowing, which generates a new version when a tuple is modified, software controlled Twin Tuple, which constantly maintains two versions of each tuple and HotCold Shadowing, which effectively combines Tuple Shadowing and hardware-supported Page Shadowing by clustering update-intensive objects. We evaluate their performance based on the mixed workload CH-BenCHmark which combines the TPC-C and the TPC-H benchmarks on the same database schema and state.

1. INTRODUCTION

Harizopoulos et al. [6] investigated the performance bottlenecks of traditional database management systems and found out that 31% of the time is spent on synchronization and 35% on page and buffer management. Consequently, a new generation of main memory DBMS has been engineered to remove these bottlenecks. Overhead caused by page and buffer management can be removed in in-memory DBMS by relying entirely on virtual memory management instead of devising costly software controlled mechanisms. One of the most prominent examples of this generation of in-memory database systems is VoltDB [15], a commercial product based on the H-Store research prototype [8]. There, costly mechanisms like locking and latching for concurrency control are avoided by executing all transactions sequentially, therefore yielding serializability without overhead. In order to increase the level of parallelism, the database can be logically partitioned to accommodate one transaction per partition [3].

Lockless, sequential execution has proven to be very efficient for OLTP workloads, specifically short transactions that modify only a handful of tuples and terminate within microseconds. With the need for "real-time business intelligence" as advocated by Plattner of SAP [13] among others, serial execution is bound to fail. Long-running OLAP queries cannot be executed sequentially with short OLTP transactions since transaction throughput would be severely diminished every time a long-running query stops OLTP transactions from being executed. To solve this dilemma, we advocate the use of consistent snapshots for the execution of long-running OLAP queries which yields both: High OLTP throughput by executing OLTP transactions sequentially as well as a way of executing OLAP queries on a fresh, transaction consistent state of the database, as sketched:



This mechanism has previously been demonstrated in our research prototype, HyPer [9]. For this approach to work, being able to efficiently maintain and create a consistent snapshot of the database at short intervals of seconds is paramount. In this work, we will examine different techniques for maintaining a consistent snapshot of the database without diminishing OLTP performance. We go beyond what has been done in previous work, for instance by Lorie [11] or Cao et al. [1]. First, we extend the mechanisms to enable high performance query execution on snapshots as their most important use – instead of just recovery as previously suggested, e.g., by Molina et al. [5] or in [14]. Therefore, our approaches yield higher order snapshots for query processing as opposed to those snapshots primarily used for recovery. Second, we adapt the mechanisms for use in main memory database systems: In case of Lorie's shadowing approach, we show how the limitations when used in on-disk database systems can completely be alleviated in main memory. With Cao et al.'s twin objects approach (called ZigZag approach in [1]), we extended the implementation for use in a general purpose database system instead

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.

of a specialized application. Third, we offer a thorough evaluation of the different approaches taking both OLTP as well as OLAP throughput into account.

The remainder of this paper is structured as follows: In Section 2, we introduce the concept of virtual memory snapshots. In Section 3, we will reexamine different techniques for maintaining a consistent snapshot of the database and discuss our implementations and the improvements we added to make them viable for OLAP query execution. Section 4 offers a classification of these snapshotting techniques. In Section 5, all techniques are evaluated using a combination of the TPC-C and TPC-H benchmarks called the CH-BenCHmark [4]. Section 6 concludes this paper.

2. HARDWARE SNAPSHOTTING

In this section, we will focus on hardware supported virtual memory snapshotting as proposed in [9]. Purely software-controlled as well as hybrid approaches will be described in the next section.

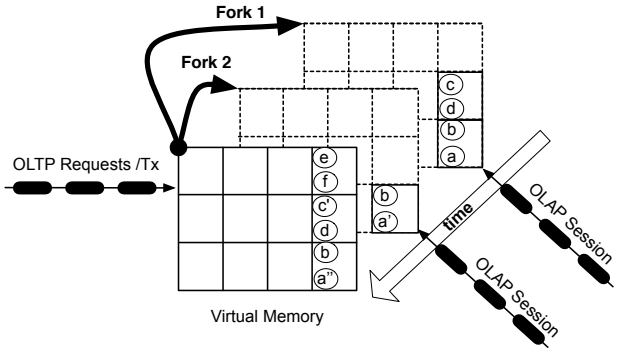


Figure 1: Hardware Page Shadowing with multiple snapshots taken at different transaction consistent states of the database

Hardware Page Shadowing is a new snapshotting technique that we developed in our HyPer main memory database system. It creates virtual memory snapshots by cloning (forking) the process that owns the database. In essence, the virtual memory snapshot mechanism constitutes an OS/hardware supported shadow paging mechanism as proposed by Lorie [11] decades ago for disk based database systems. However, the original proposal incurred severe costs as it had to be software-controlled and it destroyed page clustering on disk. Additionally, virtual memory is not replaced by Lorie's approach, but instead an additional layer of indirection is added, further decreasing performance. None of these drawbacks occurs in virtual memory snapshotting as clustering across RAM pages is not an issue – as we examined in microbenchmarks. Furthermore, the sharing of pages (between OLTP and OLAP snapshots) and the necessary *copy-on-update* is managed by the operating system, efficiently supported by the MMU hardware (memory management unit). This way, translations between virtual and physical addresses via the page table as well as page replication (*copy-on-update*) do not need to be implemented in the database management system. Replicating a page is highly efficient, taking only 2μs for a 4kb page as we measured in a microbenchmark on a standard processor.

Virtual memory snapshots exploit the OS facilities for memory management to create low-overhead memory snapshots. All modern operating systems and hardware support and widely use virtual memory management. This means that physical memory is not directly assigned to a process that requests memory but is mapped through a layer of indirection called virtual memory. All memory accesses use virtual addresses and do not need to know which physical memory pages back a given virtual address. Translations from virtual to physical addresses are done in hardware by providing a lookup table to the memory management unit of the CPU as per the specifications of the processor vendor.

In unix environments, new processes are created by cloning an existing process using the `fork` system call. Since an identical copy of an existing process has to be created, all memory used by this process has to be copied as well. With virtual memory, an eager copy of the memory pages used by the process issuing the fork system call (parent process) is not required, only the table used to translate virtual to physical addresses used by the parent process needs to be copied. Therefore, the physical pages backing both the parent's as well as the child's virtual address space are shared at first. All shared physical pages are marked as read-only during the execution of the fork system call. When a read-only page is modified by either process, a CPU trap is generated causing the operating system to copy the page and change the virtual memory mapping to point to the copy before any modifications are applied. Effectively, this implements a hardware controlled copy-on-write mechanism.

Applied to main memory database systems, we use the fork to generate a lazy copy of the database system's memory with little delay. In order for this snapshot to be consistent, we execute the fork system call in between two serial transactions. This is not strictly necessary though, since undo information is also part of the memory copy. Therefore it is possible to quiesce transaction execution without waiting for transactions to end, execute the fork system call and clean the action-consistent snapshot using the undo log.

The forked child process obtains an exact copy of the parent processes' address space, as exemplified in Figure 1 by the overlaid page frame panel. This virtual memory snapshot will be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 1.

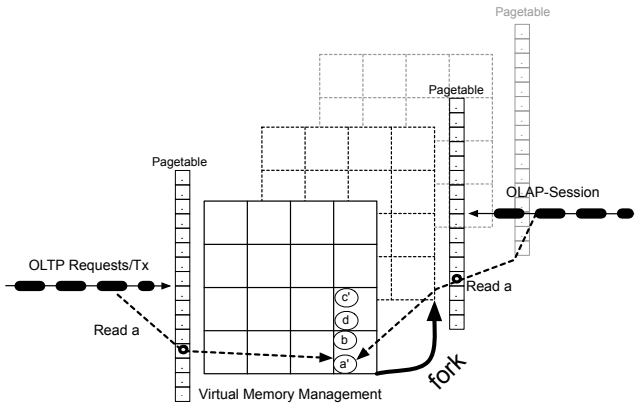


Figure 2: The page table after invoking the `fork` system call.

The snapshot stays in precisely the state that existed at the time the `fork` took place. Fortunately, state-of-the-art operating systems do not physically copy the memory segments right away. Rather, they employ a lazy *copy-on-update* strategy – as sketched out in Figure 2. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., for object a) to the same physical main memory location. The sharing of the memory segments is highlighted in the graphics by the dotted frames. A dotted frame represents a virtual memory page that was not (yet) replicated. Only when an object, like data item a', is updated, the OS- and hardware-supported copy-on-update mechanism initiates the replication of the virtual memory page on which a' resides as is illustrated in Figure 3. Thereafter, there is a new state denoted a'' accessible by the OLTP-process that executes the transactions and the old state denoted a', that is accessible by an OLAP query session. As shown in Figure 1, multiple snapshots representing different consistent states of the database can be maintained with low overhead. Here, an older snapshot is shown which was taken before data item a was modified to a'. The page on which data item a lies is a copy denoted by the solid border of the page, most other pages are shared between all snapshots.

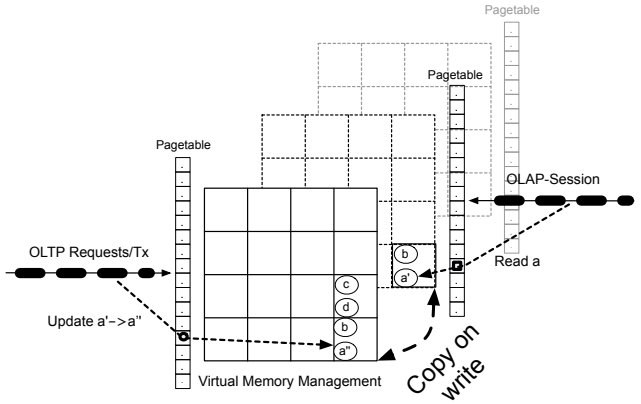


Figure 3: The page table after a page was modified, causing a page copy on update.

Unlike the figure suggests, the additional page is really created for the OLTP process that initiated the page change and the OLAP snapshot refers to the old page – this detail is important for estimating the space consumption if several such snapshots are created. It can be expected that most pages and objects on those pages are of the nature of the page inhabited by e and f. That is, they contain older, no longer mutated objects.

3. SNAPSHOTTING IMPLEMENTATIONS

We implemented a total of four snapshot mechanisms, each anchored deeply within the storage scheme of our main memory database system prototype. All storage backends integrate a mechanism which maintains a consistent snapshot that can be periodically refreshed within short intervals of seconds to minutes.

3.1 Hardware Page Shadowing (VM-Fork)

In Figure 4, the backend implementation of the virtual memory snapshotting approach is illustrated on a more detailed level. Here, a relation spanning multiple pages as well as SQL statements executed on this relation are shown. Statement A) deletes a tuple which is deleted in-place in the memory accessed by the OLTP process. Since the memory used by OLAP queries is lazily shadow copied, the deletion causes an actual copy of the modified page, denoted by the gray memory used by OLAP queries in the background of the figure. Statement B) modifies a tuple, again causing the page that has previously been shared between OLTP transactions and OLAP queries to be copied. Insertions – as exemplified by statement C) – are performed at the end of the vector which – contrary to the depiction in Figure 4 – is not maintained in any specific order to allow for high OLTP throughput without any added level of indirection for newly inserted tuples. The architecture sketched here has been successfully implemented in our main memory database research prototype HyPer [9].

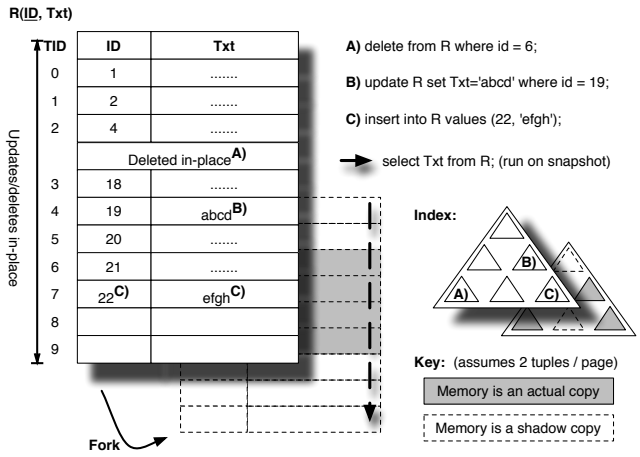


Figure 4: Query execution when using hardware page shadowing

3.2 Tuple Shadowing

Instead of shadowing on page level, shadow copies can be created on tupel level, thus possibly lowering the memory overhead of keeping a consistent snapshot. To manage per tuple shadow copies, we have to resort to software control. Thus, a more complex indirection has to be established since for each access, the current version of the data being used has to be determined on a per-tuple level. No hardware mechanism can be specifically exploited to speed-up tuple-shadowing, thus all indirection has to be dealt with in software.

A straight forward implementation of tuple shadowing is shown in Figure 5. There, a consistent snapshot was taken after the insertion of the tuple with TID 7. When a tuple with TID lower or equal to 7 is modified, modifications are not done in-place but rather by copying the tuple to the end of the relation and establishing a link between the original and the shadow copy. Statement B) is an example of such an update. Instead of changing the tuple with TID 5, the tuple is copied to the end of the relation and modified there. The original tuple is annotated with the TID of its shadow copy

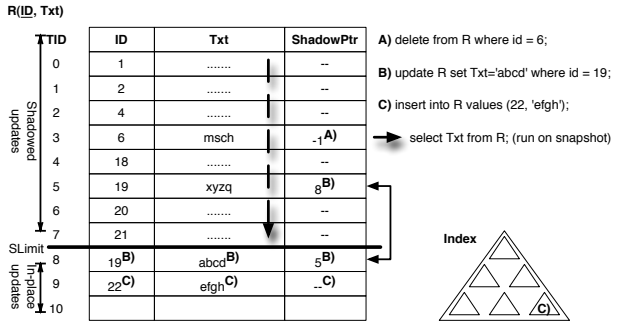


Figure 5: Implementation of the Tuple Shadowing approach.

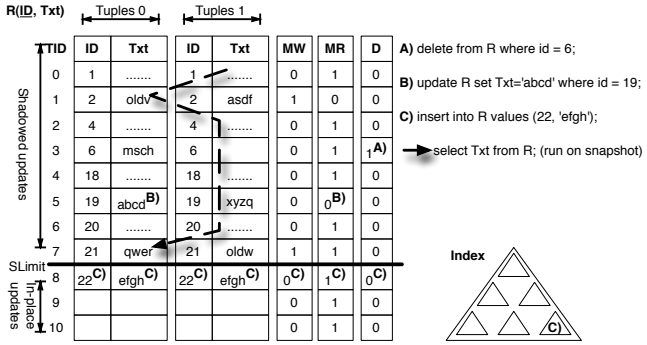


Figure 6: Implementation of the Twin Tuples approach.

to allow merging the two different version of the tuple at a later time and to allow different readers to access the two different versions. Since no index update is performed, all accesses to both the new and the old version of the tuple go through the original index entry, adding indirection. Deletions of tuples – as exemplified by statement A) – are not executed in place but rather accomplished by flagging the tuple in question as deleted. Insertions – like statement C) – are performed at the end of the relation without maintaining a specific order.

All OLTP accesses have to be directed to the most recent version of the tuple. Therefore, these accesses have to consult the pointer saved in the *ShadowPtr* field to check whether or not a more recent version of the data exists. OLAP queries work on the consistent snapshot and thus do not need to check the *ShadowPtr* for a more recent version. Additionally, checking whether or not a tuple has been flagged as deleted is not necessary for snapshot accesses, since deletion flags only apply to the most recent version of all tuples used by OLTP queries. Refreshing the snapshot incurs substantial copy and merge costs.

3.3 Twin Tuples

To mitigate the overhead caused by periodically merging the database as is necessary in Tuple Shadowing, a technique referred to as the Twin Block approach or – when done on a per tuple level – Twin Tuples approach can be employed [1].

In the Twin Tuples approach, two copies of every data item exist as is illustrated in Figure 6. Two bitmaps indicate which version of a tuple is valid for reads and writes by OLTP transactions. Reads are performed on the tuple denoted by the *MR* bit, the tuple that writes are performed on is denoted by the *MW* bit. A consistent snapshot of the data can be accessed by always reading the tuples that are not modified as indicated by the negation of the *MW* bit.

Figure 6 shows that deletions like statement A) are performed by flagging the tuple in question as deleted. Insertions like statement C) are again performed at the end of the storage vector without maintaining a specific order. Updates on the other hand are now performed on one of the two stored tuples. When the first update on a tuple is performed, the *MR* flag is set to the value of the *MW* flag and the update is performed on the tuple denoted by the *MW* flag. Since *MW* is atomically toggled only when the consistent snapshot is being refreshed, for the duration of a snapshot the tuple denoted by $\neg MW$ is never written to and thus stays in a consistent state.

This behavior is exemplified by statement B) updating the tuple with TID 5. Here, the original tuple is read from the tuple denoted by *MR*, an update is performed and the result is written to the twin location denoted by *MW*. Afterwards, *MR* is set to *MW*.

OLAP queries read the tuples that have not been written, that is, tuples in twin location $\neg MW$. Because the value of the *MW* flag can vary for different tuples, scans are performed in a zigzag pattern on the data array, potentially causing diminished scan performance.

3.4 HotCold approach

With hardware page shadowing, an update to a single value on a page causes the entire page to be copied. The HotCold approach is intended to cluster update-intensive tuples to the so called hot section in memory. Updates which would modify a tuple which is not in the hot section are copied to that section and marked as deleted in the cold section. That way, modifications only takes place in the hot part. The technique is a combination of Tuple Shadowing and Hardware Page Shadowing as the update clustering is software controlled whereas shadow copying is done using the VM-Fork mechanism.

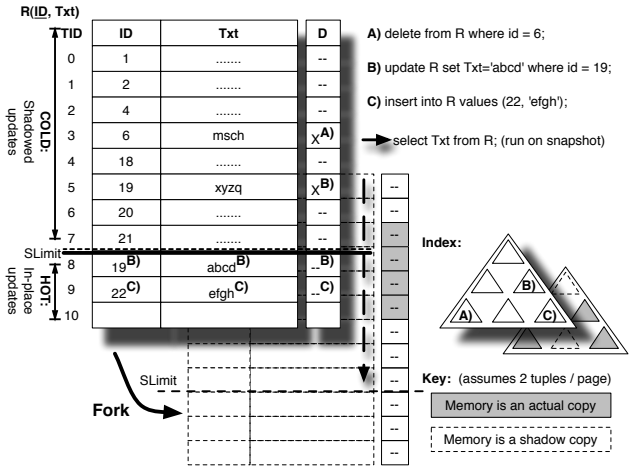


Figure 7: Implementation of the HotCold approach.

In Figure 7, statement B) modifies a tuple with a TID before the current *SLimit*, which makes the tuple a cold

tuple. Instead of modifying the tuple in place, a deletion flag is set and the tuple is copied to a slot inside the hot part of the store. There, modifications are applied in-place.

Statement C) in Figure 7 inserts tuples at the end of the vector equally to the way tuples are inserted in Tuple Shadowing. This – on average – does not cause a page to be copied since the vector is lazily backed with physical memory pages and areas that have not been backed yet are not copied via the hardware page shadowing mechanism. Deletions in the cold part of the store are performed by flagging a tuple as deleted as can be seen with statement A).

3.5 Index structure synchronization

For approaches where index structures are used both for OLTP transactions as well as for OLAP queries, we have to take index synchronization into account. When index structures are shared, updates to the index can conflict with lookups. We examined four approaches to alleviate this problem:

1. Abandon indexes for OLAP queries or creating OLAP indexes on demand.
2. Eagerly copying indexes when a snapshot is created.
3. Employing hardware page shadowing to lazily maintain index snapshots after database snapshot creation.
4. Latching indexes to synchronize conflicting index operations.

Approach 1) is interesting when all OLAP queries rely entirely on table scans with no particular order. Since none of our implementations guarantees any specific order on the data, we assume that having indexes available on the snapshots is oftentimes required and thus do not further investigate this option. Additionally, 1) does not require any specific implementation or synchronization considerations.

Approach 2) generates a separate copy of the indexes for each snapshot by eagerly duplicating the index when a snapshot is created Therefore, no synchronization is necessary as no indexes are shared but reorganization speed decreases. Hardware page shadowing for indexes as done in approach 3) achieves the same result but does not create a complete copy of the indexes. Rather, it copies only the page table of the pages used to store index data and applies the *copy-on-update* mechanism as introduced in Section 3.1. In the last approach, 4), index data between OLTP transactions and OLAP queries is shared making synchronization necessary.

For our Hardware Page Shadowing approach as well as for the HotCold approach, sharing an index structure between OLTP transactions and OLAP queries is implicit with the process cloning/forking. Thus, index latching and shared usage of the index is not applicable with these approaches. Approaches 1), 2) and 3) are applicable.

4. CLASSIFICATION

The following section contains a classification of the different snapshotting techniques examined in this paper.

4.1 Snapshotting method

The techniques discussed in this work can be subdivided by the method they use to achieve a consistent snapshot while still allowing high throughput OLAP transactions on the data. The HotCold approach as well as the plain hardware page shadowing approach use a hardware supported copy on write mechanism to create a snapshot. In contrast to that, tuple shadowing as well as the twin object approach use software mechanisms to keep a consistent snapshot of the data intact while modifications are stored separately. This is also displayed in Figure 8 where all techniques are classified by whether snapshot maintenance is done in software, in hardware or both:

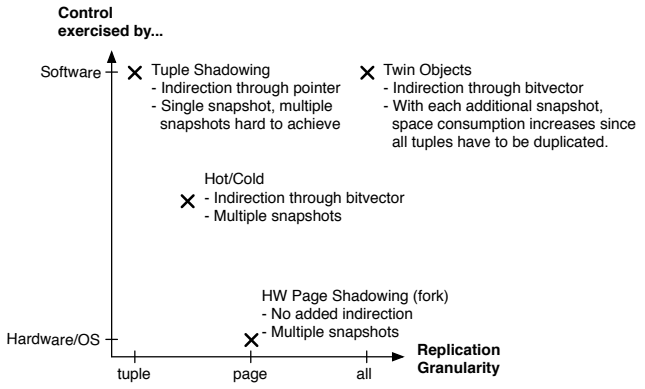


Figure 8: Techniques classified by granularity and control mechanism.

The snapshotting mechanism has a direct impact on the amount of reorganization required when the snapshot needs to be refreshed. The hardware supported page shadowing approach requires no reorganization whatsoever, only the OLTP process needs to be quiesced and the `fork` system call needs to be executed. Since the HotCold approach relies on the same mechanism to generate a consistent snapshot, no reorganization is required either but an optional reorganization can be performed. This saves memory by actually removing tuples that have been flagged as deleted and it also increases scan performance as a deletion flag checks become unnecessary during scans.

With software based snapshotting approaches, reorganization is mandatory on snapshot refresh. First, tuples flagged as deleted need to be actually removed or at least marked as unused so that they can be overwritten with new tuples at a later point. In case of tuple shadowing, updates saved in shadow copies have to be written back to the original version to prevent a long chain of versions from forming which would linearly increase the level of indirection when accessing tuples. Whereas the approaches based on hardware page shadowing only required quiescing the OLTP process, software based snapshotting techniques usually require the entire database to quiesced incurring

To conclude, approaches maintaining a snapshot using software mechanisms require a reorganization phase to refresh that snapshot whereas approaches relying on hardware page shadowing need no reorganization or at most an optional reorganization phase.

4.2 Indirection

With hardware page shadowing, all indirection required is handled by the operating system's virtual memory mechanisms. Since virtual memory is used by all approaches since direct allocation of physical memory is neither useful nor technically simple, no additional level of indirection is added by hardware supported page shadowing.

Software based approaches introduce a level of indirection: Tuple shadowing keeps a pointer to updated shadow copies

of each tuple forcing OLTP queries to check whether a newer version exists or not. The twin object approach requires a bitmap to be checked on each read access and thereby introduces indirection on tuple access.

4.3 Memory overhead and granularity

As the name suggests, hardware supported page shadowing uses a page as its smallest granularity level causing an entire page to be copied on modification. This is also displayed in Figure 8 where all techniques are classified by the granularity in which memory consumption grows due to modification.

Since in hardware page shadowing all pages end up being replicated in a worst case scenario, the memory used for OLTP transaction processing is at most doubled to maintain one consistent snapshot for OLAP processing. Because of the page level granularity, not all tuples need to be modified to cause worst-case memory consumption: If at least one bit is modified on each page, all pages will end up being copied.

Compared to pure hardware page shadowing, the HotCold approach lowers the rate at which memory consumption increases. This is done by clustering updates in a designated part of the memory called the hot area. In a worst case scenario, memory consumption still doubles to maintain a consistent snapshot, but every tuple has to be modified to cause worst case behavior. Thus, the HotCold approach effectively decreases the speed at which memory is consumed by OLTP transactions.

Tuple shadowing as well as twin objects work with a per-tuple granularity. Tuple shadowing copies a tuple on modification thus increasing memory consumption linearly with the number of modified tuples. Twin objects saves two versions of each tuple by default, thus exhibiting worst case memory consumption right from the start – the approach is mainly used to illustrate the varying degrees of overhead introduced by reorganization.

4.4 Concurrency in indexes

A low cost snapshot of the database does not necessarily allow for high performance query execution. One of the reasons is that metadata like indexes are missing. In section 3.5, we introduced four ways of dealing with indexes which we will now revisit for a classification.

4.4.1 Abandoning indexes

The trivial solution of abandoning all index structures consumes no additional memory and at the same time offers no help when accessing data form OLAP queries. Required indexes can be regenerated online incurring a runtime performance overhead during query execution.

4.4.2 Eager index copy

Indexes can be duplicated eagerly when the snapshot is taken. This results in an increase in memory consumption but retains indexes for use in OLAP queries. Since OLTP as well as OLAP queries need to be quiesced for index copies to be generated in a consistent fashion, the additional time spend while refreshing a snapshot decreases OLTP as well as OLAP throughput. At transaction and query runtime, no overhead is incurred.

4.4.3 Index fork

Equivalently to data, indexes can be copied using the hardware snapshotting technique discussed in section 3.1. In a worst case scenario, memory consumed by indexes duplicates over time as index entries are updated. When the snapshot is created, the only delay incurred is the duration of the fork system call which is short compared to eagerly copying the entire index (see Figure 5.1). At runtime, pages which are modified for the first time have to be copied. This is done by the OS/MMU and takes only about 2 microseconds for a 4 kilobyte page [9].

4.4.4 Index synchronization

For techniques where index sharing is possible, namely Tuple Shadowing and Twin Objects, inconsistencies due to concurrent access have to be prevented. This can be done by latching index structures so that writers get exclusive access to an index whereas multiple readers can access it concurrently. In this case, indexes do not have to be duplicated but the latches incur a comparably small memory overhead. Minimally, every index access has to pass at least one latch. Thus the runtime overhead for this approach consists of the time it takes to acquire a latch as well as possible wait-time in case the latch is held by another process.

4.5 Classification summary

Backend	Snapshot mechanism	Indir.	Gran.	Index sharing
Fork	hw	VM only	page	n/a
Tuple	sw	VM + ptr	tuple	yes
Twin	sw	VM + bit	all	yes
HotCold	hw/sw	VM + bit	tuple	n/a

Figure 9: Classification overview between all presented techniques and index synchronization mechanisms.

5. EVALUATION

In this section, all proposed techniques for the hybrid execution of OLTP transactions and read only OLAP queries will be thoroughly evaluated.

5.1 Snapshotting performance

For all techniques, OLTP processing has to be quiesced when the snapshot is refreshed. Since this directly impacts OLTP transaction throughput, we measured the total time it takes before OLTP processing can be restarted. For techniques employing hardware page shadowing, the time required to finish the `fork` system call is measured. For software based approaches, the time required for memory reorganization is measured. When reorganization is optional, the time required for the optional part of the reorganization is given in braces.

Backend	4kb pages	2mb pages
VM-Fork	47ms	13ms
Tuple	500ms	483ms
Twin	94ms	85ms
HotCold	50ms	13ms
	(2829ms)	(2097ms)

Table 1: Reorganization time by backend, optional reorganization runtime given in brackets.

Table 1 shows the time required to refresh a snapshot for the different techniques. Reorganization took place after loading the data of the TPC-C benchmark scaled to 5 warehouses and then running the TPC-C transaction mix until a total of 100,000 transactions (roughly 44,000 neworder transactions) were finished. A snapshot of the database containing the data that was initially loaded was maintained while executing the transactions.

5.2 Raw scan performance

In addition to the tests conducted during the execution of the CH-BenCHmark, we measured scan performance in a microbenchmark setting. First, we evaluated the time it takes to determine which tuples inside the store are valid, that is, time to find all valid TIDs. Second, we evaluated the predicates *min(4b)* and *min(50b)* which determines the lowest value for a 4 byte integer and for a 50 byte string, respectively.

Backend	Valid Tuples	4kb pages		2mb pages	
		Min(4b)	Min(50b)	Min(4b)	Min(50b)
VM-Fork	72ms	188ms	702ms	186ms	701ms
Tuple	72ms	216ms	715ms	212ms	708ms
Twin	74ms	242ms	813ms	250ms	796ms
HotCold	146ms	199ms	769ms	197ms	767ms

Table 2: Scan performance on snapshot after removing 1% and updating 2% of the tuples.

The two queries were run on a snapshot taken after 30 million tuples were loaded into the table being tested. Before running the queries, OLTP transactions changing a total of 2% of the tuples inside the table and deleting another 1% were run.

The time it takes to determine all valid TIDs is given as the 'Valid Tuples' value. It is a baseline for table scan execution speed. 'Valid Tuples' performance is inferior on the HotCold store. This stems from the fact that reorganization of that store is optional and a snapshot can therefore contain tuples which have been marked as deleted. If reorganization was changed to be mandatory in the HotCold approach, checking for deleted tuples would no longer be necessary and the runtime would be in the same ballpark as it is on the other stores.

When comparing the relative difference in speed of execution between the *min(4b)* query – which loads a 4 byte int value per tuple – and the *min(50b)* query – which loads 50 bytes of data per tuple – we can observe that the dominating factor in query execution is loading data. Differences between the VM-Fork and other backends are caused by added indirection in case of the HotCold approach or bigger tuple size because of added metadata (e.g. the *ShadowPtr*) resulting in higher memory pressure in the other approaches.

Both *min*-queries were run on memory backed by 4kb as well as 2mb pages. Large pages reduce the number of TLB misses since lookup information of larger chunks of memory can be resolved using the entries inside the TLB. With table scans, no significant improvement can be observed. We sampled the number of TLB misses that occur during scan operations both with 4kb as well as with 2mb pages[1]. In both scenarios, the number of TLB misses is zero or close to zero suggesting that TLB misses have no impact on scan operations since misses are rare to begin with. It is assumed that the low number of TLB misses is due to predictive address resolution done by the CPU because of the sequential access pattern of scan queries.

[1]Samples were taken with *oprofile* [10] which periodically acccesses CPU performance counters during execution.

5.3 OLTP&OLAP CH-BenCHmark

To be able to measure the performance of a hybrid system running OLTP transactions as well as OLAP queries in parallel, the CH-BenCHmark was developed [4, 2]. The benchmark extends the TPC-C schema so that TPC-C transactions as well as queries semantically equivalent to TPC-H queries can be executed on the same database state.

For the purpose of measuring the memory overhead incurred by different granularities in the tested snapshotting techniques, we extended the transactional part of the benchmark to include a transaction implementing warranty and return cases. This changes the access pattern of the TPC-C on the *orderline* relation so that a small number of older tuples (2% on average) is updated even after delivery.

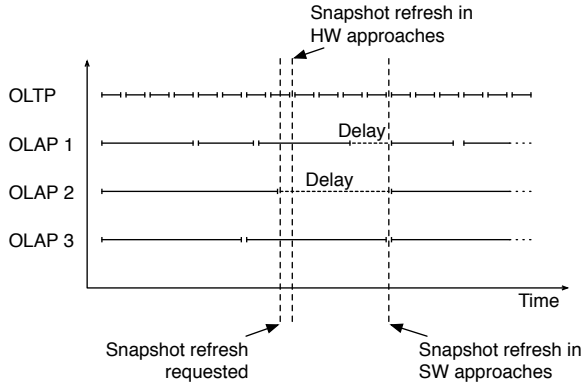


Figure 10: Schematic representation of the CH-BenCHmark used to evaluate all storage backends.

The benchmark was run with one thread executing OLTP transactions and 3 threads concurrently running OLAP queries (specifically, queries 1 and 5 of the TPC-H) on a snapshot. A snapshot refresh was triggered every 200,000 OLTP transactions. A schematic representation of the benchmark is shown in Figure 10. There, the difference between snapshot refresh delays between Hardware Page Shadowing and software controlled snapshotting mechanisms is displayed. When a hardware supported technique is used, only OLTP execution has to be quiesced. With software controlled mechanisms – like Tuple Shadowing – all threads executing OLAP queries have to stopped as well which reduces throughput because queries have to be either aborted or delayed.

When the snapshot renewal is requested and the delaying strategy is employed, no new OLAP queries are admitted in software controlled snapshotting techniques. As soon as all existing OLAP queries have finished, OLTP processing is quiesced and a the snapshot is refreshed. All OLAP threads finished with their query inhibit a delay until the new snapshot is ready, thus reducing OLAP throughput. When a hardware controlled snapshotting mechanism is used, the snapshot can be renewed as soon as all OLTP transactions have been quiesced. Here, the coexistence of multiple snapshots possible in the hardware-controlled mechanisms VM-

Fork and HotCold is beneficial, as the creation of a new snapshot is independent of parallel query execution on old snapshots.

5.3.1 OLTP/OLAP throughput

Table 3 shows the throughput for OLTP transactions as well as OLAP queries. The OLTP transactions correspond to the transactions of the TPC-C. The OLAP queries consist of queries semantically equivalent to queries 1 and 5 of the TPC-H. The two representative OLAP queries are repeatedly executed in an alternating pattern.

Backend	raw	index fork		index copy		index share	
	OLTP	OLTP	OLAP	OLTP	OLAP	OLTP	OLAP
VM-Fork	85k	60k	10.3	59k	9.7	n/a	n/a
Tuple	25k	22k	6.8	19k	5.9	24k	5.9
Twin	33k	29k	7.0	26k	5.9	27k	7.0
HotCold	84k	59k	9.6	59k	9.9	n/a	n/a

Table 3: OLTP and OLAP throughput per second in the CH-BenCHmark.

Looking at OLTP throughput, it can be observed that techniques based on Hardware Page Shadowing yield higher throughput. This has two major reasons: First, Hardware Page Shadowing allows for faster reorganization than software controlled mechanisms as we observed in Section 5.1. Second, there is no indirection as opposed to Tuple Shadowing where a shadow tuple has to be checked, or Twin Tuples where the tuple to be read or written has to be found using a bit flag (c.f. Appendix 4.2).

OLAP query performance is influenced less by the choice of snapshotting mechanism. Compared to a 50% slowdown as seen in OLTP throughput, OLAP queries run about 25% slower when a software controlled snapshotting mechanism is employed. Here, the slowdown is caused by two main factors: Reorganization time and the delay caused by quiescing OLAP queries. All backends have been architected so that OLAP query performance is as high as possible. This is achieved by maintaining tuples included in the snapshot in their original form and position and adding redirection only for new, updated or deleted tuples which can only be seen by OLTP transactions, not OLAP queries.

Throughput for both OLTP transactions as well as OLAP queries varies with different index synchronization mechanisms. For index copy, performance degradation is caused by an increase in reorganization delay of about 1 second per 1000 megabytes index size. When indexes are shared between transactions and queries, reorganization time is unaffected but instead a runtime overhead for acquiring latches is incurred. For index fork – where indexes are shadow copied with vm page granularity – the decrease in OLTP throughput compared to the raw throughput given in Table 3 is the result of both increased fork time as well as runtime overhead. Here, the part of the page table used for index pages needs to be copied as part of the fork causing a small delay in the order of milliseconds per gigabyte index size. Additionally, more significant delays occur whenever a physical page has to be copied causing a significant performance decrease compared to the raw OLTP throughput given. It should be noted that the raw values shown in Table 3 were measured without any index synchronization causing all indexes to be inaccessible for OLAP query processing.

The benchmark was executed on both 4kb and 2mb pages. Techniques involving the `fork` system call experience better fork performance with larger pages. This is due to the fact that the page table – which is copied eagerly on snapshot refresh – is 512 times smaller when using 2mb pages instead of 4kb pages (see 5.1). Apart from performance gains related to smaller delays caused by higher fork performance, a measurable performance gain from bigger pages is experienced by OLTP transactions. There, the memory access pattern is non sequential – as opposed to OLAP table scans. On most architectures (see, for instance, [7]), the size of virtual memory for which address resolution can be performed in hardware using the TLB is significantly larger when using large pages than when small pages are used. Therefore, TLB misses occur less frequently thus increasing transaction throughput. In measurements we conducted for a TPC-C workload, the number of TLB misses was reduced to about 50% when using large pages as opposed to small pages.

Absolute gains in OLTP performance are higher for approaches with a higher inclination to TLB misses. This is the case for both the HotCold and the Tuple Shadowing approach. Since shadow copies can not be easily located close to the original tuple without high memory consumption overhead or diminished OLTP performance, opportunities for TLB misses during OLTP transactions effectively double. Therefore, reducing the number of TLB misses by roughly 50% results in higher absolute savings compared to, for example, Twin Tuples.

OLAP processing does not significantly profit from using large pages. As noted before, any gains from using large pages are either due to shorter reorganization time or less TLB misses. The effects of shorter reorganization time when the `fork` system call is used are insignificant since the entire delay caused by this operation does only account for about 1% of the total runtime of the benchmark. A throughput increase due to a lower TLB miss rate as observed for OLTP transactions can not be observed for queries because the access pattern of OLAP queries is sequential, making prefetching possible. Therefore, the TLB miss rate in OLAP queries can be assumed to be low and thus no significant performance increase can be expected from reducing TLB misses.

5.3.2 Memory consumption

For the same CH-BenCHmark configuration as used in Section 5.3.1, we measured the total memory usage. Figure 11 shows the absolute memory used by our prototypes after executing a given number of OLTP transactions. Memory measurements were taken by monitoring total memory consumption on the test hardware.

It can be observed that the memory curve of all approaches is approximately linear. As the insertion-/update-rate of the CH-BenCHmark is constant, this course was to be expected. The fluctuations in memory consumption result from the parallel execution of OLAP queries which require a certain amount of memory for computations and intermediate results. With no OLAP processing in parallel, the deviation from a linear shape of the curve is no longer visible given the scale used in Figure 11.

The figure includes the four approaches discussed in this paper as well as curves labeled 'baseline' and 'Row w/o OLAP'. 'Baseline' is the minimum amount of memory required to save the tuples without applying compression, its value is calculated. 'Row w/o OLAP' is the memory required by a row store implementation without any snapshot

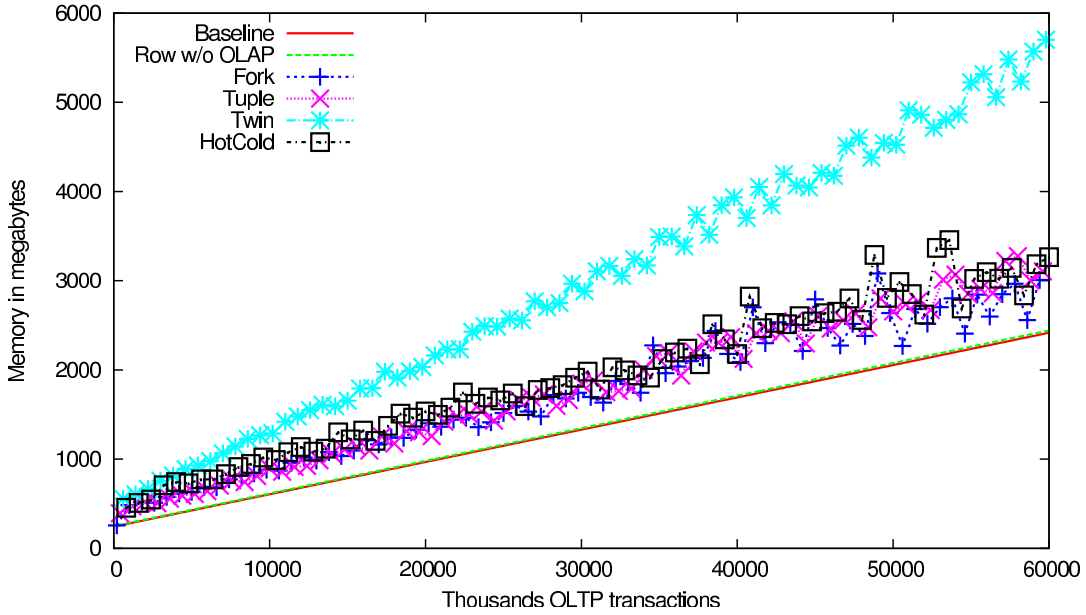


Figure 11: Memory consumed by the different snapshotting techniques over the course of a 6 million OLTP transaction run of the CH-BenCHmark.

mechanism and was measured the same way memory consumption was measured for our snapshot approaches.

The VM-Fork, Tuple Shadowing and HotCold approach each consume roughly equivalent amounts of memory during the benchmark. Compared to baseline memory consumption, the three approaches require roughly 20 to 30% more memory than what is necessary to save the raw data contained in all tuples. The Twin Tuples approach requires about twice as much memory compared to the baseline, which is caused by constantly saving every tuple twice.

The difference in memory consumption between 'Row w/o OLAP' and the approaches researched in this paper can be explained by multiple factors: First, shadow copies consume memory that would not be needed when updating in place. Second, parallel OLAP processing requires memory for intermediate results. Third, more metadata like bitmaps or page table copies need to be kept in memory.

In Figure 11, no clear savings from approaches with finer shadowing granularity can be observed. We believe this is caused by high locality of the TPC-C benchmark as well as small tuple size of those tables which are actually updated.

6. CONCLUSION

Satisfying the emerging requirement for real-time business intelligence demands to execute a mixed OLTP&OLAP workload on the same database system state. In this paper, we analyzed 4 different snapshotting techniques for in-memory DBMS that allow to shield mission-critical OLTP from the longer-running OLAP queries without any additional concurrency control overhead: VM-fork that creates the snapshot by cloning the virtual memory of the database process, Twin Tuples that keeps two copies of each tuple, software-controlled Tuple Shadowing and the HotCold adaptation of the VM-fork. The clear winner in terms of OLTP performance, OLAP query response times and memory consumption is the VM-fork technique which exploits modern multi-core architectures effectively as it allows to create an arbitrary number of time-wise overlapping snapshots with parallel query sessions. The snapshot maintenance is completely delegated to the MMU&OS as they detect and perform the necessary page replications (*copy-on-write*) ultra-efficiently. Thus, the re-emergence of in-memory databases and the progress in hardware supported virtual memory management have led to a promising reincarnation of the shadow paging of the early database days. Unlike the original shadow page snapshots, the hardware controlled VM snapshots are very well suited for processing OLAP queries in a mixed OLTP&OLAP workload.

7. REFERENCES

[1] T. Cao, M. V. Salles, B. Sowell, Y. Yue, J. Gehrke, A. Demers, and W. White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *SIGMOD*, 2011.

[2] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompaß, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest*, 2011.

[3] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[4] F. Funke, A. Kemper, and T. Neumann. Benchmarking Hybrid OLTP&OLAP Database Systems. In *BTW*, 2011.

[5] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[6] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992,

2008.

[7] Intel. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf, 2008.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. 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. *PVLDB*, 1(2):1496–1499, 2008.

[9] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[10] J. Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

[11] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Trans. Database Syst.*, 2(1):91–104, 1977.

[12] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[13] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[14] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *ICDE*, pages 452–462, 1989.

[15] VoltDB LLC. VoltDB Technical Overview. http://voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf, 2010.

APPENDIX

A. HARD- AND SOFTWARE USED

All tests were conducted on a database server with two Intel Xeon X5570 quad-core CPUs at 2.93GHz. The machine is equipped with 64 gigabytes of main memory and did not swap during any of the tests. All code is written in C++ and compiled with g++ version 4.4 using `-O3` optimization settings. Transactions as well as queries are compiled to machine code using the LLVM 2.8 compiler infrastructure with default optimization passes; see [12] for more information on query compilation.

B. WORDLOAD USED

During the run of the CH-Benchmark, OLAP queries 1 and 5 were execute in an alternating sequence. The queries roughly mimic queries 1 and 5 from the TPC-H benchmark. The schema of the CH-Benchmark which is a combination of the schemas of TPC-C and TPC-H is shown in Figure 12. Further information can be found in the CH-Benchmark paper [2].

B.1 Query 1

```
select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
       count(*) as count_order
from orderline
where ol_delivery_d >
    timestamp '2010-01-01␣00:00:00'
group by ol_number order by     ol_number
```

B.2 Query 5

```
select n_name, sum(ol_amount) as revenue
from customer, order, orderline,
       stock, supplier, nation, region
where
       c_id = o_c_id
       and c_w_id = o_w_id
       and c_d_id = o_d_id
       and ol_o_id = o_id
       and ol_w_id = o_w_id
       and ol_d_id=o_d_id
       and ol_w_id = s_w_id
       and ol_i_id = s_i_id
  -- instead of FK :
       and mod((s_w_id * s_i_id),10000) =
         su_suppkey
       and ascii(substr(c_state,1,1)) =
         su_nationkey
       and su_nationkey = n_nationkey
       and n_regionkey = r_regionkey
       and r_name = 'Europa'
       and o_entry_d >=
         timestamp '2010-01-01␣00:00:00'
group by
      n_name
order by
      revenue desc
```

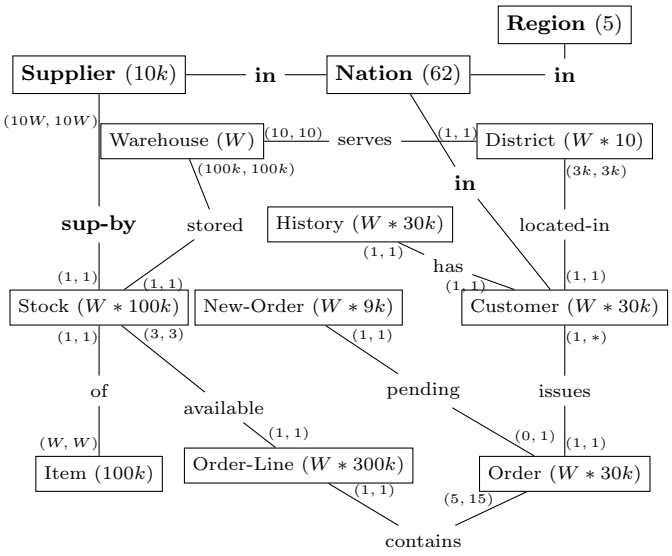


Figure 12: The schema of the CH-Benchmark.

B.3 OLTP Transactions

- New-Order
- Payment
- Delivery
- Order-Status
- Stock-Level

Towards Highly Parallel Event Processing through Reconfigurable Hardware

Mohammad Sadoghi, Harsh Singh, Hans-Arno Jacobsen
Middleware Systems Research Group
Department of Electrical and Computer Engineering
University of Toronto, Canada

ABSTRACT

We present *fpga-ToPSS* (Toronto Publish/Subscribe System), an efficient event processing platform to support high-frequency and low-latency event matching. *fpga-ToPSS* is built over reconfigurable hardware—FPGAs—to achieve line-rate processing by exploring various degrees of parallelism. Furthermore, each of our proposed FPGA-based designs is geared towards a unique application requirement, such as flexibility, adaptability, scalability, or pure performance, such that each solution is specifically optimized to attain a high level of parallelism. Therefore, each solution is formulated as a design trade-off between the degree of parallelism versus the desired application requirement. Moreover, our event processing engine supports Boolean expression matching with an expressive predicate language applicable to a wide range of applications including real-time data analysis, algorithmic trading, targeted advertisement, and (complex) event processing.

1. INTRODUCTION

Efficient event processing is an integral part of growing number of data management technologies such as real-time data analysis [27, 5, 29], algorithmic trading [26], intrusion detection system [5, 8], location-based services [30], targeted advertisements [9, 25], and (complex) event processing [1, 7, 2, 6, 17, 3, 24, 25, 9].

A prominent application for event processing is algorithmic trading; a computer-based approach to execute buy and sell orders on financial instruments such as securities. Financial brokers exercise investment strategies (*subscriptions*) using autonomous high-frequency algorithmic trading fueled by real-time market *events*. Algorithmic trading is dominating financial markets and now accounts for over 70% of all trading in equities [11]. Therefore, as the computer-based trading race among major brokerage firms continues, it is crucial to optimize execution of buy or sell orders at the microsecond level in response to market events, such as corporate news, recent stock price patterns, and fluctuations in currency exchange rates, because every microsecond translates into opportunities and ultimately profit [11]. For instance, a simple classical arbitrage strategy has an estimated annual profit of over $21 billion according to TABB Group [12]. Moreover, every 1-millisecond reduction in response-time is estimated to generate the staggering amount of over $100 million a year [19]; such requirements greatly increases the burden placed on event processing platform.

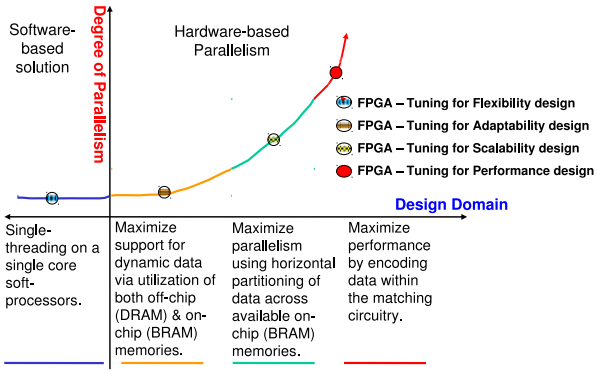


Figure 1: Degrees of Offered Parallelism

Therefore, a scalable event processing platform must efficiently determine all subscriptions that match incoming events at a high rate, potentially up to a million events per second [4]. Similar requirements are reported for event processing in network monitoring services [27].

To achieve throughput at this scale, we propose and evaluate a number of novel FPGA-based event processing designs (Field Programmable Gate Array). An FPGA is an integrated circuit designed to be reconfigurable to support custom-built applications in hardware. Potential application-level parallelism can be directly mapped to purpose-built processing units operating in parallel. Configuration is done through encoding the application in a programming language-style description language and synthesising a configuration uploaded on the FPGA chip [14]. FPGA-based solutions are increasingly being explored for data management tasks [20, 21, 22, 28, 26].

This promising outlook has a few caveats that make the acceleration of any data processing with FPGAs a challenging undertaking. First, current FPGAs (e.g., 800MHz Xilinx Virtex 6) are still much slower compared to commodity CPUs (e.g. 3.2 GHz Intel Core i7). Second, the accelerated application functionality has to be amenable to parallel processing. Third, the on-/off-chip data rates must keep up with chip processing speeds to realize a speedup by keeping the custom-built processing pipeline busy. Finally, FPGAs restrict the designer's flexibility and the application's dynamism[1], both of which are hardly a concern in standard software solutions. However, the true success of FPGAs is rooted in three distinctive features: hardware parallelism, hardware reconfigurability, and substantially higher throughput rates.

[1]e.g., subscription insert and delete operations are not a given.

DaMoN'11, June 13, 2011, Athens, Greece.

Thus, each of our solutions is formulated as a design trade-off between the degree of exploitable parallelism (cf. Fig. 1) versus the desired application-level requirements. Requirements considered are: the ease of the development and deployment cycle (*flexibility*), the ability of updating a large subscription workload in real-time (*adaptability*), the power of obtaining a remarkable degree of parallelism through horizontal data partitioning on a moderately sized subscription workload (*scalability*), and, finally, the power of achieving the highest level of throughput by eliminating the use of memory and by specialized encoding of subscriptions on FGPA (*performance*).

We experiment with four novel system designs that exhibit different degrees of parallelism (cf. Fig. 1) and capture different application requirements. In our application context, achievable performance is driven by the degree of parallelism (in which FPGAs dominate) and the chip operating frequency (in which CPUs dominate). Therefore, our solution design space is as follows: a single thread running on single CPU core (PC), a single thread on a single soft-processor (*flexibility*) [2], up to four custom hardware matching units (MUs) running in parallel in which the limiting factor is off-chip memory bandwidth (*adaptability*), horizontally partitioning data across m matching units running in parallel in which the limiting factor is the chip resources and the on-chip memory (*scalability*), and, lastly, n (where $n \geq m$) matching units running in parallel (with no memory access because the data is also encoded on the chip), in which the limiting factor is the amount of chip resources, particularly, the required amount of wires (*performance*).

The ability of an FPGA to be re-configured on-demand into a custom hardware circuit with a high degree of parallelism is key to its advantage over commodity CPUs for data and event processing. Using a powerful multi-core CPU system does not necessarily increase processing rate (Amdahl's Law) as it increases inter-processor signaling and message passing overhead, often requiring complex concurrency management techniques at the program and OS level. In contrast, FPGAs allow us to get around these limitations due to their intrinsic highly inter-connected architecture and the ability to create custom logic on the fly to perform parallel tasks. In our design, we exploit parallelism, owing to the nature of the matching algorithm (Sec. 3), by creating multiple matching units which work in parallel with multi-giga bit throughput rates (Sec. 4), and we utilize reconfigurability by seamlessly adapting relevant components as subscriptions evolve (Sec. 4).

2. RELATED WORK

FPGA An FPGA is a semiconductor device with programmable lookup-tables (*LUTs*) that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6 typically). FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (a few kilobits), that together provide small storage capacity but a large bandwidth for circuits in the FPGA. Thousands of these building blocks are connected with a programmable interconnect to implement larger-scale circuits.

Past work has shown that FPGAs are a viable solution for building custom accelerated components [20, 21, 22, 28, 26]. For instance, [20] demonstrates a design for accelerated XML processing, and [21] shows an FPGA solution for processing market feed data. As opposed to these approaches, our work concentrates on supporting a more general event processing platform specifically designed to accelerate the event matching computation. Similarly, [22] presents a data stream processing framework that uses FPGAs to efficiently run hardware-encoded queries (without join). Lastly, on the path toward supporting stream processing, a novel framework on how to efficiently run hardware-encoded regular expression queries in parallel on an FPGA is proposed [28]; a key insight was the realization that the deterministic finite automata (DFA), although suitable for software solutions, results in explosion of space; thereby, to bound the required space on the FPGA, a non-deterministic finite automata (NFA) is utilized. Our approach differs from [28] as we primarily focus on supporting large scale conjunctive Boolean queries.

On a different front, a recent body of work has emerged that investigates the use of new hardware architectures for data management systems [10]; for instance, multi-core architectures were utilized to improve the performance of database storage manager [13] and to enhance the transaction and query processing [23].

Finally, the sketch of our initial proposal was presented in [26]; our current work not only delves into much more technical depth which was omitted from [26], it also reformulates our FPGA-based solutions based on the extent of parallelism in each design, a formulation that permits a concrete performance comparison of various designs accompanied by a comprehensive experimental analysis. Most importantly, this work introduces key insights and novel system designs through introducing an effective horizontal data partitioning to achieve an unprecedented degrees of parallelism on moderate-size subscription workload.

Matching The matching is one of the main computation intensive components of event processing which has been well studied over the past decade (e.g., [1, 7, 2, 6, 17, 3, 9, 24, 25]). In general, the matching algorithms are classified as (1) counting-based [7], and (2) tree-based [1, 25]. The counting algorithm is based on the observation that subscriptions tend to share many common predicates; thus, the counting method minimizes the number of predicate evaluations by constructing an inverted index over all unique predicates. Similarly, the tree-based methods are designed to reduce predicate evaluations; in addition, they recursively cut through space and eliminate subscriptions on the first encounter with an unsatisfiable predicate. The counting- and tree-based approaches can be further classified as either key-based (in which for each subscription a set of predicates are chosen as identifiers [7]), or as non-key method [1]. In general, the key-based methods reduce memory access, improve memory locality, and increase parallelism, which are essentials for a hardware implementation. One of the most prominent counting-based matching algorithms are Propagation [7], a key-based method while one of the most prominent tree-based approach, BE-Tree, which is also a key-based method [25].

3. EVENT PROCESSING MODEL

Subscription Language & Semantics The matching algorithm takes as input an event (e.g., market event and user profile) and a set of subscriptions (e.g., investment strategies and targeted advertisement constraints) and returns matching subscriptions. The event is modeled as a value assignment to attributes and the subscription is modeled as a Boolean expression (i.e., as conjunction of Boolean predicates). Each Boolean predicate is a triple of either [attribute$_i$, operator, values] or [attribute$_i$, operator, attribute$_j$]. Formally, the matching problem is defined as follows: *given an event* e *and a set of subscriptions* $\mathbf{s}$*, find all subscriptions* $s_i \in \mathbf{s}$ *satisfied by* e.

Matching Algorithm The Propagation algorithm is a state-of-the-art key-based counting method that operates as follows [7]. First, each subscriptions is assigned a key (a set of predicates) based on which the typical counting-based inverted index is replaced by a

[2] A soft-processor is a processor encoded like an application running on the FPGA. It supports compiled code written in a higher-level language, like for example C without operating system overhead.

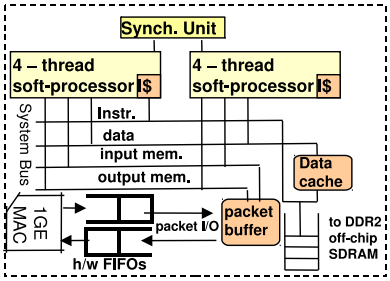


Figure 2: Tunning for Flexibility Design

set of multi-attribute hashing schemes. The multi-attribute hashing scheme uniquely assigns subscriptions into a set of disjoint clusters. Second, keys are selected from a candidate pool using a novel cost-based optimization tuned by the workload distribution to minimize the matching cost [7]. The Propagation data structure has three main strengths which makes it an ideal candidate for a hardware-based implementation: (1) subscriptions are distributed into a set of disjoint clusters which enables highly parallelizable event matching through many specialized custom hardware matching units (MUs), (2) within each cluster, subscriptions are stored as contiguous blocks of memory which enables fast sequential access and improves memory locality, and (3) the subscriptions are arranged according to their number of predicates which enables prefetching and reduces memory accesses and cache misses [7].

4. FPGA-BASED EVENT PROCESSING

Commodity servers are not quite capable of processing event data at line-rate. The alternative is to acquire and maintain high cost purpose-built event processing applications. In contrast, our design uses an FPGA to significantly speed up event processing computations involving event matching. FPGAs offer a cost effective event processing solutions, since custom hardware can be altered and scaled to adapt to the prevailing load and throughput demands. Hardware reconfigurability allows FPGAs to house *soft-processors*—processors composed of programmable logic. A soft-processor has several advantages: it is easier to program on it (e.g., using `C` as opposed to `Verilog` which requires specialized knowledge and hardware development tools), it is portable to different FPGAs, it can be customized, and it can be used to communicate with other components and accelerators in the design. In this project, the FPGA resides on a NetFPGA [18] network interface card and communicates through DMA on a PCI interface to a host computer. FPGAs have programmable I/O pins that in our case provide a direct connection to memory banks and to the network interfaces, which in a typical server, are only accessible through a network interface card.

In this section, we describe our four implemented designs each of which is optimized for a particular characteristic such as flexibility in development and deployment process, adaptability in supporting changes for a large workload size, scalability through horizontal data partitioning for moderate workload size, and performance in maximizing throughput for small workload size. Most notably, the distinguishing feature among our proposed designs is the level of parallelism that ranges from running all subscriptions on a single processor (*flexibility*) to running every subscription on its own custom hardware unit (*performance*).

4.1 Tuning for Flexibility

Our first approach is the soft-processor(s)-based solution (cf. Fig. 2), which runs on a soft-processor that is implemented on the NetFPGA platform. This solution also runs the same C-based event matching code that is run on the PC-based version (our baseline);

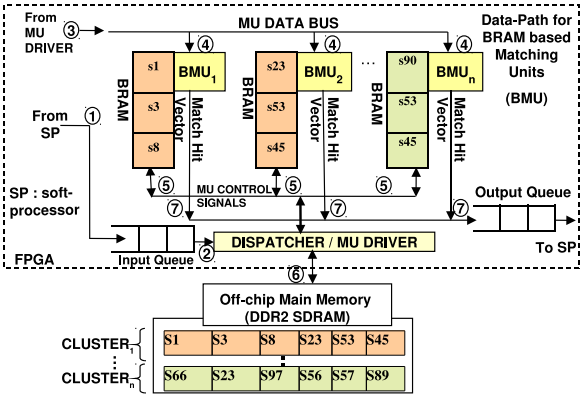


Figure 3: Tuning for Adaptability Design

thus, this design is the easiest to evolve as message formats and protocols change. In order to maximize throughput of our event processing application, we chose NetThreads [15] as the baseline soft-processor platform for the FPGA. NetThreads has two single-issue, in-order, 5-stage, 4-way multi-threaded processors (cf. Fig. 2), shown to deliver more throughput than simpler soft-processors [16]. In a single core, instructions from four hardware threads are issued in a round-robin fashion to hide stalls in the processor pipeline and execute computations even when waiting for memory. Such a soft-processor system is particularly well-suited for event processing: The soft-processors suffer no operating system overhead compared to conventional computers, they can receive and process packets in parallel with minimal computation cost, and they have access to a high-resolution system clock (much higher than a PC) to manage timeouts and scheduling operations. One benefit of not having an operating system in NetThreads is that packets appear as character buffers in a low latency memory and are available immediately after being fully received by the soft-processor (rather than being copied to a user-space application). Also, editing the source and destination IP addresses only requires changing a few memory locations, rather than having to comply with the operating system's internal routing mechanisms. Because a simpler soft-processor usually executes one instruction per cycle, it suffers from a raw performance drawback compared to custom logic circuits on FPGAs; a custom circuit can execute many operations in parallel as discussed next.

4.2 Tuning for Adaptability

In order to utilize both hardware-acceleration while supporting large dynamic subscriptions on both off-chip and on-chip memories, we propose a second scheme (cf. Fig. 3). Since FPGAs are normally programmed in a low-level hardware-description language, it would be complex to support a flexible communication protocol. Instead, we instantiate a soft-processor (SP) to implement the packet handling in software. After parsing incoming event data packets, the soft-processor offloads the bulk of the event matching to a dedicated custom hardware matching unit. Unlike the subscription-encoded matching units used in the tuned for *performance* design, these matching units use low-latency on-chip memories, Block RAMs (BRAMs) available on FPGAs, that can be stitched together to form larger dedicated blocks of memory. The FPGA on the NetFPGA platform [18] has 232 18kbit BRAMs which are partially utilized to cache a subset of subscriptions. Having an on-chip subscription data cache allows event matching to be initiated even before the off-chip subscription data can be accessed. However, our matching algorithm leverages data locality in the storage of dynamic subscriptions, which may be updated during run time, in contiguous array clusters thereby exploiting burst-oriented

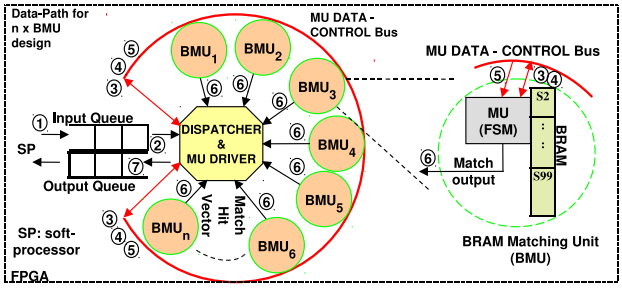


Figure 4: Tuning for Scalability Design

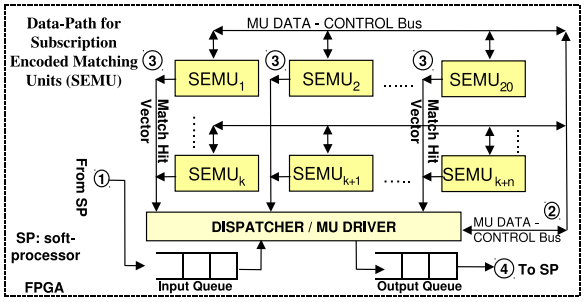


Figure 5: Tuning for Performance Design

data access feature of the DDR2 (or SDRAM), off-chip memory, while fetching the subscription data clusters. Thus, having an on-chip subscription cache partially masks the performance penalty (latency) of fetching the subscriptions from the off-chip DDR2 memory, which is the main throughput bottleneck of our FPGA-based event processing solution. Therefore, the maximum amount of useful parallelism in this design is limited by the memory bandwidth; in particular, no more than four custom hardware matching units can be sustained simultaneously; any additional matching units will remain idle because only a limited amount of data can be transfered from off-chip memory to on-chip memory in each clock cycle.

In our design tuned for *adaptability*, we employ a more generalized design that enables the matching units to support a dynamic and a larger subscription workload than can be supported in our designs that tuned either for *scalability* or *performance*. Our *adaptability* design employs the BRAM-based Matching Units (BMUs) which allows a subset of subscriptions to be stored on the on-chip dedicated low latency BRAMs; thus making the design less hardware resource intensive compared to the our pure hardware implementation (tuned for *performance* design). Furthermore, coalescing dynamic subscription data into an off-chip memory image is achieved using the Propagation algorithm. The resulting subscription data image is downloaded to the off-chip main memory (e.g DDR2 SDRAM) while loading FPGA configuration bitstream. Nevertheless, any hardware performance advantage promised by a FPGA-based design soon dwindles when the data must be accessed from an off-chip memory. We adopt two approaches to reduce the impact of off-chip memory data access latency on the overall system throughput. Firstly, we take advantage of high degree of the data locality inherent in Propagation's data structure which helps to minimize random access latency. Secondly, to achieve locality subscriptions are grouped into non-overlapping clusters using attribute-value pair as access keys. Therefore, this data structure is optimized for storing large number of subscriptions in off-chip memory. In addition, we incorporate a fast (single cycle latency) but smaller capacity BRAMs for each matching unit to store subset of subscriptions, which helps mask the initial handshaking setup delay associated with off-chip main memory access, i.e., the event matching can begin against these subscriptions as soon as the event arrives; in the meantime the system prepares to setup data access from the off-chip DDR2 main memory.

The stepwise operation of this design is depicted in Fig. 3. Upon arrival of an event, the SP transfers (1) the data packets to the input queue of the system. A custom hardware submodule, the DISPATCHER unit, extracts subscription predicates-value pairs, which are input to hash functions to generate cluster addresses. Cluster addresses are used to look-up the memory locations (2) of the relevant subscription clusters residing both in BMU BRAMs and in off-chip main memory. The DISPATCHER then feeds the event (3) and previously computed cluster addresses (4) on the MU DATA BUS (common to all BMUs). Next, the MU DRIVER unit activates all parallel BMUs to initiate matching (5) using on-chip static subscriptions stored in each BMU, while simultaneously queuing up read requests for the off-chip main memory. The transfer (6) of dynamic subscription data between the BMUs is pipelined to avoid stalling the matching units due to data exhaustion. Finally the match results are pushed (7) into the output queue from which the SP transfers the results to the network interface to be sent to the intended host(s).

4.3 Tuning for Scalability

The key property of our proposed design tuned for *scalability* is the horizontal data partitioning that maximizes parallelism (cf. Fig. 4). This design offers the ability to adjust the required level of parallelism (which directly translates into matching throughput) by adjusting the degree of data partitioning for a moderate size workload, yet without significantly compromising the feature offered in our *adaptability* design. It achieves this by fully leveraging the available on-chip (BRAM) memory to partition the global Propagation's data structure across BRAM blocks such that each subset of BRAMs is dedicated to each matching unit, in which the matching unit has an exclusive access to a chunk of the global Propagation's structure. Unlike our *adaptability* design in which the degree of parallelism is quite restricted due to the off-chip memory's access latency, resulting in several data starved or stalled matching units, this design employs matching units (BMUs) (cf. Fig. 4) that are each provisioned with a dedicated BRAM memory in order to keep them fully supplied with subscription data. Therefore, the degree of parallelism achieved is simply a function of the number of BMUs that can be supported by the underlying FPGA. Finally, a non-performance critical soft-processor (SP) can be employed to update the on-chip memory tables attached to each BMU in the design; hence, supporting dynamic subscription workload.

The overall stepwise operation of our tuned for *scalability* design, depicted in Fig. 4, is similar to that which occurs in the tuned for *adaptability* design for steps (1) to (5), with the difference being in the absence of the off-chip main memory used for storing the dynamic subscriptions. The operation and logic of the DISPATCHER and MU DRIVER submodule is further simplified as the off-chip memory access arbitration and data dissemination to BMUs is eliminated. Every BMU consists of a four-state Finite State Machine, that upon receiving the event data (3) initiates matching by sequentially fetching one subscriptions every clock cycle from the dedicated BRAM memory containing the cluster starting at the address (4) that was dispensed by the DISPATCHER unit. Since all BMUs are ran in parallel and in sync with each other, the DISPATCHER must dispense the next cluster address only when all BMUs have completed matching all subscriptions in the current cluster. In final phase (6), once all BMUs finish matching all the subscriptions' clusters corresponding to the predicates present in the incoming event, the final result tallying phase is initiated where matched subscriptions or number of matches found are placed on the match hit

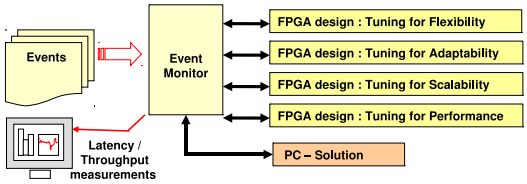


Figure 6: Evaluation Testbed

vectors and consolidated as a final result value by the DISPATCHER unit to be transfered to SP via the output queue.

4.4 Tuning for Performance

Our final approach (cf. Fig. 5) is a purely hardware solution: custom hardware components perform necessary steps involving event parsing and matching of event data against subscriptions. This method provides near line-rate performance, but also involves a higher level of complexity in integrating custom heterogeneous accelerators in which both the performance-critical portion of the event processing algorithm and the encoding of subscriptions are incorporated within the design of the matching unit logic; thereby, completely eliminating all on- and off-chip memory access latencies. Essentially, each subscription is transformed into a self-contained custom hardware unit; this subscription encoding achieves the highest level of parallelism in our design space because all subscriptions are ran in parallel.

The *performance* design offers the highest rate at which incoming events can be matched against subscriptions, which are encoded in the SUBSCRIPTION ENCODED MATCHING UNIT (SEMU) logic on the FPGA. This method avoids the latency of both on and off-chip memory access, but significantly constraints the size of the subscription base that can be supported. A diagram of this design is shown in Fig. 5. This setup is massively parallelized and offers event matching at extremely high rates (i.e. one per clock cycle).

The stepwise operation of the our tuned for *performance* design is depicted in Fig. 5. In this design, the soft-processor (SP) only serves to transfer (1) the received event data packets from the network interface input buffer to the input queue of the our system. Custom hardware submodule, the DISPATCHER module, parses (2) the incoming events and feeds the current event data to all the matching units while the MU DRIVER module generates all the necessary control signals to run all SEMUS synchronously. Each unit is able to match all encoded subscriptions against the current event in one clock cycle. However, subsequent clock cycles are spent in tallying the matches and preparing the final responses (e.g. forward address look-up or consolidating system wide match counts) that is eventually pushed (3) into the output queue. The SP then transfers (4) the final result from the output queue to the network interface to be sent to the intended host(s).

5. EXPERIMENTAL RESULTS

This section describes our evaluation setup including the hardware used to implement our FPGA-based event processing system and the measurement infrastructure.

Evaluation Platform Our FPGA based solutions are instantiated on the NetFPGA 2.1 [18] platform, operating at 125MHz and have access to four 1GigE Media Access Controllers (MACs) via high-speed hardware FIFO queues (cf. Fig. 2) allowing a theoretical 8Gbps of concurrently incoming and outgoing traffic capacity. In addition, a memory controller to access the 64 Mbytes of on-board off-chip DDR2 SDRAM is added. The system is synthesized to meet timing constraints with the Xilinx ISE 10.1.03 tool and targets a Virtex II Pro 50 (speed grade 7ns). Our soft-processor and matching units run at the frequency of the Ethernet MACs (125MHz).

	1x MU	4x MUs	32x MUs	128x MUs
250	7.5	5.5	5.0	3.6
1K	9.3	6.1	4.3	4.3
10K	64.0	19.0	6.8	5.4
50K	223.5	59.9	12.3	7.3

Table 1: Latency (μs) vs. the # of MUs (Scalability Design)

Evaluation & Evaluation Setup For our experiments, we used HP DL320 G5p servers (Quad Xeon 2.13GHz) equipped with an HP NC326i PCIe dual-port gigabit network card running Linux 2.6.26. As shown in Fig. 6, we exercised our event processing solutions from the server executing a modified `Tcpreplay 3.4.0` that sends event packet traces at a programmable fixed rate. Packets are timestamped and routed to either the FPGA-based designs or PC-based design. Each FPGA-based design is configured as one of the solutions described in Sec. 4 and PC-based is a baseline serving as comparison only. The network propagation delays are similar for all designs. Both FPGA-based or PC-based designs forward market events on the same wire as incoming packets which allows the Event Monitor (EM), cf. Fig. 6, to capture both incoming and outgoing packets from these designs. The EM provides a $8ns$ resolution on timestamps and exclusively serves for the measurements.

Evaluation Workload We generate a workload of tens of thousands of subscriptions derived from investment strategies such as arbitrage and buy-and-hold. In particular, we vary the workload size from 250 subscriptions to over 100K subscriptions. In addition, we generate market events using the Financial Information eXchange (FIX) Protocol with FAST encoding[3].

Evaluation Measurements We characterize the system throughput as the maximum sustainable input packet rate obtained through a bisection search: the smallest fixed packet inter-arrival time where the system drops no packets when monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at the given input rate. The latency of our solutions is the interval between the time an event packet leaves the Event Monitor output queue to the time the first forwarded version of the market event is received and is added to the output queue of the Event Monitor.

5.1 FPGA Performance Benefits

Packet Processing Measuring the baseline packet processing latency of both PC and FPGA-based solutions is essential in order to establish a basis for comparison. When processing packets using the PC solution, we measured an average round-trip time of $49\mu s$ with a standard deviation of $17\mu s$. With the NetThreads processor on the FPGA replying, we measure a latency of $5\mu s$ with a standard deviation of $44ns$. Because of the lack of operating system and more deterministic execution, the FPGA-based solution provides a much better bound on the expected packet processing latency; hence, our FPGA-based solution outperformed the PC-based solution in baseline packet processing by orders of magnitude.

Event Processing Before we begin our detailed comparison of various designs, we study the effect of the number of matching units (MUs) on the matching latency for our *scalability* design, Table 1. As expected, as we increase the number of MUs, moving from 1 MU to 128 MUs, the latency is improved significantly especially for the larger subscriptions workload (with chip resources permitting). This improvement is directly proportional to the degree of parallelism obtained by using a larger number of MUs.

In Table 2, we demonstrate the system latency as the subscription workload size changes from 250 to 100K. In summary, even though our FPGA (125MHz Virtex II) is much slower than the latest FPGA

[3] `fixprotocol.org`

	PC	Flexibility	Adaptability	Scalability	Performance
250	53.9	71.0	6.4	3.6	3.2
1K	60.7	199.4	7.5	4.3	N/A
10K	150.0	1,617.8	87.8	5.4	N/A
100K	2,001.2	16,422.8	1,307.3	N/A	N/A

Table 2: End-to-end System Latency (μs)

	PC	Flexibility	Adaptability	Scalability	Performance
250	122,654	14,671	282,142	740,740	1,024,590
1K	66760	5,089	202,500	487,804	N/A
10K	9594	619	11,779	317,460	N/A
100K	511	60	766	N/A	N/A

Table 3: System Throughput (market events/sec)

(800MHz Virtex 6) and significantly slower than our CPU (Quad Xeon 2.13GHz), our deign tuned for *adaptability* is 8x faster than the PC-based solution on workload sizes of less than 1K and continued to improve over the PC solution by up to a factor of two on workload of sizes of 100K. Similarly, the design tuned for *performance*, while currently feasible only for smaller workloads due to lack of resources on the FPGA, is 21.2x faster. Most importantly, our design tuned for *scalability* takes advantage of both of our *adaptability* and *performance* designs by finding the right balance between using the on-chip memory to scale the workload size while using the highly parallel nature of the *performance* design to scale the event processing power. Thus, the *scalability* design is 16.2x faster than our *adaptability* design and is 27.8x faster than the PC design. In addition, a similar trend was also observed for the system throughput experiment as shown in Table 3.

Therefore, the *adaptability* design is limited because of slower off-chip memory bandwidth which greatly hinders the degree of parallelism while the *performance* design is limited because encoding the subscriptions in the logic fabric of the chip consumes much more area than storing them in BRAM or DDR2 providing much denser storage. Finally, contrary to general perspective that software solution cannot be utilized in hardware, the success of our *scalability* design (which adapts a software-based solution) suggests that in order to scale our solution to large subscription workloads, certain software data structures for data placement become a viable solution in conjunction with hardware acceleration and parallelism.

6. CONCLUSIONS & DISCUSSIONS

We observe that event processing is at the core of many data management applications such as real-time network analysis and algorithmic trading. Furthermore, to enable the high-frequency and low-latency requirements of these applications, we presented an efficient event processing platform over reconfigurable hardware that exploits the high degrees of hardware parallelism for achieving line-rate processing. In brief, the success of our *fpga-ToPSS* framework is through the use of reconfigurable hardware (i.e., FPGAs) that enables hardware acceleration using custom logic circuits and elimination of OS layer latency through on-board event processing together with hardware parallelism and novel horizontal data partitioning scheme. As a result, our design tuned for *performance* outperformed the PC-based solution by a factor of 27x on small size subscription sets while our design tuned for *scalability* outperformed the PC-based solution by a factor of 16x even as the workload size was increased; in fact, this gap further widens as workload size increases due an increased opportunity to process a larger amount of data in parallel.

7. REFERENCES

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC'99*.

[2] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. ICDCSW'02.

[3] L. Brenna, A. Demers, J. Gehrke, M. Hong, Ossher, Panda, Riedewald, Thatte, and White. Cayuga: high-performance event processing engine. *SIGMOD'07*.

[4] J. Corrigan. Updated traffic projections. *OPRA, March'07*.

[5] C. Cranor, T. Johnson, and O. Spataschek. Gigascope: a stream database for network applications. In *SIGMOD'03*.

[6] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE'02*.

[7] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for fast pub/sub systems. *SIGMOD'01*.

[8] A. Farroukh, M. Sadoghi, and H.-A. Jacobsen. Towards vulnerability-based intrusion detection with event processing. In *DEBS'11*.

[9] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD'10*.

[10] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD'08*.

[11] K. Heires. Budgeting for latency: If I shave a microsecond, will I see a 10x profit? *Securities Industry, 1/11/10*.

[12] R. Iati. The real story of trading software espionage. *TABB Group Perspective, 10/07/09*.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT'09*.

[14] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.'08*.

[15] M. Labrecque et al. NetThreads: Programming NetFPGA with threaded software. In *NetFPGA Dev. Workshop'09*.

[16] M. Labrecque and J. G. Steffan. Improving pipelined soft processors with multithreading. In *FPL'07*.

[17] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. ICDCS '05.

[18] J. W. Lockwood et al. NetFPGA - an open platform for gigabit-rate network switching and routing. In *MSE'07*.

[19] R. Martin. Wall street's quest to process data at the speed of light. *Information Week, 4/21/07*.

[20] A. Mitra et al. Boosting XML filtering with a scalable FPGA-based architecture. *CIDR'09*.

[21] G. W. Morris et al. FPGA accelerated low-latency market data feed processing. *IEEE 17th HPI'09*.

[22] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *VLDB'09*.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB'10*.

[24] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. GPX-Matcher: a generic boolean predicate-based XPath expression matcher. In *EDBT'11*.

[25] M. Sadoghi and H.-A. Jacobsen. BE-Tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space. In *SIGMOD'11*.

[26] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB '10*.

[27] D. Srivastava, L. Golab, R. Greer, T. Johnson, J. Seidel, V. Shkapenyuk, O. Spatscheck, and J. Yates. Enabling real time data analysis. *PVLDB'10*.

[28] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB'10*.

[29] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD'06*.

[30] Z. Xu and H.-A. Jacobsen. Processing proximity relations in road networks. SIGMOD '10.

Vectorization vs. Compilation in Query Execution

Juliusz Sompolski[1]
VectorWise B.V.
julek@vectorwise.com

Marcin Zukowski
VectorWise B.V.
marcin@vectorwise.com

Peter Boncz[2]
Vrije Universiteit Amsterdam
p.a.boncz@vu.nl

ABSTRACT

Compiling database queries into executable (sub-) programs provides substantial benefits comparing to traditional interpreted execution. Many of these benefits, such as reduced interpretation overhead, better instruction code locality, and providing opportunities to use SIMD instructions, have previously been provided by redesigning query processors to use a *vectorized execution model*. In this paper, we try to shed light on the question of how state-of-the-art compilation strategies relate to vectorized execution for analytical database workloads on modern CPUs. For this purpose, we carefully investigate the behavior of vectorized and compiled strategies inside the Ingres VectorWise database system in three use cases: Project, Select and Hash Join. One of the findings is that compilation should always be combined with block-wise query execution. Another contribution is identifying three cases where "loop-compilation" strategies are inferior to vectorized execution. As such, a careful merging of these two strategies is proposed for optimal performance: either by incorporating vectorized execution principles into compiled query plans or using query compilation to create building blocks for vectorized processing.

1. INTRODUCTION

Database systems provide many useful abstractions such as data independence, ACID properties, and the possibility to pose declarative complex ad-hoc queries over large amounts of data. This flexibility implies that a database server has no advance knowledge of the queries until runtime, which has traditionally led most systems to implement their query evaluators using an interpretation engine. Such an engine evaluates plans consisting of algebraic *operators*, such as Scan, Join, Project, Aggregation and Select. The operators internally include *expressions*, which can be boolean conditions used in Joins and Select, calculations used to introduce new columns in Project, and functions like MIN, MAX and SUM used in Aggregation. Most query interpreters follow the so-called iterator-model (as described in Volcano [5]), in which each operator implements an API that consists of open(), next() and close() methods. Each next() call produces one new tuple, and query evaluation follows a "pull" model in which next() is called recursively to traverse the operator tree from the root downwards, with the result tuples being pulled upwards.

It has been observed that the tuple-at-a-time model leads to interpretation overhead: the situation that much more time is spent in evaluating the query plan than in actually calculating the query result. Additionally, this tuple-at-a-time interpretation model particularly affects high performance features introduced in modern CPUs [13]. For instance, the fact that units of actual work are hidden in the stream of interpreting code and function calls, prevents compilers and modern CPUs from getting the benefits of deep CPU pipelining and SIMD instructions, because for these the work instructions should be adjacent in the instruction stream and independent of each other.

Related Work: Vectorized execution. MonetDB [2] reduced interpretation overhead by using *bulk processing*, where each operator would fully process its input, and only then invoking the next execution stage. This idea has been further improved in the X100 project [1], later evolving into VectorWise, with *vectorized execution*. It is a form of block-oriented query processing [8], where the next() method rather than a single tuple produces a block (typically 100-10000) of tuples. In the vectorized model, data is represented as small single-dimensional arrays (vectors), easily accessible for CPUs. The effect is (i) that the percentage of instructions spent in interpretation logic is reduced by a factor equal to the vector-size, and (ii) that the functions that perform work now typically process an array of values in a tight loop. Such tight loops can be optimized well by compilers, e.g. unrolled when beneficial, and enable compilers to generate SIMD instructions automatically. Modern CPUs also do well on such loops, as function calls are eliminated, branches get more predictable, and out-of-order execution in CPUs often takes multiple loop iterations into execution concurrently, exploiting the deeply pipelined resources of modern CPUs. It was shown that vectorized execution can improve data-intensive (OLAP) queries by a factor 50.

Related Work: Loop-compilation. An alternative strategy for eliminating the ill effects of interpretation is using Just-In-Time (JIT) query compilation. On receiving a query

[1]This work is part of a MSc thesis being written at Vrije Universiteit Amsterdam.
[2]The author also remains affiliated with CWI Amsterdam.

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.

for the first time, the query processor compiles (part of) the query into a routine that gets subsequently executed. In Java engines, this can be done through the generation of new Java classes that are loaded using reflection (and JIT compiled by the virtual machine) [10]. In C or C++, source code text is generated, compiled, dynamically loaded, and executed. System R originally skipped compilation by generating assembly directly, but the non-portability of that approach led to its abandonment [4]. Depending on the compilation strategy, the generated code may either solve the whole query ("holistic" compilation [7]) or only certain performance-critical pieces. Other systems that are known to use compilation are ParAccel [9] and the recently announced Hyper system [6]. We will generalise the current state-of-the-art using the term "loop-compilation" strategies, as these typically try to compile the core of the query into a single loop that iterates over tuples. This can be contrasted with vectorized execution, which decomposes operators in multiple basic steps, and executes a separate loop for each basic step ("multi-loop").

Compilation removes interpretation overhead and can lead to very concise and CPU-friendly code. In this paper, we put compilation in its most favourable light by assuming that compilation-time is negligible. This is often true in OLAP queries which tend do be rather long-running, and technologies such as JIT in Java and the LLVM framework for C/C++ [12] nowadays provide low (milliseconds) latencies for compiling and linking.

Roadmap: vectorization vs. compilation. Vectorized expressions process one or more input arrays and store the result in an output array. Even though systems like VectorWise go through lengths to ensure that these arrays are CPU cache-resident, this materialization constitutes extra load/store work. Compilation can avoid this work by keeping results in CPU registers as they flow from one expression to the other. Also, compilation as a general technique is orthogonal to any execution strategy, and can only improve performance. We used the VectorWise DBMS[3] to investigate three interesting use cases that highlight the issues around the relationship between compilation and vectorization.

As our first case, Section 2 shows how in Project expression calculations loop-compilation tends to provide the best results, but that this hinges on using block-oriented processing. Thus, compiling expressions in a tuple-at-a-time engine may improve some performance, but falls short of the gains that are possible. In Section 3, our second case is Select, where we show that branch mispredictions hurt loop-compilation when evaluating conjunctive predicates. In contrast, the vectorized approach avoids this problem as it can transform control-dependencies into data-dependencies for evaluating booleans (along [11]). The third case in Section 4 concerns probing large hash-tables, using a HashJoin as an example. Here, loop-compilation gets CPU cache miss stalled while processing linked lists (i.e., hash bucket chains). We show that a mixed approach that uses vectorization for chain-following is fastest, and robust to the various parameters of the key lookup problem. These findings lead to a set of conclusions which we present in Section 5.

[3]See `www.ingres.com/vectorwise`. Data storage and query evaluation in VectorWise is based on the X100 project [1].

Algorithm 1 Implementation of an example query using vectorized and compiled modes. Map-primitives are statically compiled functions for combinations of operations (OP), types (T) and input formats (col/val). Dynamically compiled primitives, such as c000(), follow the same pattern as pre-generated vectorized primitives, but may take arbitrarily complex expressions as OP.

```
// General vectorized primitive pattern
map_OP_T_col_T_col(idx n,T* res,T* col1,T* col2){
  for(int i=0; i<n; i++)
    res[i]=OP(col1[i],col2[i]);
}

// The micro-benchmark uses data stored in:
const idx LEN=1024;
chr tmp1[LEN], tmp2[LEN], one = 100;
sht tmp3[LEN];
int tmp4[LEN];  // final result

// Vectorized code:
map_add_chr_val_chr_col(LEN,tmp1,&one,l_discount);
map_sub_chr_val_chr_col(LEN,tmp2,&one,l_tax);
map_mul_chr_col_chr_col(LEN,tmp3,tmp1,tmp2);
map_mul_int_col_sht_col(LEN,tmp4,l_extprice,tmp3);

// Compiled equivalent of this expression:
c000(idx n,int *res,int *col1,chr* col2,chr* col3){
  for(idx i=0; i<n; i++)
    res[i]=col1[i]*((100-col2[i])*(100+col3[i]));
}
```

2. CASE STUDY: PROJECT

Inspired by the expressions in Q1 of TPC-H we focus on the following simple Scan-Project query as micro-benchmark:

```
SELECT l_extprice*(1-l_discount)*(1+l_tax) FROM lineitem
```

The scanned columns are all decimals with precision two. VectorWise represents these internally as integers, using the value multiplied by 100 in this case. After scanning and decompression it chooses the smallest integer type that, given the actual value domain, can represent the numbers. The same happens for calculation expressions, where the destination type is chosen to be the minimal-width integer type, such that overflow is prevented. In the TPC-H case, the price column is a 4-byte integer and the other two are single-byte columns. The addition and subtraction produce again single bytes, their multiplication a 2-byte integer. The final multiplication multiplies a 4-byte with a 2-byte integer, creating a 4-byte result.

Vectorized Execution. VectorWise executes functions inside a Project as so-called map-primitives. Algorithm 1 shows the example code for a binary primitive. In this, `chr`, `sht`, `int` and `lng` represent internal types for 1-, 2-, 4- and 8-byte integers and `idx` represents an internal type for representing sizes, indexes or offsets within columns of data (implemented as integer of required width). A _val suffix in the primitive name indicates a constant (non-columnar) parameter. VectorWise pre-generates primitives for all needed combinations of operations, types and parameter modes. All functions to support SQL fit in ca. 9000 lines of macro-code, which expands into roughly 3000 different primitive functions producing 200K LOC and a 5MB binary.

It is reasonable to expect that a compiler that claims support for SIMD should be able to vectorize the trivial loop in the `map_` functions. On x86 systems, gcc (we used 4.5.1) usually does so and the Intel compiler icc never fails to. With

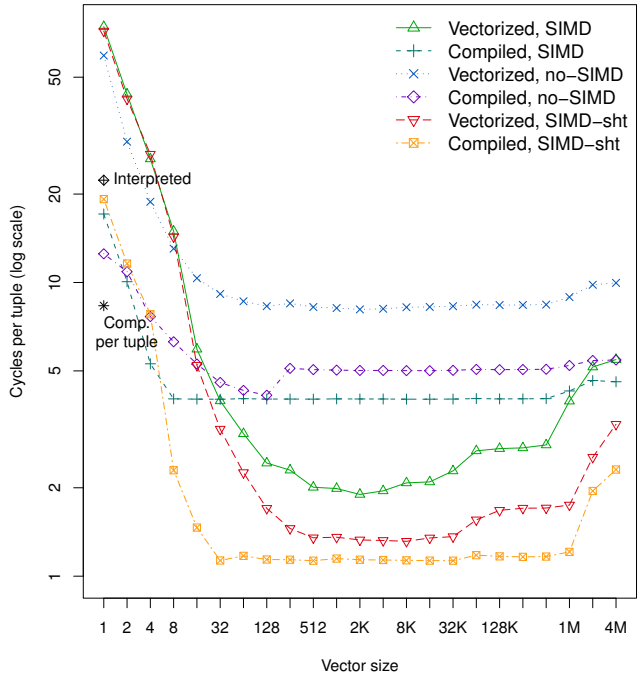


Figure 1: Project micro-benchmark: with and without {compilation, vectorization, SIMD}. The "SIMD-sht" lines work around the alignment sub-optimality in icc SIMD code generation.

a *single* SSE instruction, modern x86 systems can add and subtract 16 single-byte values, multiply 8 single-byte integers into a 2-byte result, or multiply four 4-byte integers. Thus, 16 tuples in this expression could be processed with 8 SIMD instructions: one 8-bit addition, one 8-bit subtraction, two 8-bit multiplications with 16-bit results, and four 32-bit multiplications. All of these instructions store one result and the first two operations load one input (with the other parameter being a constant) while the other two load two inputs. With these 22 (2*2+6*3) load/stores, we roughly need 30 instructions – in reality some additional instructions for casts, padding and looping are required, such that the total for processing 16 tuples is around 60. In comparison, without SIMD we would need 4 instructions (2 loads, 1 calculation, 1 store) per calculation such that a single tuple requires 16 instructions, >4 times more than with SIMD.

The vectorized "SIMD" and "no-SIMD" lines in Figure 1, show an experiment in which expression results are calculated, using different vector-sizes. We used a 2.67GHz Nehalem core, on a 64-bits Linux machine with 12GB of RAM. The no-SIMD vectorized code, produced by explicitly disabling SIMD generation in the compiler (icc 11.0[4], here), is indeed 4 times slower than SIMD. The general trend of decreasing interpretation overhead with increasing vector-size until around one thousand, and performance deteriorating due to cache misses if vectors start exceeding the L1 and L2 caches, has been described already in detail in [13, 1].

Compiled Execution. The lower part of Algorithm 1 shows the compiled code that a modified version of VectorWise can now generate on-the-fly: it combines vectorization with compilation. Such a combination in itself is not new ("compound primitives" [1]), and the end result is similar to what a holistic query compiler like HIQUE [7] generates for this Scan-Project plan, though it would also add Scan code. However, if we assume a HIQUE with a simple main-memory storage system and take `l_tax`, etc. to be pointers into a column-wise stored table, then `c000()` would be the exact product of a "loop-compilation" strategy.

The main benefit of the compiled approach is the absence of load/stores. The vectorized approach needs 22 load/stores, but only the bottom three loads and top-level store are needed by the compiled strategy. Comparing vectorized with compiled, we are surprised to see that the vectorized version is significantly faster (4 vs. 2 cycles/tuple). Close investigation of the generated code revealed that icc chooses in its SIMD generation to align all calculations on the widest unit (here: 4-byte integers). Hence, the opportunities for 1-byte and 2-byte SIMD operations are lost. Arguably, this is a compiler bug or sub-optimality.

In order to show what compilation could achieve, we retried the same, now assuming that `l_extprice` would fit into a 2-byte integer; which are the "SIMD-sht" lines in in Figure 1. Here, we see compilation beating vectorized execution, as one would normally expect in Project tasks. A final observation is that compiled map-primitives are less sensitive to cache size (only to L2, not L1), such that a hybrid vectorized/compiled engine can use large vector-sizes.

Tuple-at-a-time compilation. The black star and diamond in Figure 1, correspond to situations where primitive functions work tuple-at-a-time. The non-compiled strategy is called "interpreted", here. An engine like MySQL, whose whole iterator interface is tuple-at-a-time, can only use such functions as it has just one tuple to operate on at any moment in time. Tuple-at-a-time primitives are conceptually very close to the functions in Algorithm 1 with vector-size `n=1`, but lack the for-loop. We implemented them separately for fairness, because these for-loops introduce loop-overhead. This experiment shows that if one would contemplate introducing compilation in an engine like MySQL without breaking its tuple-at-a-time operator API, the gain in expression calculation performance could be a factor 3 (23 vs 7 cycle/tuple). The absolute performance is clearly below what block-wise query processing offers (7 vs 1.2cycle/tuple), mainly due to missed SIMD opportunities, but also because the virtual method call for every tuple inhibits speculative execution across tuples in the CPU. Worse, in tuple-at-a-time query evaluation function primitives in OLAP queries only make up a small percentage (<5%) of overall time [1], because most effort goes into the tuple-at-a-time operator APIs. The overall gain of using compilation without changing the tuple-at-a-time nature of a query engine can therefore at most be a few percent, making such an endeavour questionable.

3. CASE STUDY: SELECT

We now turn our attention to a micro-benchmark that tests conjunctive selections:

```
WHERE col1 < v1 AND col2 < v2 AND col3 < v3
```

Selection primitives shown in Algorithm 2 create vectors of indexes for which the condition evaluates to true, called

[4]Compiler options are -O3 for gcc, supplemented for icc with -xSSE4.2 -mp1 -unroll

Algorithm 2 Implementations of < selection primitives. All algorithms return the number of selected items (return j). For mid selectivities, branching instructions lead to branch mispredictions. In a vectorized implementation such branching can be avoided. VectorWise dynamically selects the best method depending on the observed selectivity, but in the micro-benchmark we show the results for both methods.

```
// Two vectorized implementations
// (1.) medium selectivity: non-branching code
idx sel_lt_T_col_T_val(idx n,T*res,T*col1,T*val2,
                                              idx*sel){
  if(sel== NULL) {
    for(idx i=0, idx j=0; i<n; i++) {
      res[j] = i; j += (col1[i] < val2[0]);
    }
  } else {
    for(idx i=0, idx j=0; i<n; i++) {
      res[j] = sel[i]; j += (col1[sel[i]] < *val2);
    }
  }
  return j;
}
// (2.) else: branching selection
idx sel_lt_T_col_T_val(idx n,T*res,T*col1,T*val2,
                                              idx*sel){
  if(sel==NULL) {
    for(idx i=0, idx j=0; i<n; i++)
      if(col1[i] < *val2) res[j++] = i;
  } else {
    for(idx i=0, idx j=0; i<n; i++)
      if(col1[sel[i]] < *val2) res[j++] = sel[i];
  }
  return j;
}

// Vectorized conjunction implementation:
const idx LEN=1024;
idx sel1[LEN],sel2[LEN],res[LEN],ret1,ret2,ret3;
ret1 = sel_lt_T_col_t_val(LEN,sel1,col1,&v1,NULL);
ret2 = sel_lt_T_col_t_val(ret1,sel2,col1,&v1,sel1);
ret3 = sel_lt_T_col_t_val(ret2,res,col1,&v1,sel2);
```

selection vectors. Selection primitives can also take a selection vector as parameter, to evaluate the condition only on elements of the vectors from the positions pointed to by the selection vector [5]. A vectorized conjunction is implemented by chaining selection primitives with the output selection vector of the previous one being the input selection vector of the next one, working on a tightening subset of the original vectors, evaluating this conjunction lazily only on those elements for which the previous conditions passed.

Each condition may be evaluated with one of two implementations of selection primitive. The naive "branching" implementation of selection evaluates conditions lazily and branches out if any of the predicates fails. If the selectivity of conditions is neither very low or high, CPU branch predictors are unable to correctly guess the branch outcome. This prevents the CPU from filling its pipelines with useful future code and hinders performance. In [11] it was shown that a branch (control-dependency) in the selection code can be transformed into a data dependency for better performance.

The `sel_lt` functions in Algorithm 2 contain both approaches. The VectorWise implementation of selections uses a mechanism that chooses either the branch or non-branch

[5]In fact, other primitives are also able to work with selection vectors, but it was removed from code snippets where not necessary for the discussed experiments.

Algorithm 3 Four compiled implementations of a conjunctive selection. Branching cannot be avoided in loop-compilation, which combines selection with other operations, without executing these operations eagerly. The four implementations balance between branching and eager computation.

```
// (1.) all predicates branching ("lazy")
idx c0001(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(i=0; i<n; i++)
    if (col1[i]<*v1 && col2[i]<*v2 && col3[i]<*v3)
      res[j++] = i;
  return j; // return number of selected items.
}

// (2.) branching 1,2, non-br. 3
idx c0002(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(j=0; i<n; i++)
    if (col1[i]<*v1 && col2[i] < *v2) {
      res[j] = i; j += col3[i] < *v3;
    }
  return j;
}

// (3.) branching 1, non-br. 2,3
idx c0003(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(i=0; i<n; i++)
    if (col1[i]<v1) {
      res[j] = i; j += col2[i]<*v2 & col3[i]<*v3
    }
  return j;
}

// (4.) non-branching 1,2,3, ("compute-all")
idx c0004(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i, j=0;
  for(i=0; i<n; i++) {
    res[j] = i;
    j += (col1[i]<*v1 & col2[i]<*v2 & col3[i]<*v3)
  }
  return j;
}
```

strategy depending on the observed selectivity [6]. As such, its performance achieves the minimum of the vectorized branching and non-branching lines in Figure 2.

In this experiment, each of the columns `col1`, `col2`, `col3` is an integer column, and the values `v1`, `v2` and `v3` are constants, adjusted to control the selectivity of each condition. Here, we keep the selectivity of each branch equal, hence to the cube root of the overall selectivity, which we vary from 0 to 1. We performed the experiment on 1K input tuples.

Figure 2 shows that compilation of conjunctive Select is inferior to the pure vectorized approach. The lazy compiled program does slightly outperform vectorized branching, but for the medium selectivities branching is by far not the best option. The gist of the problem is that the trick of (i) converting all control dependencies in data dependencies while still (ii) avoiding unnecessary evaluation, cannot be achieved in a single loop. If one avoids all branches (the "compute-all" approach in Algorithm 3), all conditions always get evaluated, wasting resources if a prior condition already failed.

[6]It even re-orders dynamically the conjunctive predicates such that the most selective is evaluated first.

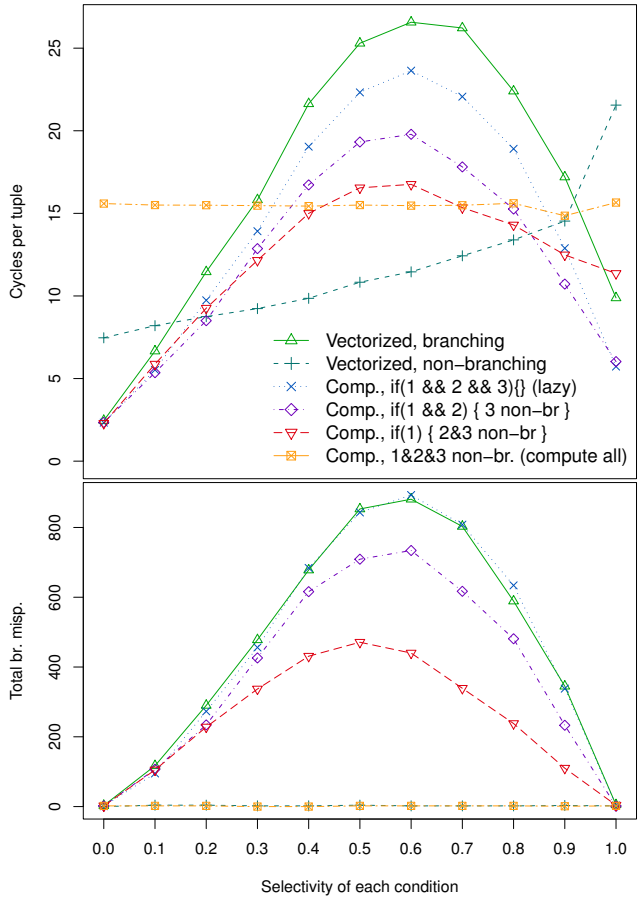


Figure 2: Conjunction of selection conditions: total cycles and branch mispredictions vs. selectivity

One can try mixed approaches, branching on the first predicates and using data dependency on the remaining ones. They perform better in some selectivity ranges, but maintain the basic problems – their worst behavior is when the selectivity after branching predicates is around 50%.

4. CASE STUDY: HASH JOIN

Our last micro-benchmark concerns Hash Joins:

```
SELECT build.col1, build.col2, build.col3
WHERE probe.key1 = build.key1 AND probe.key2 = build.key2
FROM probe, build
```

We focus on an equi-join condition involving keys consisting of two (integer) columns, because such composite keys are more challenging for vectorized executors. This discussion assumes simple bucket-chaining, such as used in VectorWise, presented in Figure 3. This means that keys are hashed on buckets in an array B with size N which is a power of two. Each bucket contains the offset of a tuple in a value space V. This space can either be organized using DSM or NSM layout; VectorWise supports both [14]. It contains the values of the build relation, as well as a next-offset, which implements the bucket chain. A bucket may have a chain of length > 1 either due to hash collisions, or because there are multiple tuples in the build relation with the same key.

Algorithm 4 Vectorized implementation of hash probing.

```
map_hash_T_col(idx n, idx* res, T* col1){
  for(idx i=0; i<n; i++)
    res[i] = HASH(col1[i]);
}
map_rehash_idx_col_T_col(idx n, idx* res,
        idx* col1, T* col2) {
  for (idx i=0; i<n; i++)
    res[i] = col1[i] ^ HASH(col2[i]);
}
map_fetch_idx_col_T_col(idx n, T* res,
        idx* col1, T* base, idx* sel){
  if(sel) {
    for (idx i=0;i<n; i++)
      res[sel[i]] = base[col1[sel[i]]];
  } else {/* sel == NULL, omitted */}
}
map_check_T_col_idx_col_col_T_col(idx n, chr* res,
        T* keys, T* base, idx* pos, idx* sel) {
  if(sel) {
    for(idx i=0; i<n; i++)
      res[sel[i]] =
            (keys[sel[i]]!=base[pos[sel[i]]]);
  } else {/* sel == NULL, omitted */}
}
map_recheck_chr_col_T_col_idx_col_T_col(idx n,
        chr* res, chr* col1,
        T* keys, T* base, idx* pos, idx* sel) {
  if(sel) {
    for(idx i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] ||
            (keys[sel[i]]!=base[pos[sel[i]]]);
  } else {/* sel == NULL, omitted */}
}
ht_lookup_initial(idx n, idx *pos, idx* match,
        idx* H, idx* B) {
  for(idx i=0,k=0; i<n; i++) {
    // saving found chain head position in HT
    pos[i] = B[H[i]];
    // saving to a sel. vector if non-zero
    if(pos[i]) { match[k++] = i; }
  }
}
ht_lookup_next(idx n,idx* pos,idx* match,
        idx* next) {
  for(idx i=0,k=0; i<n; i++) {
    // advancing to next in chain
    pos[match[i]] = next[pos[match[i]]];
    // saving to a sel. vec. if non-empty
    if(pos[match[i]]) { match[k++] = match[i]; }
  }
}
```

procedure HTPROBE($V, B[0..N-1], \vec{K_{1..k}}$**(in)**, $\vec{R_{1..v}}$**(out)**)
  // Iterative hash-number computation
  $\vec{H} \leftarrow map\_hash(\vec{K_1})$
  for each remaining key vectors K_i **do**
    $\vec{H} \leftarrow map\_rehash(\vec{H}, \vec{K_i})$
  $\vec{H} \leftarrow map\_bitwiseand(\vec{H}, N-1)$
  // Initial lookup of candidate matches
  $\vec{Pos}, \vec{Match} \leftarrow ht\_lookup\_initial(H, B)$
  while $\vec{Match}$ not empty **do**
    // Candidate value verification
    $\vec{Check} \leftarrow map\_check(\vec{K_1}, V_{key_1}, \vec{Pos}, \vec{Match})$
    for each remaining key vector $\vec{K_i}$ **do**
      $\vec{Check} \leftarrow map\_recheck(\vec{Check}, \vec{K_i}, V_{key_i}, \vec{Pos}, \vec{Match})$
    $\vec{Match} \leftarrow sel\_nonzero(\vec{Check}, \vec{Match})$
    // Chain following
    $\vec{Pos}, \vec{Match} \leftarrow ht\_lookup\_next(\vec{Pos}, \vec{Match}, V_{next})$
  $\vec{Hits} \leftarrow sel\_nonzero(\vec{Pos})$
  // Fetching the non-key attributes
  for each result vector $\vec{R_i}$ **do**
    $\vec{R_i} \leftarrow map\_fetch(\vec{Pos}, V_{value_i}, \vec{Hits})$

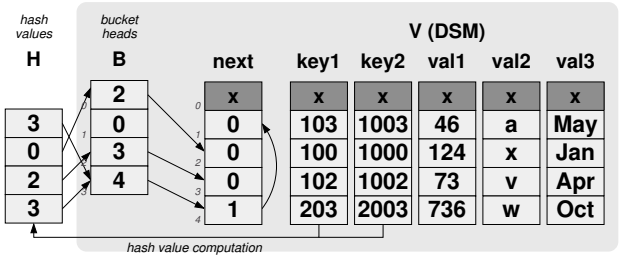


Figure 3: Bucket-chain hash table as used in VectorWise. The value space V presented in the figure is in DSM format, with separate array for each attribute. It can also be implemented in NSM, with data stored tuple-wise.

Vectorized Hash Probing. For space reasons we only discuss the probe phase in Algorithm 4, we show code for the DSM data representation and we focus on the scenario when there is at most one hit for each probe tuple (as is common with relations joined with a foreign-key referential constraint). Probing starts by vectorized computation of a hash number from a key in a column-by-column fashion using map-primitives. A `map_hash_T_col` primitive first hashes each key of type `T` onto a `lng` long integer. If the key is composite, we iteratively refine the hash value using a `map_rehash_lng_col_T_col` primitive, passing in the previously computed hash values and the next key column. A bitwise-and map-primitive is used to compute a bucket number from the hash values: $H\&(N\text{-}1)$.

To read the positions of heads of chains for the calculated buckets we use a special primitive `ht_lookup_initial`. It behaves like a selection primitive, creating a selection vector $\vec{Match}$ of positions in the bucket number vector H for which a match was found. Also, it fills the $\vec{Pos}$ vector with positions of the candidate matching tuples in the hash table. If the value (offset) in the bucket is 0, there is no key in the hash table – these tuples store 0 in $\vec{Pos}$ and are not part of $\vec{Match}$.

Having identified the indices of possible matching tuples, the next task is to "check" if the key values actually match. This is implemented using a specialized map primitive that combines fetching a value by offset with testing for non-equality: `map_check`. Similar to hashing, composite keys are supported using a `map_recheck` primitive which gets the boolean output of the previous check as an extra first parameter. The resulting booleans mark positions for which the check failed. The positions can then be selected using a select `sel_nonzero` primitive, overwriting the selection vector $\vec{Match}$ with positions for which probing should advance to the "next" position in the chain. Advancing is done by a special primitive `ht_lookup_next`, which for each probe tuple in $\vec{Match}$ fills $\vec{Pos}$ with the next position in the bucket-chain of V. It also guards for ends of chain by reducing $\vec{Match}$ to its subset for which the resulting position in $\vec{Pos}$ was non-zero.

The loop finishes when the $\vec{Match}$ selection vector becomes empty, either because of reaching end of chain (element in $\vec{Pos}$ equals 0, a miss) or because checking succeeded (element in $\vec{Pos}$ pointing to a position in V, a hit).

Hits can be found by selecting the elements of $\vec{Pos}$ which ultimately produced a match with a `sel_nonzero` primitive.

Algorithm 5 Fully loop-compiled hash probing: for each NSM tuple, read hash bucket from B, loop through a chain in V, fetching results when the check produces a match

```
for(idx i=0, j=0; i<n; i++) {
  idx pos, hash = HASH(key1[i])^HASH(key2[i]);
  if (pos = B[hash&(N-1)]) do {
    if(key1[i]==V[pos].key1 &&
       key2[i]==V[pos].key2) {
        res1[i] = V[pos].col1;
        res2[i] = V[pos].col2;
        res3[i] = V[pos].col3;
        break; // found match
    }
  } while(pos = V.next[pos]); // next
}
```

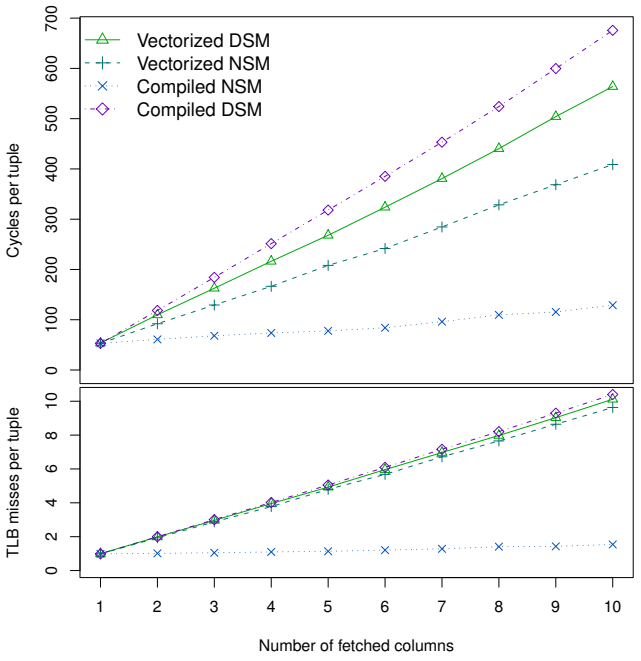


Figure 4: Fetching columns of data from a hash table: cycles per tuple and total TLB misses

$\vec{Pos}$ with selection vector $\vec{Hits}$ becomes a pivot vector for fetching. This pivot is subsequently used to fetch (non-key) result values from the build value area into the hash join result vectors; using one fetch primitive for each result column.

Partial Compilation. There are three opportunities to apply compilation to vectorized hashing. The first is to compile the full sequence of hash/rehash/bitwise-and and bucket fetch into a single primitive. The second combines the check and iterative re-check (in case of composite keys) and the select > 0 into a single select-primitive. Since the test for a key in a well-tuned hash table has a selectivity around 75%, we can restrict ourselves to a non-branching implementation. These two opportunities re-iterate the compilation benefits of Project and Select primitives, as discussed in the previous sections, so we omit the code.

The third opportunity is in the fetch code. Here, we can generate a composite fetch primitive that, given a vector of positions, fetches multiple columns. The main benefit of this

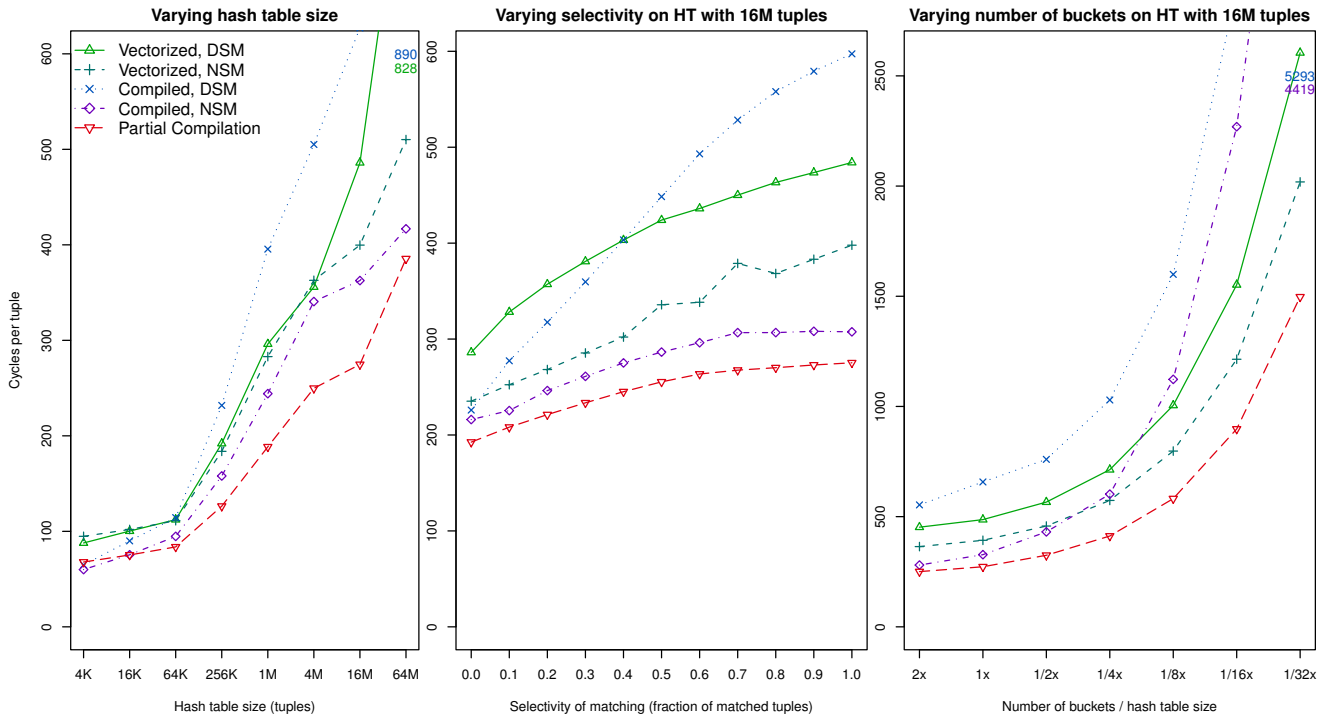


Figure 5: Hash table probing. Left: different sizes of the hash table value space V. Middle: different match rates of the probe. Right: different sizes of the bucket array B w.r.t. value space V (different chain lengths).

is obtained in case of NSM organisation of the value space V. Vectorization fetches values column-at-a-time, hence passes over the NSM value space as many times as there are result columns (here: 3), accessing random positions from the value space on each pass. On efficient vector-sizes, the amount of random accesses is surely larger than TLB cache size and may even exceed the amount of cache lines, such that TLB and cache trashing occurs, with pages and cache lines from the previous pass being already evicted from the caches before the next. The compiled fetch fetches all columns from the same position at once, achieving more data locality. Figure 4 shows that in normal vectorized hashing performance of NSM and DSM is similar, but compilation makes NSM clearly superior.

Full Compilation. It is possible to create a loop-compiled version of the full join, like e.g. proposed in HIQUE [7]. Algorithm 5 shows the hash probe section of such an algorithm, which we also tested. It loops over probe keys, and for each probe key fetches the corresponding bucket, then iterates over the bucket-chain, checking the key for equality, and if equal, fetches the needed result columns. We try to explain in the following why this fully compiled algorithm is inferior to the vectorized alternative with partial compilation.

Parallel Memory Access. Because memory latency is not improving much (∼100ns), and cache line granularities must remain limited (64bytes), memory bandwidth on modern hardware is no longer simply the division between these two. A single Nehalem core can achieve a factor 10 more than this 0.64GB/s, thanks to automatic CPU prefetching on sequential access. Performance thus crucially depends on having multiple outstanding memory requests at all times. For random access, this is hard to achieve, but the deeply pipelined out-of-order nature of modern CPUs does help. That is, if a load stalls, the CPU might be able to speculate ahead into upstream instructions and reach more loads. The Intel Nehalem architecture can have four outstanding loads, improving bandwidth by a factor four[7]. Success is not guaranteed, since the instruction speculation window of a CPU is limited, depends on branch prediction, and only independent upstream instructions can be taken into execution.

The Hard-To-Understand Part. The crux here is that the vectorized fetch primitive trivially achieves whatever maximum outstanding loads the CPU supports, as it is a tight loop of independent loads. The same holds for the partially compiled variants. The fully compiled hash probe, however, can run aground while following the bucket chain. Its performance is only good if the CPU can speculate ahead across multiple probe tuples (execute concurrently instructions from multiple for-loop iterations on `i`). That depends on the branch predictor predicting the `while(pos..)` to be false, which will happen in join key distributions where there are no collisions. If, however, there are collisions or if the build relation has multiple identical keys, the CPU will stall with a single outstanding memory request (`V[pos]`), because the branch predictor will make it stay in the while-loop, and it will be unable to proceed as the value of `pos = V.next[pos]` is unknown because it depends on the current cache/TLB miss. A similar effect has been described in the context of using explicit prefetching instructions in hash-joins [3]. This

[7]Speculation-friendly code is thus more effective than manual prefetching, which tends to give only minor improvement, and is hard to tune/maintain for multiple platforms.

effect causes fully compiled hashing to be four times slower than vectorized hashing in the worst case.

Experiments. Figure 5 shows experiments for hash probing using the vectorized, fully and partially compiled approaches, using both DSM and NSM as the hash table (V) representation. We vary hash table size, selectivity (fraction of probe keys that match something), and bucket chain length; which have default values resp. 16M, 1.0 and 1. The left part shows that when the hash table size grows, performance deteriorates; it is well understood that cache and TLB misses are to blame, and DSM achieves less locality than NSM. The middle graph shows that with increasing hit rate, the cost goes up, which mainly depends on increasing fetch work for tuple generation. The compiled NSM fetch alternatives perform best, as explained. The right graph shows what happens with increasing chain length. As discussed above, the fully compiled (NSM) variant suffers most, as it gets no parallel memory access. The overall best solution is partially compiled NSM, thanks to its efficient compiled multi-column fetching (and to a lesser extent efficient checking/hashing, in case of composite keys) and its parallel memory access, during lookup, fetching and chain-following.

5. CONCLUSIONS

For database architects seeking a way to increase the computational performance of a database engine, there might seem to be a choice between vectorizing the expression engine versus introducing expression compilation. Vectorization is a form of block-oriented processing, and if a system already has an operator API that is tuple-at-a-time, there will be many changes needed beyond expression calculation, notably in all query operators as well as in the storage layer. If high computational performance is the goal, such deep changes cannot be avoided, as we have shown that if one would keep adhering to a tuple-a-time operator API, expression compilation alone only provides marginal improvement.

Our main message is that one does not need to choose between compilation and vectorization, as we show that best results are obtained if the two are combined. As to what this combining entails, we have shown that "loop-compilation" techniques as have been proposed recently can be inferior to plain vectorization, due to better (i) SIMD alignment, (ii) ability to avoid branch mispredictions and (iii) parallel memory accesses. Thus, in such cases, compilation should better be split in multiple loops, materializing intermediate vectorized results. Also, we have signaled cases where an interpreted (but vectorized) evaluation strategy provides optimization opportunities which are very hard with compilation, like dynamic selection of a predicate evaluation method or predicate evaluation order.

Thus, a simple compilation strategy is not enough; state-of-the art algorithmic methods may use certain complex transformations of the problem at hand, sometimes require run-time adaptivity, and always benefit from careful tuning. To reach the same level of sophistication, compilation-based query engines would require significant added complexity, possibly even higher than that of interpreted engines. Also, it shows that vectorized execution, which is an evolution of the iterator model, thanks to enhancing it with compilation further evolves into an even more efficient and more flexible solution without making dramatic changes to the DBMS architecture. It obtains very good performance while maintaining clear modularization, simplified testing and easy performance- and quality-tracking, which are key properties of a software product.

6. REFERENCES

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[2] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[3] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. ICDE*, Boston, MA, USA, 2004.

[4] D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[5] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[6] A. Kemper and T. Neumann. HyPer: Hybrid OLTP and OLAP High Performance Database System. Technical report, Technical Univ. Munich, TUM-I1010, May 2010.

[7] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[8] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proc. ICDE*, Heidelberg, Germany, 2001.

[9] ParAccel Inc. Whitepaper. *The ParAcel Analytical Database: A Technical Overview*, Feb 2010. `http://www.paraccel.com`.

[10] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *Proc. ICDE*, Atlanta, GA, USA, 2006.

[11] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[12] The LLVM Compiler Infrastructure . `http://llvm.org`.

[13] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. Ph.D. Thesis, Universiteit van Amsterdam, Sep 2009.

[14] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. 2008.

QMD: Exploiting Flash for Energy Efficient Disk Arrays

Sean M. Snyder† Shimin Chen* Panos K. Chrysanthis† Alexandros Labrinidis†
†University of Pittsburgh *Intel Labs

ABSTRACT

Energy consumption for computing devices in general and for data centers in particular is receiving increasingly high attention, both because of the increasing ubiquity of computing and also because of increasing energy prices. In this work, we propose QMD (Quasi Mirrored Disks) that exploit flash as a write buffer to complement RAID systems consisting of hard disks. QMD along with partial on-line mirrors, are a first step towards energy proportionality which is seen as the "holy grail" of energy-efficient system design. QMD exhibits significant energy savings of up 31%, as per our evaluation study using real workloads.

1. INTRODUCTION

A growing concern, energy consumption in data centers has been the focus of numerous white papers, research studies, news reports, and recent NSF workshops [4, 23, 10, 7, 1, 2]. According to a report to U.S. congress [23], the total energy consumption by servers and data centers in U.S. was about 61 billion kWh in 2006, and is projected to nearly double by 2011 [23]. To make matters worse, global electricity prices increased 56% between 2002 and 2006 [7]. The two trends of growing energy consumption and rising energy prices lead to increasingly higher electricity bills for data centers. Energy cost can become a dominant factor in the total cost of ownership of computing infrastructure [4], and the annual electricity cost of data centers in U.S. in 2011 is projected to be $7.4 billion [23]. Among the components in data centers, it has been shown that storage experienced the fastest annual growth (20% between 2000 and 2006) in energy consumption [23]. As hard disk drives (HDDs) are the dominant technology for data storage today, we are interested in improving energy efficiency for data storage that consists of mainly HDDs.

A key goal in energy efficient system design is to achieve energy proportionality [5], i.e., energy consumption being proportional to the system utilization. Unlike solid state devices, such as microprocessors, hard disk drives (HDDs) contain moving components, making it difficult to achieve this goal. For example, for an enterprise class disk, the idle power for spinning the disk platters is often about 60–80% of its active energy [20, 21]. While a disk can be spun down to standby mode for saving most of this power, it takes on the order of 10 seconds to spin up a disk, potentially incurring significant slowdowns in application response times.

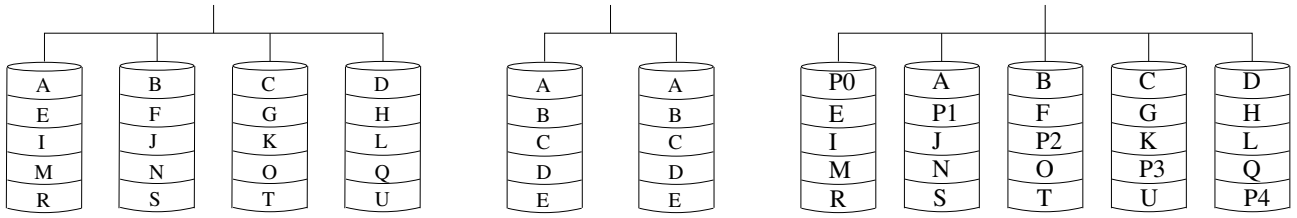
Previous Approach: Exploit Redundancy and NVRAM. One promising solution is to exploit the inherent redundancy in storage systems for conserving energy [11, 24, 16]. Today, most storage systems employ redundancy (e.g., RAID) to achieve high reliability, high availability, and high performance requirements for many important applications. For example, the TPC-E benchmark, which models transaction processing in a financial brokerage house, requires redundancy for the data and logs [22]. As seen by published TPC-E reports on the TPC web site, this requirement is typically achieved by the use of RAID disk arrays.

For saving energy, the idea is to keep only a single copy of the data active and spin down disks containing redundant copies of the data under low load. We call the disks containing the active copy of data, the *active* disks, and the disks that are spun down, the *standby* disks. Note that the approach must guarantee the same level of reliability for write operations under low load (e.g., writing to two non-volatile devices). To achieve this, previous studies [11, 24, 16] propose to use NVRAM (i.e., battery-backed RAM) as non-volatile write buffers. When the system is under low utilization, reads are sent to the active disks, while writes are sent to both the active disks and to the NVRAM buffers. When the system sees high load or when the NVRAM buffers are full, the standby disks are spun up and the buffered writes are applied. Depending on the RAID organization, this approach may potentially save up to 50% energy when the system is under low utilization.

Limitation of NVRAM-Based Approach. Besides concerns about correlations between battery failure and power loss, battery-backed RAM is expensive. The NVRAM write buffer size is often limited to a few hundred MB for an entire RAID array. Typically, a server-class disk can support about 100MB/s read/write bandwidth. Suppose that under low load, a disk sees 1MB/s write traffic. Then, a 500MB NVRAM buffer will be filled up for the write traffic of a single disk in less than 9 minutes. When the buffer is filled, the standby disks must be spun up to apply the buffered writes. However, a disk supports only a limited number of spin-up/down operations because they introduce wear to the motor and the heads in a disk. In particular, server and desktop disks are often rated at 50,000 spin-up/down cycles (a.k.a. start-stop cycles) [19]. Given a five-year lifetime, this puts a limitation of an average 1.1 spin-up/down per hour. Therefore, the above example with a *single* standby disk will significantly shorten the disk lifetime by about 6 times! Note that in real-world disk arrays, an NVRAM buffer in a RAID controller often handles tens of disks, and thus the situation could be an order of magnitude worse.

Our Solution: Exploiting Flash as Write Buffer. We propose to exploit flash as the write buffer for addressing this problem. There are several desirable properties of flash: (i) it is non-volatile; (ii) flash is much cheaper with much larger capacity; and (iii) flash is

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.



(a) RAID 0 (striping) (b) RAID 1 (mirrored disks) (c) RAID 5 (striping with distributed parity)

Figure 1: Basic RAID schemes. (Capital letters represent data blocks; Px represents a parity block.)

energy efficient and supports energy proportionality well. Moreover, flash-based cache products with hundreds of GB capacity are already available for storage systems [13]. One can utilize the same flash for saving energy. This nicely shares the resource: *the flash-based cache improves I/O performance under high system load and saves energy under low load*.

In our design, we aim to achieve low spin-up/down counts while preserving the performance of RAID under low utilization, under high utilization, and during state transitions. We call the solution in this paper, QMD (Quasi Mirrored Disks), as we mainly focus on mirror-based RAID schemes (e.g., RAID 1 and RAID 10), and we study partial on-line mirrors in order to achieve the ideal goal of energy proportionality. We present preliminary evaluation of our solution by simulating I/O traces of real-world applications. Experimental results show that QMD can save 11%–31% energy, and reduce the number of spin-up/downs by 80%.

Outline. Section 2 provides background on exploiting redundancy for RAID arrays. Section 3 presents our solution, QMD, and discusses our research direction for achieving the energy proportionality goal. Section 4 evaluates QMD using real-world I/O traces. Section 5 describes related work. Section 6 concludes the paper.

2. EXPLOITING STORAGE REDUNDANCY FOR SAVING ENERGY

We start by refreshing our memory of the common RAID schemes in Section 2.1. Then, in Section 2.2, we describe the basic operations for exploiting RAID redundancy for saving energy.

2.1 Background: Common RAID Schemes

Figure 1 shows three basic RAID schemes [14]. They are widely used and serve as building blocks for composite RAID schemes.

- *RAID 0 (a.k.a. striping)* places data blocks across disks in a round robin fashion for high performance. RAID 0 does not provide redundancy, and therefore is often combined with other RAID schemes for reliability and availability.
- *RAID 1 (a.k.a. mirrored disks)* mirrors data blocks between two (or more) disks. Every write is sent to both disks, while a read can be served by either disk. Therefore, RAID 1 of two disks achieves twice the read bandwidth of a single disk and 100% data redundancy. It can tolerate one disk failure.
- *RAID 5 (a.k.a. striping)* stripes data across N ($N \geq 3$) disks. The N blocks at the same disk address form a stripe group. One of them is a parity block, computed as the bit-wise XOR of the other $N-1$ data blocks. RAID 5 places the parity blocks across disks in a round robin fashion. Every write must modify both the target data block and the associated parity block, requiring two reads for fetching the two old blocks followed by two writes. RAID 5 can tolerate a single disk failure. A data block of the failed disk can be reconstructed as the bit-wise XOR of the other $N-1$ blocks in the same stripe group.
- *Composite RAID Schemes:* RAID 10 (i.e., 1+0) is a stripe of mirrors, where every disk in RAID 0 (in Figure 1(a)) is replaced with a pair of mirrored disks. A RAID 10 of $2N$ disks can tolerate N disk failures (each in a disjoint mirror). Similarly, RAID 50 replaces every disk in RAID 0 with a RAID 5.

Redundancy is achieved by either mirror-based schemes (e.g., RAID 10) or parity-based schemes (e.g., RAID 5). The main advantage of the latter is that it saves disk capacity; RAID 5 of N disks utilizes $1 - \frac{1}{N}$ of the total capacity, while RAID 1 utilizes only 50%. In other words, given the same individual disk capacity and the target total capacity, RAID 5 uses fewer number of disks than RAID 10. However, as disk capacity has been growing exponentially, total capacity is less of a concern today. The number of disks in RAID is often determined by application performance requirements in terms of throughput and IOs per seconds (IOPS), rather than total available capacity, as evidenced by many TPC results.

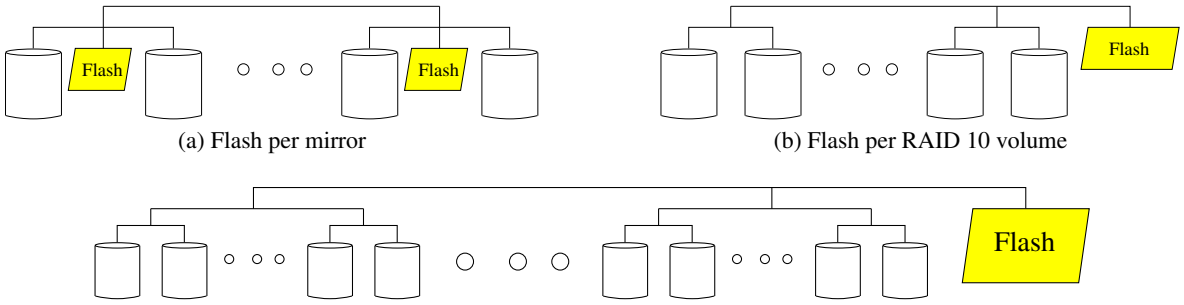
On the other hand, mirror-based schemes have higher write performance than parity-based schemes during normal operations. Moreover, when a disk fails, mirror-based schemes can serve data directly from good disks, while parity-based schemes suffer from large performance degradation due to the many I/Os for retrieving blocks in the same stripe group to reconstruct a block on the failed disk. As a result, mirror-based schemes become increasingly popular today. For example, many TPC-E benchmark results that store data on disks use RAID 10 for reliability.[1] We focus on mirror-based schemes in this paper.

2.2 Basic Operations for Saving Energy

Previous studies [11, 24, 16] propose to exploit the RAID redundancy for saving energy and uses NVRAM for achieving reliability when the system is under low utilization. The basic operations are:

1. *All Disks under High Utilization:* The system performs normal RAID operations with all disks running under high load.
2. *Active Disks with NVRAM under Low Utilization:* When the system is under low utilization, a number of disks is spun down to save energy. This number depends on the RAID scheme. In mirror-based schemes, one disk in every mirror can be spun down, thus potentially saving up to 50% energy. In parity-based schemes (e.g., RAID 5 with N disks), one disk in every parity group can be spun down (saving up to $\frac{1}{N}$ energy in RAID 5). Reads are handled by the active disks; parity-based schemes require the costly XOR computation for reconstructing disk blocks on the standby disks. Writes are sent to both

[1] A few published TPC-E results (including the current Watts/Performance lead, Fujitsu PRIMERGY RX300 S6 12x2.5) store data mainly on arrays of flash-based SSDs for reducing energy consumption. In such configurations, SSD capacity is a much more significant concern and therefore RAID 5 is often employed. However, the focus of this paper is on using a small amount of flash for improving the energy efficiency of disk storage, which is the dominant storage technology today.



(a) Flash per mirror

(b) Flash per RAID 10 volume

(c) Flash for all RAID 10 volumes

Figure 2: Flash-enhanced RAID 10 schemes.

the active disks and the NVRAM to guarantee reliability. Comparing mirror-based and parity-based schemes, it is clear that the approach fits mirror-based schemes better.

3. *Flushing NVRAM Data to Standby Disks:* When the NVRAM write buffer is full or when the system transitions from low utilization back to high utilization, the data cached on NVRAM must be flushed to bring the standby disks up to date.

There are two main problems of this approach. First, the capacity of NVRAM is typically limited to a few hundred MB, potentially incurring frequent flushing operations, as shown in the back-of-envelope computation in the Introduction. Frequent flushing operations can both dramatically reduce disk lifetimes and reduce the energy savings because the standby disks must be spun up for writing the buffered data. In this paper, we address this problem by exploiting flash as a large-capacity nonvolatile write buffer.

Second, the energy savings are bounded by the RAID schemes, leaving a big gap to reach the ideal goal of energy proportionality. For example, when the system is under 1% load, RAID 10 still keeps 50% of the disks active. We discuss potential solutions to this problem, which requires coordination between applications and storage systems to avoid performance problems.

3. QUASI MIRRORED DISKS

We propose QMD (Quasi Mirrored Disks) that exploits flash for removing the limitation of NVRAM in Section 3.1, and discuss *partial on-line mirrors* as future research direction for achieving the goal of energy proportionality in Section 3.2.

3.1 Flash-Based Write Cache

We analyze the access pattern of the write cache. Under low utilization, block writes are appended to the write cache, resulting in sequential writes. During flushing, the write cache must support random reads for two reasons: (i) we would like to reorder and optimize the block writes to disks; and (ii) during the transition from low utilization to high utilization mode, we want the write cache to serve incoming I/O requests in order to minimize the impact of flushing on front-end operations. As flash supports both the above access patterns well, we propose to use flash as the write cache.

Figure 2 shows three ways to include flash-based write caches in QMD, using RAID 10 as an example RAID scheme. In (a), we enhance every pair of mirrored disks with a separate flash device. In (b), we use a flash device for an entire RAID 10 volume that stripes data across the mirror pairs. In (c), multiple RAID 10 volumes share the same flash device. From (a) to (c), we reduce the number of flash devices in the system. The benefits are two folds: lowering the total cost and allowing dynamic sharing of the flash capacity across disks. The latter is especially important for multiple RAID 10 volumes because different volumes present separate

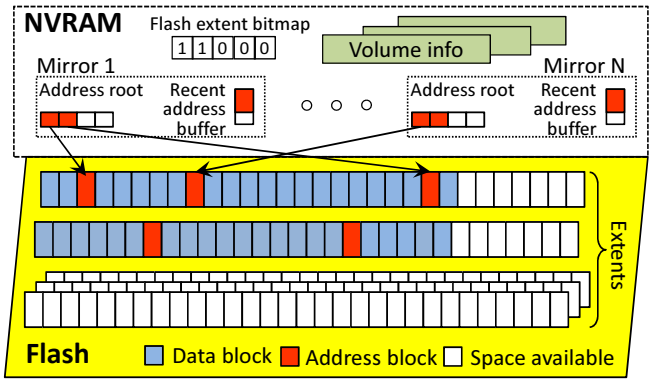


Figure 3: QMD data structures.

I/O address spaces to software and typically store different files. Therefore, they may see very different utilizations. For example, some volumes may be under high utilization, while others under low utilization. The volumes under low utilizations may see different write traffic and thus consume the write cache at different speeds. Dynamic sharing can balance the spin-up/down (flushing) frequencies across different volumes and achieve the same maximum flushing frequency with reduced total write cache size.

However, dynamic sharing introduces complexities in flash cache management. For example, a naive design may employ a single log-structured layout for the flash space. However, different volumes may perform the flush operations at different times, leaving many holes in the log. This either incurs random writes or requires expensive garbage collection operations. In the following, we describe our proposal to efficiently address these complexities.

Flash Space Management. Figure 3 illustrates the data structures of QMD on flash and in NVRAM. Since most functions of NVRAM (e.g., non-volatile write cache during both high and low utilizations) can be satisfied by the flash cache, cost-effective storage designs may reduce the size of NVRAM significantly (e.g., by 10X). Therefore, we use only a small amount of NVRAM in our design mainly as a staging area to reduce wasteful flash writes.

The flash space is divided into fixed sized (e.g., 1GB) extents, as shown in Figure 3. Extent is the unit of flash space allocation. A bitmap in NVRAM keeps track of the availability of all extents. A RAID 10 volume allocates a flash extent at a time. Writes to the volume are appended to the current extent; only when the extent is filled does the volume allocates a new extent. Let V be the number of volumes, C the flash capacity, and E the extent size. We choose E to satisfy $\frac{C}{E} >> V$, i.e., the number of extents is much larger

```
1: Load address blocks of this mirror into memory and sort the address
   entries in disk address order (let A[0..M−1] be the sorted array, where
   M is the total number of entries);
2: Queue is the incoming request queue for this mirror;
3: ToWrite = M; Last = −1; Direction = 1; Bound = M;
4: while ToWrite > 0 do
5:   while Queue.not_empty && Queue.head is a write do
6:     R = Queue.dequeue();
7:     Record R's data in flash and R's target address in NVRAM;
8:   end while
9:   if Queue.not_empty && Queue.head is a read then
10:    R = Queue.dequeue();
11:    Search R's disk address in A[...], using binary search;
12:    if there is a match then
13:      Complete request R by reading the data block from flash;
14:    else
15:      Send R to the disk;
16:      A[i] is immediately to the left of R;
17:      if Last < i then
18:        Direction = 1; Bound = M; Last = i;
19:      else
20:        Direction = −1; Bound = −1; Last = i + 1;
21:      end if
22:    end if
23:  end if
24:  WriteBack = 0;
25:  for (j = Last + Direction; (j != Bound) &&
       (WriteBack < K); j = j + Direction) do
26:    if A[j] is valid then
27:      Process A[j]: read its data block from flash and send write
         request to the disk;
28:      Mark A[j] to be invalid;
29:      WriteBack++;
30:    end if
31:  end for
32:  ToWrite = ToWrite − WriteBack; Last = j;
33: end while
```

Figure 4: Flush operation for a pair of mirrored disks.

than the number of volumes. In this way, the flash space can be shared to effectively balance the needs of multiple volumes[2].

For every volume, we keep a volume info structure in NVRAM. This structure records all the extents that belong to the volume, the next flash offset to write, and a 1MB sized staging area for flash writes. The latter ensures that writes are performed in large sizes, thereby avoiding the problems of small random flash writes.

Handling Writes under Low Utilization. A write request consists of a data block and the target disk address. We append the data block to the current flash extent, and generate an address entry: (target disk address, flash address of data). However, it is wasteful to incur a (e.g., 4KB) flash write for the small sized address entry. Instead, we store it in a recent address buffer in NVRAM. When this buffer is full, we flush the buffer as an address block to the current flash extent.

As shown in Figure 3, we use a two-level structure to keep track of the data blocks. To facilitate the flushing operation for every pair of mirrored disks, we maintain this structure for every mirror. The first level is the address root in NVRAM, which records the flash offsets of the address blocks for the mirror. The second level consists of the address blocks, which in turn point to the data blocks in flash extents.

Optimizing the Flushing Operation. The system decides to transition a RAID volume from low utilization to high utilization mode

[2]For example, if $C = 100GB$ and $V = 10$, we can choose $E = 1GB$. Then, the scheme can effectively handle even extreme cases such as one volume seeing 90X write traffic than the other volumes.

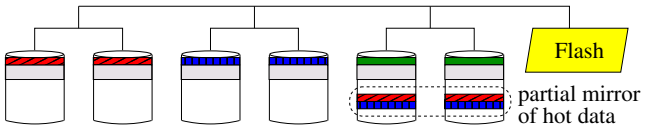


Figure 5: QMD partial on-line mirror. (Red/blue/green: address range for hot data; gray: address range for cold data)

by monitoring the I/O activities. (The implementation will be described in Section 4.) It spins up the standby disks and applies the buffered writes to bring the disks up to date. We would like to optimize the flushing operation for two goals: (i) efficiently writing the buffered data blocks; and (ii) servicing incoming requests during the transition. The key to achieve these goals is to reorder the disk writes to avoid disk seeks.

Figure 4 shows the algorithm for the flushing operation. We can run this algorithm for every pair of mirrored disks in parallel. The algorithm assumes that the volatile DRAM in the storage controller is large enough to hold all the address entries. At the beginning of the algorithm (Line 1), it reads all the address blocks of the mirror and sort the address entries. Sorting serves two purposes: (i) we can easily search incoming read requests to see if there is a hit (Line 11–13); (ii) schedule the write-backs in a disk friendly manner. The algorithm goes into a loop (Line 4–33), which processes K write backs in every iteration. (K is an algorithm parameter that we determine experimentally in Section 4.2.) An iteration first checks incoming requests. For incoming write requests, we simply cache them in the flash and NVRAM (Line 5–8). If there is an incoming read request and it is not a hit, then we send the request to the disk (Line 15). The algorithm remembers the last disk head position, and the previous direction of head movement. It chooses to schedule K write backs from the last head position following the same head movement (Line 25–31).

Space Requirement. In NVRAM, we keep a 1MB sized buffer per RAID volume, and a 4KB recent address buffer per mirror. Suppose there are 10 volumes, and 1000 mirrored disk pairs. Then we require 14MB NVRAM space, which is quite modest.

We also require volatile DRAM for the flushing operation. An address entry consists of a disk block address, and an offset in the flash extent. We can use a 32-bit integer for both addresses. For 4KB blocks, this is sufficient to support 16TB capacity. Therefore, an address entry takes 8 bytes, a 1:512 size ratio to the block size. Suppose the flash cache is 512GB large, then we require 1GB DRAM for the flushing operation, which is reasonable for a large disk array system.

3.2 Partial On-line Mirrors

By exploiting RAID redundancy, we can save at most 50% energy in mirror-based RAID schemes. (The savings in parity based schemes are even smaller.) There is still a big gap to the energy proportionality goal. In this subsection, we discuss our ideas for closing this gap that we would like to explore in future research.

To further save energy, we have to spin down more disks. Therefore, not all data can be available on the active disks. While storage systems may guess the future access patterns based on block-level access history, the penalty of wrong guesses (i.e., the spin-up delay) is high especially for latency sensitive applications.

We propose to allow application software (e.g., database system) and storage systems to collaborate on addressing this problem. Compared to a storage-only solution, application software has higher-level knowledge about data accesses, has more flexibility to schedule data accesses, and can also make end users aware of the

relationship of energy consumption, performance, and data placement. We propose the following two interfaces between application software and storage systems:

- Software can divide the address range of a RAID volume into hot and cold address ranges, as shown in Figure 5. For example, DBMS can create a hot and a cold table spaces. It determines the temperature of database objects based on high-level knowledge of user workloads, then stores them in corresponding table spaces. DBMS may opt to expose the choices to the end users (e.g., DBMS admin) showing also the estimate energy costs and response times for query workloads. The storage system guarantees that data in the hot address range will always be available on active disks, while it may take a spin-up delay to access the cold data.
- Software can use an interface to query the status of a cold address range, i.e., whether a spin-up will be necessary to access it. Software may use this information to intelligently schedule its work. For example, if DBMS finds that a database query must access both hot and cold data, DBMS can choose to process the part of the query involving hot data first, and postpone accessing the cold data to hide the spin-up delay as much as possible.

Given the above collaboration, we can spin down disks based on the system utilization. If the system is utilized 50% or more, then we can exploit redundancy as described previously to spin down at most one disk per pair of mirror. If the system utilization is below 50%, we will spin down both disks in some mirrors. However, we must kept the hot data available on active disks.

As shown in Figure 5, we take advantage of the fact that disk capacity is often under utilized. In many important applications, such as OLTP, the number of disks is determined mainly by the performance requirement rather than capacity demands. Therefore, we can copy the hot data to the active mirror, essentially creating a mirrored copy of the hot data. In Figure 5, we plan to spin down the first and second pairs of mirrored disks, and keep the third pair active. Therefore, we copy their hot data to the third pairs of mirrored disks. We call this approach *partial on-line mirrors*.

This approach utilizes all disks under peak utilization, and is capable of spinning down almost all disks for saving energy for low utilization. One major cost is the overhead for copying the hot data. The copying may be improved in two ways. First, we can copy hot data and spin down the mirrors one mirror at a time. For example, in Figure 5, we can copy hot data from the first mirror then spin down it, before copying hot data from the second mirror. This saves energy during the copying process. Second, we may keep an old version of the hot data on the third mirror. Then the copying process needs to only update the old version, potentially avoiding significant fractions of data copying.

4. EXPERIMENTAL EVALUATION

In this section, we present preliminary evaluation of our proposed QMD solution. In Section 4.1, we perform trace-based simulation study using real-world disk traces for quantifying the overall benefits of our solution in terms of energy savings, reduced spin-up/down cycles, and impact on I/O performance. In Section 4.2, we perform real-machine experiments for understanding the benefits of the proposed flushing operation.

4.1 Simulation Study

We implemented a trace driven RAID controller simulator to evaluate the effectiveness of QMD. We used three real workload traces on RAID10 systems and find that (i) significant energy savings can be achieved with minimal impact on response times, and (ii) increasing the non-volatile write buffer size can significantly reduce the number of disk spin down cycles during a trace.

Table 1: QMD default simulator parameters.

Parameter	Default value	Parameter	Default value
Disk block size	512B	Flash read latency	65 us
RAID Stripe size	128KB	Flash write latency	85 us
Power, under load	13.2W	Flash read bandwidth	250MB/s
Power, spinning idle	7.7W	Flash write bandwidth	70MB/s
Power, spun down	2.5W	Epoch length	1 sec
Disk avg. seek time	3.5 ms	Spin down utilization thld.	0.10
Disk avg rot. latency	2.0 ms	Spin down time threshold	30 epochs
Disk transfer rate	120MB/s	Spin up utilization threshold	0.25
Spin up time	10.9 sec	Spin up time threshold	2 epochs
Spin down time	1.5 sec	Flash buffer size	16GB

Simulator Implementation. Our simulator uses block level I/O traces for its workloads. Traces must have four basic fields for each request: arrival time at the controller, read or write, start address, and size. For each request, the simulator progresses time up until the arrival time of the request, then maps the request or pieces of the request to appropriate disks and/or flash based on the current state of the system. When time has progressed to the point that a request at a disk (or flash) finishes, the controller is notified, and response time is taken to be the finish time of the last piece of a request minus the request's arrival time.

The default parameters for the simulator are shown in Table 1. The disk parameters are taken either from values measured in [16] or from the Hitachi Ultrastar 15K600 300GB enterprise drive spec sheet [8]. The flash parameters are taken from the Intel X-25M spec sheet [9].

Disks are not modeled in detail; access time is estimated as average rotational latency plus average seek time plus data transfer time. Sequential accesses are accounted for by removing rotational latency and seek time for accesses following the initial one. Flash access time is similarly modeled as flash read/write latency plus read/write transfer time, with latency removed for sequential accesses. Controller processing time is not accounted for, but is assumed to be insignificant.

Disk utilization is tracked in terms of epochs. Utilization is defined as the amount of time during an epoch a disk spent servicing requests, divided by the total length of the epoch. The threshold for a mirrored pair to transition to or from the low utilization state is set in terms of a utilization and a number of epochs. At the end of each epoch, the state of the system is evaluated to see if any disks should be transitioned to a different state. To transition to the low utilization state, a pair must be below the utilization threshold for the set number of epochs. A pair transitions to high utilization mode either when the utilization of the active disk exceeds its threshold for the set number of epochs, or the space used in the write buffer is above the buffer fill threshold.

Energy used by a hard disk during a trace is computed from the time spent in each of three states times the power used in each state. The states are spinning and serving requests, spinning but idle, and spun down. The energy used by the flash buffer is computed as the time spent idle and active times the power used in each state by the Intel X-25M. Energy savings for the whole system during a trace is computed as the percent difference in energy used when compared to a simulation with no energy savings enabled. The costs due to thermal power (i.e., cooling) are not considered but it is expected that spinning down disks will lead to additional energy savings because of reduced demand for running fans.

Table 2: QMD overall energy savings potential.

Trace file	Description	Length	Peak IO Rate	Energy Savings
cambridge-src1_1	enterprise server	7 days	3.5k/s	31%
cambridge-usr2	enterprise server	7 days	2.3k/s	28%
LiveMaps	global web service	1 day	4.7k/s	11%

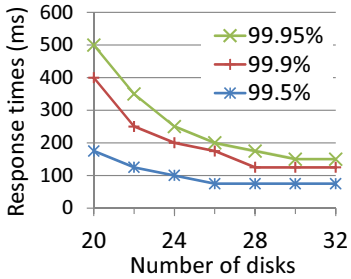


Figure 6: Impact of number of disks on response time (cambridge-src1_1)

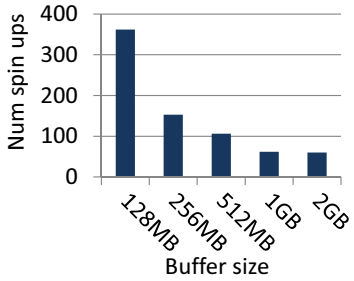


Figure 7: Number of spin ups over 7 days varying buffer size (cambridge-src1_1)

Real-World Traces. We use three real-world traces in our evaluation, as described in Table 2. The first two were recorded on servers in Microsoft Research Cambridge's enterprise data center, and we refer to them as cambridge-src1_1 and cambridge-usr2. The two traces were taken over seven days. They exhibit clear diurnal usage patterns, with peaks during the day and long low utilization periods between them. (The two traces are described in more detail in [12].) The third trace was taken from production Microsoft LiveMaps backend servers over one day. As LiveMaps is a global web service, the third trace shows no diurnal pattern and has very spiky activity throughout the day with only very short low utilization periods.

To evaluate QMD on a given trace, we first perform a range of simulations varying the number of disks without QMD enabled to determine the appropriate number of disks to use for this trace. The number of disks is chosen as the point at which increasing the number further provides minimal response time improvements. This is illustrated in Figure 6. The 99.5th percentile, 99.9th percentile, and 99.95th percentile response times are shown as the number of disks increases for the trace cambridge-src1_1. For this trace, we chose to use 28 disks for further experiments. We also use the simulation without QMD enabled to obtain a baseline energy usage which we use to determine the percent savings when QMD is enabled.

Overall Results. Our overall energy savings results for each trace are shown in Table 2. The energy savings ranges from about 31% in the best case to about 11% in the worst case. The cambridge-src1_1 trace benefits the most from QMD due to its strong diurnal usage pattern, and the cambridge-usr2 trace similarly has very long periods of low utilization. The LiveMaps server's trace has very

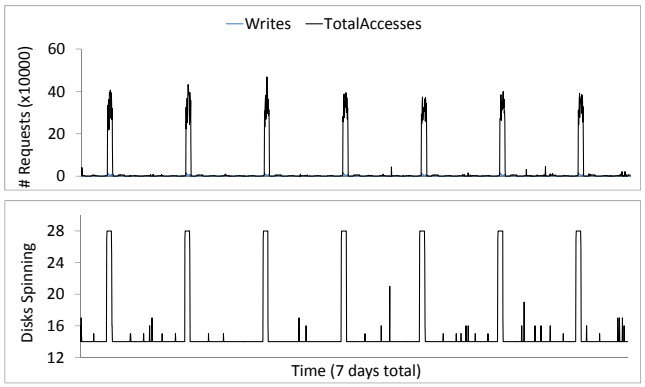


Figure 8: Cambridge-src1_1 - Total IO and writes above, average number of disks spinning below.

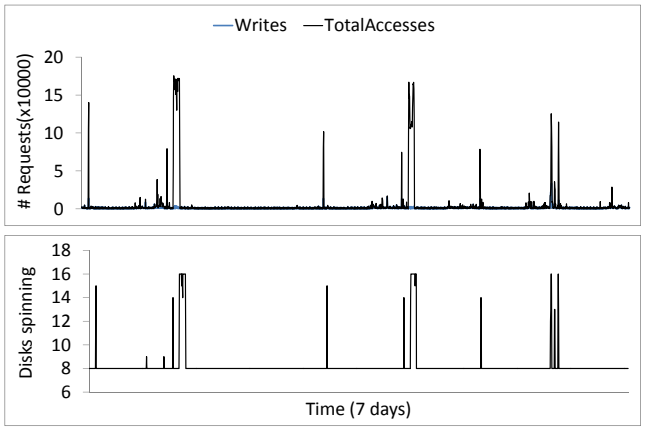


Figure 9: cambridge-usr2 - Total IO and writes above, average number of disks spinning below.

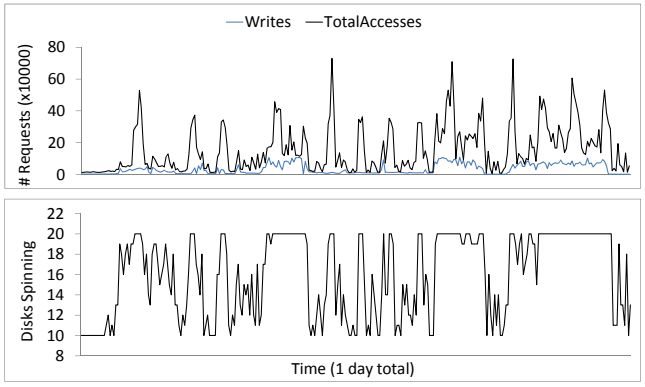
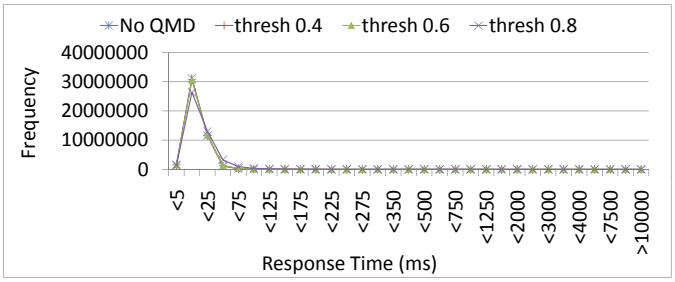


Figure 10: LiveMaps - Total IO and writes above, average number of disks spinning below.

short low utilization periods, and therefore it is much more difficult for QMD to be effective.

Figure 8, 9, and 10 compare I/O request arrival rates and the average number of active disks over the duration of the traces for the three traces, cambridge-src1_1, cambridge-usr2, and LiveMaps, respectively. The number of disks spinning ranges from 14 (half) during low utilization to 28 (all) during the peak utilization's. From the figures, we see that the spikes of the number of active disks correspond to the spikes in I/O arrival rates in Figure 8 and 9, indicating



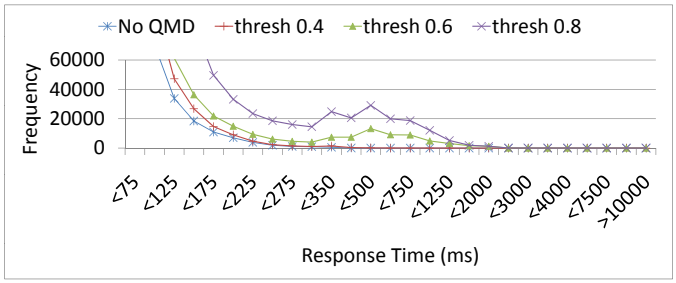
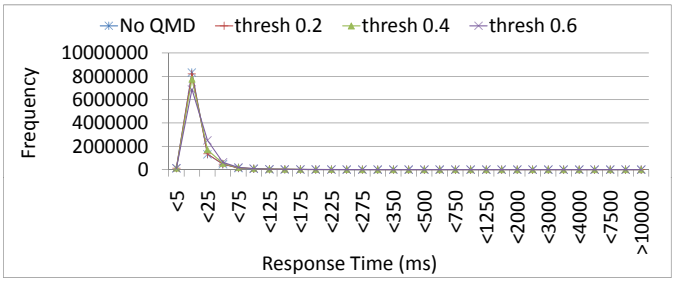


Figure 11: Cambridge-src1_1 - Overall response time distribution left, close up of tail end of distribution right, both shown while varying spin up utilization threshold.



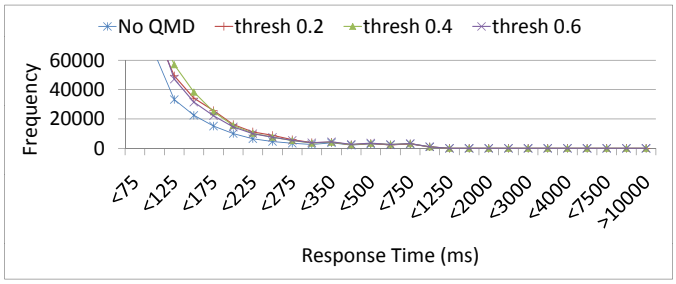
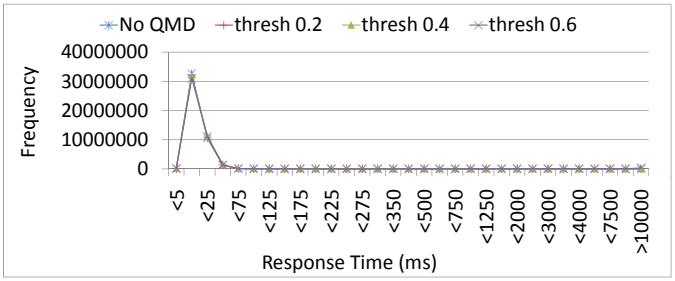


Figure 12: Cambridge-usr2 - Overall response time distribution left, close up of tail end of distribution right, both shown while varying spin up utilization threshold.



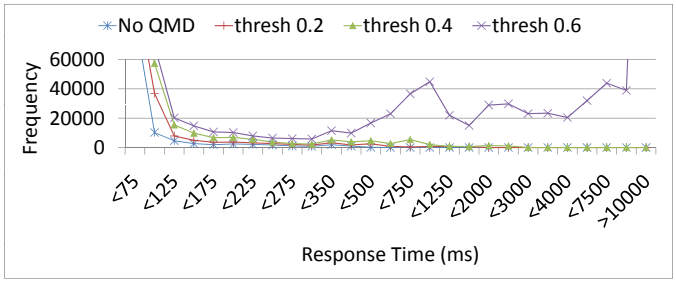


Figure 13: LiveMaps - Overall response time distribution left, close up of tail end of distribution right, both shown while varying spin up utilization threshold.

the effectiveness of QMD for the long periods of low utilization. In contrast, the LiveMaps trace sees bursty traffic with very short durations of low utilization periods, making it challenging to maintain response times and save energy at the same time.

Impact of Write Buffer Size on Spin-Up Cycles. Figure 7 shows the benefit, in terms of disk spin down/up cycles, as the buffer size increases for the trace cambridge-src1_1. epochs. At 128MB buffer size, each disk is spun down an average of 360 times over the seven day trace. As the buffer size is increased to 2GB the number of spin down cycles is reduced to 60 for 7 days, about an 83% reduction. This meets our goal for a five year HDD lifespan.

Impact of QMD on I/O Response Times. Figures 11, 12, and 13 show the effect of QMD on the response time distributions for all three traces. Each figure shows histograms of response time distributions. The horizontal axis is the upper end of response time histogram buckets, and the vertical axis is the number of requests in each bucket. Each figure shows the response time distribution for the original case where QMD is not enabled, and three curves for QMD while varying the spin up utilization threshold.

The left graphs show the overall response time distributions. We see that there is almost no variation at this scale. This means that QMD has little impact for the majority of I/O requests. The right graphs examine more closely the tail end of the distributions. We see that if the spin up utilization threshold is set too high, the scheme can significantly increase the number of requests with very high response times for some workloads (e.g. threshold 0.6 in Figure 13). When the spin up utilization threshold is too high, disks wait too long to start spinning up as I/O arrival rate increases. The active disks get overloaded, and there are deep queues by the time the standby disks have been spun up. However, it is also clear that we can conservatively choose the spin up utilization threshold (which we did for Table 2) so that the response time distribution with QMD enabled almost perfectly follows the distribution with no QMD.

4.2 Opportunity Study for Flushing Operation on Real Machine

Figure 14 shows the real machine experimental results on serving incoming requests while flushing buffered data blocks using the flushing algorithm in Figure 4. We ran the experiments on a Dell PowerEdge 1900 server equipped with two Quad-core 2.0GHz Intel Xeon E5335 CPUs and 8GB memory running 64-bit Ubuntu 9.04

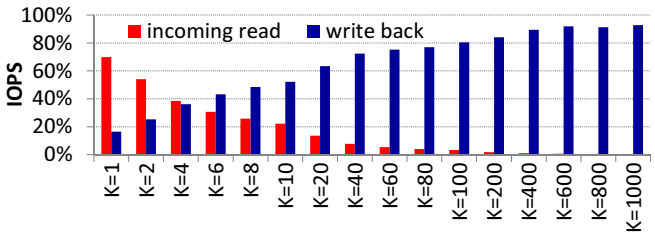


Figure 14: Opportunity study for serving incoming reads while flushing buffered writes as in the flushing algorithm in Figure 4.

server with Linux 2.6.28-17 kernel. We used a dedicated Seagate Barracuda ES.2 [21] enterprise-class drive (750GB, 7200 rpm) as the target disk. To model the buffered data blocks, we randomly generate 100,000 target addresses in the entire disk capacity range. We modeled the worst case for incoming requests: Every iteration in the algorithm processes an incoming read request. The read target addresses are randomly generated. All disk reads and writes are 8KB sized. Every experiment performed all the 100,000 writes.

We vary the number of write backs per iteration (i.e. parameter K) on the X axis in Figure 14. On the Y axis, we report IOs per second normalized to the read-only and write-back-only performance, respectively. We see that as K increases, write-back performance increases while read performance decreases gracefully. For example, at $K = 2$, the algorithm achieve 54% of the peak read and 25% of the peak write performance. At $K = 10$, the algorithm achieve 25% of the peak read, and 48% of the peak write performance. Our algorithm schedules write backs whose target addresses are close to every read request for reducing disk seek overhead. Using the measurements, a storage system can choose a K that balances the write-backs and incoming request handling for given targets of system loads and write-back times.

5. RELATED WORK

Exploiting Redundancy for Saving Disk Energy. As described in Section 1, several previous studies proposed to exploit redundancy in data storage systems to save energy. EERAID [11] and RIMAC [24] exploited redundancy in RAID arrays for saving energy. They focused on request scheduling and storage cache designs. Diverted Access[16] exploited redundancy in a more general form, where storage systems store (encoded) data in n fragments, and every subset of m fragments can be used to reconstruct the data. However, these previous proposals suffer from two problems: (i) Limited NVRAM capacity forces disks to be frequently spun up/down, which impacts their lifetimes; and (ii) there is still a significant gap to achieve the ideal goal of energy proportionality. In this paper, we exploited flash as a larger non-volatile write cache to address (i). For (ii), we proposed partial online mirror and a collaboration interface between storage systems and upper-level software for spinning down more disks while limiting performance impacts.

Migrating or Copying Data for Saving Disk Energy. MAID [6] maintains a large number of disks in standby mode for archival purpose. A disk is spun up on demand for serving a request. To reduce the accesses to standby disks, MAID uses a small number of disks to cache recently used data. PDC [15] migrates frequently used data to a subset of all the disks so that other disks can be spun down. In this paper, we exploit both redundancy and data migration for achieving energy proportionality. In addition, we propose to expose energy state information to upper-level software (e.g., database systems) so that they can collaborate to hide the spin-up delays for accessing the cold data.

Exploiting Flash as Write Buffers. Schindler et al. proposed to use flash as a write cache to optimize storage access patterns that consist of large sequential reads and random writes [17]. Chen et al. exploited a flash-based cache for enabling online updates in data warehouses [3]. While both studies maintain the efficiency of sequential reads in face of random writes, their focuses are quite different. Chen et al. developed a MaSM algorithm, for supporting fine-grain database record updates, minimizing memory footprint and flash writes, and supporting ACID properties [3]. On the other hand, storage systems see only block-sized I/Os and do not require ACID, which simplify the flash management in Schindler et al.'s solution. Instead, they focus on efficiently migrating the cached writes back to disks in the middle of large sequential read operations [17]. In this paper, we also exploit flash as a write cache, but for a very different purpose: significantly increasing the size of the non-volatile write cache for reducing the number of spin-up/down cycles for disks in energy-efficient disk arrays.

Efficient Disk Access Patterns. Sequential accesses and random accesses are the two access patterns that are studied most for disks. Because of their mechanical properties, HDDs achieve peak bandwidth for sequential access patterns but have poor performance for random accesses. In between these two extremes, previous work studied other efficient access patterns for modern disks. Schlosser et al. exploited the semi-sequential pattern and short seeks for multidimensional data layout on disks [18]. On modern disks, short seeks up to a certain number of disk tracks take similar time. A list of disk blocks on different disk tracks satisfies the semi-sequential pattern if the next block on the list can be accessed without rotational delay after seeking from the previous block. Combining these two features, Schlosser et al. identified that from any given disk blocks, there is a set of disk blocks, called adjacent blocks, that can be accessed with equally small cost. Then, they placed multidimensional data using adjacent blocks for efficient accesses in every dimension. Schindler et al. studied proximal I/Os for combining random writes into large sequential reads [17]. Proximal I/Os are a set of I/O requests with addresses close to one another. Modern disks can handle these I/Os with minimal seek overhead. Similar to both of the studies, we also aim to reduce disk seeks by scheduling I/Os with close addresses, but for a quite different workload: flushing a large number of buffered writes to disks while serving (random) incoming requests. Our proposal balances the two activities, and supports performance tuning based on a simple parameter, the number of write backs performed between two incoming requests.

6. CONCLUSION

In this paper, we investigated energy efficiency for HDD-based data storage in general and RAID systems in particular. We proposed QMD (Quasi Mirrored Disks) to effectively leverage redundancy in storage systems and flash to save significant energy. We demonstrated this through simulation using real-world workloads, including systems that experience long periods of low utilization, i.e. due to diurnal usage patterns. With a sufficiently large non-volatile write buffer, the number of spin down cycles for disks can be kept within the average lifetime limit specified by manufacturers. Moreover, we can choose conservative parameters (e.g., spin-up utilization threshold) so that QMD can achieve the energy savings with negligible impact on I/O response times.

We find that exploiting redundancy alone still leaves a big gap to the energy proportionality goal. As future research, we propose two interfaces that allow applications and storage systems to collaborate for further saving energy, and Partial On-line Mirrors for effectively taking advantage of the interfaces.

7. ACKNOWLEDGMENTS

The QMD simulator was initially developed by Katlyn Daniluk as part of her research experience for undergraduates. This work was partially supported by National Science Foundation awards IIS-0746696 and IIS-1050301.

8. REFERENCES

[1] NSF workshop on sustainable energy efficient data management (SEEDM). http://seedm.org.

[2] NSF workshop on the science of power management. http://www.cs.pitt.edu/ kirk/SciPM2.

[3] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient online updates in data warehouses. In *SIGMOD*, 2011.

[4] L. A. Barroso. The price of performance: An economic case for chip multiprocessing. *ACM Queue*, pages 48–53, Sept 2005.

[5] L. A. Barroso and U. Holzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, Dec 2007.

[6] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *SC*, 2002.

[7] Emerson Network Power. Energy logic: Reducing data center energy consumption by creating savings that cascade across systems. http://emersonnetworkpower.com/en-US/Brands/Liebert/Documents/ White Papers/Energy Logic_Reducing Data Center Energy Consumption by Creating Savings that Cascade Across Systems.pdf.

[8] Hitachi Global Storage Technologies. Ultrastar 15k600 data sheet, Sept 2009.

[9] Intel Corporation. X-25M SATA SSD 34nm product specification.

[10] J. G. Koomey. Estimating total power consumption by servers in the U.S. and the world. http://enterprise.amd.com/Downloads/ svrpwrusecompletefinal.pdf.

[11] D. Li and J. Wang. EERAID: energy efficient redundant and inexpensive disk array. In *ACM SIGOPS European Workshop*, 2004.

[12] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. In *FAST*, 2008.

[13] NetApp Corporation. Flash cache. http://www.netapp.com/us/ products/storage-systems/flash-cache/flash-cache.html.

[14] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[15] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS*, 2004.

[16] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *SIGMETRICS*, 2006.

[17] J. Schindler, S. Shete, and K. A. Smith. Improving throughput for small disk requests with proximal I/O. *FAST*, 2011.

[18] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *FAST*, 2005.

[19] Seagate Technology LLC. Barracuda 7200.12 data sheet.

[20] Seagate Technology LLC. Cheetah 15K.4 SCSI product manual, rev. d edition, May 2005. Publication number: 100220456.

[21] Seagate Technology LLC. Barracuda ES.2 data sheet, 2009.

[22] Transaction Processing Performance Council. TPC benchmark E standard specification version 1.12.0.

[23] US Environmental Protection Agency. Report to congress on server and data center energy efficiency: Public law 109-431.

[24] X. Yao and J. Wang. RIMAC: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. In *EuroSys*, 2006.

A Case for Micro-Cellstores: Energy-Efficient Data Management on Recycled Smartphones*

Stavros Harizopoulos
HP Labs
Palo Alto, CA, USA
stavros@hp.com

Spiros Papadimitriou
Google Research
Mountain View, CA, USA
spapadim@gmail.com

ABSTRACT

Increased energy costs and concerns for sustainability make the following question more relevant than ever: can we turn old or unused computing equipment into cost- and energy-efficient modules that can be readily repurposed? We believe the answer is yes, and our proposal is to turn unused smartphones into micro-data center composable modules. In this paper, we introduce the concept of a Micro-Cellstore (MCS), a stand-alone data-appliance housing dozens of recycled smartphones. Through detailed power and performance measurements on a Linux-based current-generation smartphone, we assess the potential of MCSs as a data management platform. In this paper we focus on scan-based partitionable workloads. We show that smartphones are overall more energy efficient than recently proposed low-power alternatives, based on an initial evaluation over a wide range of single-node database scan workloads, and that the gains become more significant when operating on narrow tuples (i.e., column-stores, or compressed row-stores). Our initial results are very encouraging, showing efficiency gains of up to 6×, and indicate several promising future directions.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

1. INTRODUCTION

Modern smartphones have the computational power of a 5-year-old PC, but at a fraction of the size and energy consumption (110x smaller volume than a standard 1U server, and 200x less peak power). More than 1 billion cellphones are shipped yearly; in 2010, according to IDC, over 300 million of those were smartphones (a 74.4% increase over 2009). Smartphones have a typical consumer refresh cycle of two to three years. Over the next few years, we expect a total of one billion smartphones to become obsolete; the aggregate computational power of these phones is similar to that of *all* 500 top supercomputers in the world combined—but at a fraction of their energy needs. How can this power be harnessed?

A place where cost- and energy-efficient computing units could be utilized at large numbers is a modern data center. Data center operating costs are characterized by a continuously growing energy cost component [5, 2]. Power and cooling costs are soon expected to surpass the (amortized) cost of purchasing servers. Demand for new and bigger data centers is on the rise, fueled by both consumer and enterprise applications. However, could a significantly underpowered device support applications that typically run on high-end servers? In this paper, we argue that for certain classes of enterprise data management problems, such as data warehousing and analytics, there are several emerging trends that lend themselves to a micro-data center design based on underpowered hardware (also known as "wimpy nodes" in the literature [3]). These trends are (a) MPP-style processing (massively parallel processing), (b) column-oriented and compressed data which ease pressure on the memory/network buses, and (c) offering reliability through replication instead of expensive hardware solutions.

Our proposal is to repurpose old or unused smartphones and use them to assemble units, called Micro-Cellstores, that contain dozens of interconnected smartphones which collectively act as a data-appliance mini-cluster. A concept dia-

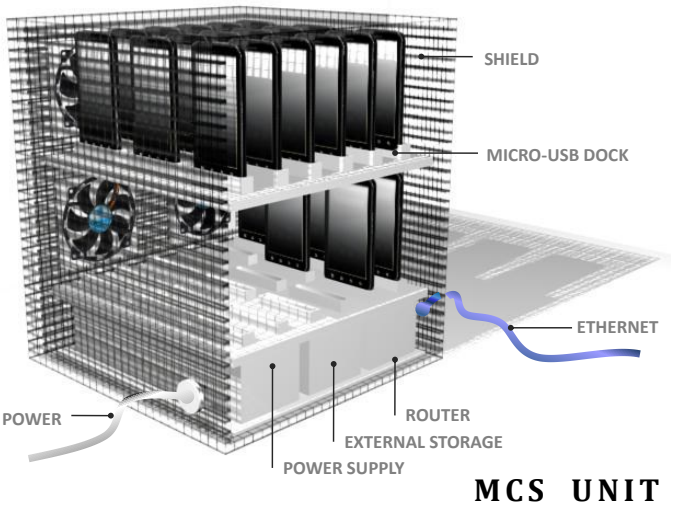


Figure 1: Micro-Cellstore Architecture.

*The views contained herein are the authors' only and do not necessarily reflect the views of Hewlett-Packard or Google.

DaMoN 2011

gram of a Micro-Cellstore is shown in Figure 1. There are several interesting questions around architecting and manufacturing Micro-Cellstores that are beyond the scope of this paper, such as: What are viable methods for networking? How can batteries and power management features be leveraged? What is the right ratio of phones, routers/hubs, and external storage? Is cooling a problem?

Our focus in this paper is exploring the types of data management workloads that can be efficiently supported by MCS units, and compare the energy-efficiency of appliances based on smartphones (ultra-wimpy nodes) against other low-power alternatives. Our contributions are the following:

- Detailed power-profile characterization of a modern smartphone (Nexus S, released in Q4 2010).
- Power efficiency measurements for partitionable, scan-intensive database workloads on smartphones and two types of wimpy platforms.
- Introducing the case for Micro-Cellstores (MCS).

The rest of the paper is organized as follows. In Section 2 we cover related work, including recent proposals for "wimpy" architectures. Section 3 motivates Micro-Cellstores and Section 4 details the characteristics of modern smartphones. Section 5 carries out our benchmarking and analysis of various single-node, scan-based database workloads. We conclude in Section 6.

2. RELATED WORK

Energy concerns are important enough to often dictate where data centers are built. A growing number of efforts to improve the energy efficiency of clusters and data centers include holistic redesigns that treat a data center as a single computer [4, 16], cluster workload consolidation to meet power constraints and reduce energy requirements [15, 13, 12], and considerations of low-power architectures [3, 19]. In this section we briefly discuss recent efforts in improving the energy efficiency of database applications.

Energy efficiency in databases. Traditionally, database systems have been optimized for performance, ignoring power-related costs. However, the proliferation of scale-out architectures has forced data management systems to consider energy as equally important to performance. Early research studies argued for the redesign of several key components such as the query optimizer, the workload manager, the scheduler and the physical database design [7, 11, 9, 20]. Many of these suggestions assumed that, like cars, computer systems had different optimal performance and energy efficiency points. However, a subsequent detailed study on the energy efficiency of a single database server [18] found that, because of the start-up power draw, the highest performing configuration was also the most energy efficient. That study did not consider multi-node configurations or low-power hardware. In this paper, we investigate the latter.

Non-server architectures. In an effort to improve the energy efficiency of clusters, a number of studies have also considered the use of low-power "wimpy" nodes consisting of low-power storage (SSDs) and processors (mobile CPUs) [3, 19, 14, 17]. Primarily, these designs target computationally "simple" data processing tasks that are extremely partitionable, such as key-value workloads [3]. For such workloads, wimpy clusters were shown to be more energy efficient compared to traditional clusters built using more power-hungry server nodes. However, this result may not hold in scenarios such as database workloads which often exhibit sub-linear scale-up characteristics, especially when full cluster cost is considered [14].

Our work in this paper can be viewed as in the same category as the above-mentioned wimpy-node architectures. To our knowledge, we are the first to characterize the energy-efficieny of modern smartphones when running database-style tasks. Throughout this paper, we define energy efficiency of a workload as the ratio of the query completion rate (e.g., scans per hour) to the average power consumed by the system.

3. A CASE FOR MICRO-CELLSTORES

There are three main emerging trends in enterprise data management that lend themselves to a micro-data center design based on underpowered hardware:

- Massively Parallel Processing (MPP): Parallel DBMSs typically adopt the shared-nothing paradigm for scaling out (rather than scaling up) to deal with increasingly larger data volumes. For queries that scale linearly with the number of nodes in a cluster, an underpowered cluster could reach acceptable performance levels by using more nodes.
- Column-oriented and highly compressed data: Column-stores have emerged as the prevalent architecture for high-performance data management. Operating on columnar, highly compressed data eases pressure on the memory and network/IO buses (which are typically under-specced in a smartphone, due to concerns over manufacturing cost).
- Reliability through replication: Modern systems increasingly rely on replication for providing reliability, rather than on expensive hardware-based solutions. Such techniques are particularly suitable for smartphones which do not compare well to server-grade components with respect to reliability.

Furthermore, recent work has demonstrated running MapReduce jobs on a network of smartphones [6]. Micro-Cellstores are inspired by the above observations, combined with the expected abundance of used smartphones in the future (as explained in the introduction).

In the concept of Figure 1, the proposed housing structure contains standardized micro-USB connectors and, possibly, WiFi routers for connectivity. We also expect that there will be some form of storage directly connected to the router/hub. Furthermore, batteries may be leveraged in interesting ways, e.g., to provide uninterrupted operation even under intermittent power availability, to charge during off-peak hours at possibly cheaper rates, or to smooth out the cluster's power profile. Studying the tradeoffs between the different types of networking, deciding the best use for external storage, and exploring ways to harness the batteries are beyond the scope of this paper.

We expect MCSs based on cheaply acquired, used smartphones to be environmentally sustainable, minimizing total exergy cost. While our primary metric for efficiency in this paper is power consumption, it should be noted that cost-efficiency may have a favorable impact on our proposal, since

Year	Model	CPU	RAM	Storage (int./ext.)	WiFi
1996	Nokia 9000	33MHz AMD Elan x486	2MB	6MB	–
2002	Sony P800	156MHz ARM9	?	16MB	–
Q2 2007	iPhone	412MHz[1] ARM	128MB	4, 8, or 16GB	b/g
Q4 2008	HTC Dream	528MHz MSM7201A (ARM11)	192MB	256MB / microSD	b/g
Q2 2009	iPhone 3GS	600MHz[2] S5PC100 (Cortex-A8)	256MB	8, 16, or 32GB	b/g
Q4 2009	Motorola Droid	550MHz[3] OMAP3430 (Cortex-A8)	256MB	512MB / microSD	b/g
Q1 2010	Nexus One	1GHz QSD8250 (Snapdragon)	512MB	512MB / microSD	b/g/n
Q2 2010	iPhone 4	1GHz[4] Apple A4 (Cortex-A8)	512Mb	16 or 32GB	b/g/n
Q4 2010	Nexus S	1GHz S5PC110 (Hummingbird)	512MB	16GB iNAND	b/g/n
Q1 2011	HTC Thunderbolt	1GHz MSM8655 (Snapdragon)	768MB	8GB / microSD	b/g/n
Q2 2011	Droid Bionic	1GHz **dual-core** Tegra 2	512MB	2GB / microSD	b/g/n
Q2 2011	Galaxy S II	1GHz **dual-core** Exynos412 or Tegra 2	1GB	16 or 32GB	a/b/g/n

Table 1: Smartphone model feature summary.

smartphones would be otherwise discarded. A broader goal of this paper is to increase awareness of environmentally sustainable solutions for computing infrastructures, and the MCS concept is aimed towards that end. The rest of the paper focuses on specific aspects of the suitability of MCSs (which consist of ultra-wimpy nodes) for database workloads.

4. MODERN SMARTPHONES

Although the concept of what we today recognize as a "smartphone" is almost two decades old [1], until very recently the dominating characteristic of a "smartphone" was the "phone." Functionality was rather rudimentary and computing power was limited. The fairly recent explosion in the availability of reasonably fast wireless data networks has spurred demand for more capable computing devices, and vice versa, creating a virtuous cycle.

The current concept of a smartphone as an always-connected computing device that runs sophisticated applications was brought into the mainstream by the Apple iPhone, which was released four years ago. Since then, new, more powerful models are constantly introduced. Table 1 summarizes some key features of various smartphone models. Especially during the past two years, the smartphone space has witnessed exponential growth. Both CPU clock speeds and RAM capacity have roughly doubled in that time. Figure 2 illustrates the clock and memory trends over time.

In 2011, several companies are expected to introduce smartphones with dual-core CPUs. Furthermore, this trend does not show signs of slowing down. The ARM Cortex-A9 core design, on which these planned devices are based, supports up to four cores on the same chip and clock speeds up to 2GHz. Furthermore, since they are aimed at mobile devices, these designs focus on maintaining power consumption characteristics while increasing performance. Therefore, power efficiency should increase even further over time.

4.1 Experimental methodology

We collected measurements on a Samsung/Google Nexus S smartphone, running Android 2.3.3 (GRI40, with Linux kernel 2.6.35). We wrote logging software that records the following:

- Battery statistics, including battery level (%) and voltage, by listening for `BATTERY_CHANGED` broadcast events.
- CPU load statistics, by polling `/proc/stat` at a user specified interval (by default 60 seconds).

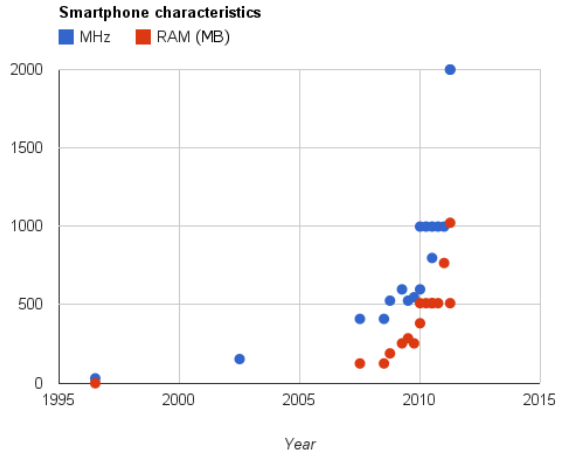


Figure 2: Smartphone clock speed and RAM over time.

- CPU frequency scaling statistics, by polling `/sys/devices/.../time_in_state`.
- Network connectivity changes, by listening for `CONNECTIVITY_ACTION` broadcast events.
- Network traffic statistics per interface, by polling `/proc/net/dev` (120 second interval, by default).
- Screen usage statistics, by listening for `SCREEN_ON` and `SCREEN_OFF` broadcast events.

Each logger is a separate component (Android service) that can be turned off when not needed. For each experiment, we only collect the statistics we need. Furthermore, we ran the device with logging fully enabled and compared baseline power consumption (see below) with logging fully disabled, and saw no measurable effect.

Log events are queued in memory and flushed to phone storage in batches (user parameter, typically 20 events). Statistics collected through broadcast event receivers are "pushed" only when something changes. For polled statistics we used Android alarm APIs. We kept logging to the minimum necessary and verified that it has no measurable effect on power consumption by comparing battery level drop with logging on and off, over a period of several hours.

During each experiment we acquired a *partial wakelock*, which prevents the CPU from sleeping (otherwise the O/S may power down the CPU when there is no user interaction,

even if processes are running). Beyond that, we kept the screen off, disabled all radios (cellular, WiFi, Bluetooth, and GPS) by setting the phone in airplane mode, and disabled all background services. Finally, before each run we charged the battery normally (i.e., no bump charging).

Since measuring battery capacity is difficult without specialized equipment, we used a fresh battery for our experiments. We converted ampere-hours to watt-hours using average voltage (time-weighted) during each experiment, based on battery voltage sensor values (typical range was 4 ± 0.03V). Android reports battery levels as integer percentages of capacity. On the Nexus S, a 1% drop corresponds to 15mAh or about 60mWh. We ensured that each experiment ran for at least one hour (much longer for idle power measurements), which implies an error of at most 10% (typical experiment power consumption was 0.7–1.2W).

The Hummingbird CPU in Nexus S uses dynamic frequency scaling (DVFS), supporting clock rates of 100, 200, 400, 800MHz and 1GHz. Effective clock rates were estimated by averaging frequencies, weighted by the fraction of time spent at each frequency based on O/S statistics in `/sys`. All experiments reported in this paper are with DVFS turned on, using the `ondemand` governor. This configuration is the most power-efficient overall. For CPU-bound workloads, the clock was indeed at or near its maximum. However, for disk-bound workloads, we observed that the O/S successfully scaled the CPU down to the minimum frequency that can handle the load (details ommited for space).

4.2 Characteristics

Table 2 summarizes some characteristic performance numbers for Nexus S. Sequential read bandwidths were measured by repeatedly reading a large enough array or file. Peak CPU power consumption was estimated at full load and 1GHz. Main memory and external storage bandwidths (361 and 25 MiB/s) are comparable to those of a mid-grade laptop (571 and 33 MiB/s; Intel SU9400 1.4GHz CPU and 500GB, 5400RPM drive). However, power consumption is an order of magnitude smaller (SU9400 TDP is 10W).

Description	Value
Memory read rate	361.9 ± 57.8 MiB/s
Storage read rate	24.8 ± 1.7 MiB/s
USB transfer rate	13.3 ± 0.8 MiB/s
CPU peak power	1090 ± 80 mW

Table 2: Nexus S characteristics.

We observed that CPU power consumption is effectively proportional to clock frequency [21] over a wide range of clock speeds. Figure 3 shows average power consumption versus effective clock frequency, with everything except the CPU turned off. Because the intercept is non-zero, power efficiency (mW/MHz) increases with clock rate.

5. ANALYSIS

Our goal in this section is to explore which types of data management workloads are best suited for MCS units, and compare the power consumption of smartphones to that of other, energy-efficient architectures, when running a range of parameterized workloads.

In previous work [18] we showed that, for a single-node DBMS server, the most energy-efficient configuration is typically the fastest one. In follow-up work-in-progress [10] we found that the same holds for low-power non-server architectures, such as laptops and desktops, and that laptops can be more energy efficient than servers for several types of database engine operations, such as scans, sorts, and joins. Therefore, for the purposes of this paper, we only compare against two energy-efficient platforms: a mini-desktop and a laptop with an Ultra Low Voltage (ULV) processor.

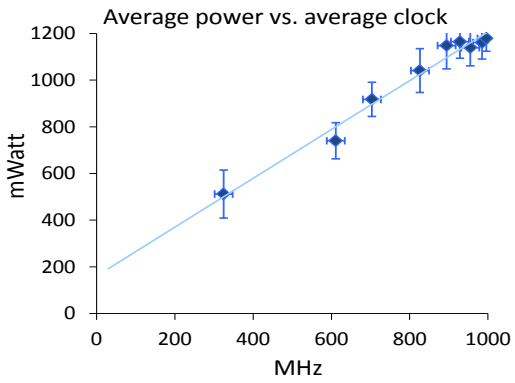


Figure 3: Averge clock frequency vs. power consumption.

In Section 5.1 we discuss what data management workloads are a natural fit for MCS units; in Section 5.2 we present the experimental setup and our results. We offer implications of our results in the concluding section.

5.1 Workload suitability for MCS

Low-power (or "wimpy") architectures trade single-node computational speed for higher energy efficiency. Compared to a server node, a task will run significantly slower on a wimpy node, but it will also consume much less energy. To make up for slower individual nodes, wimpy archictectures are typically positioned to run scale-out software infrastructure, with many more nodes than a server-based installation. For throughput-intensive tasks that can scale linearly with node count, this strategy is a win. Using more wimpy nodes increases total throughput, without changing energy efficiency (power and performance increase at the same rate). That result was also verified experimentally [19].

Recent work, however, pointed out that several complex parallel DBMS workloads exhibit sub-linear scalability, and therefore the energy efficiency of wimpy-node architectures may degrade as node count increases [14]. That analysis used total cost for purchasing and operating servers over a period of several years and showed that server-based clusters can be more cost-efficient than wimpy clusters, depending on the complexity of the workload. In our context, the final cost of an MCS unit is not clear, as it will depend on whether recycled phones come with a price tag. Thus, we compare only operating costs, i.e., energy efficiency.

The performance and power characterization of Nexus S from the previous section pointed that, while smartphones are extremely low-power devices, disk and network I/O speed as well as RAM size can be up to two orders of magnitude less than servers, or one order of magnitude less than previously proposed wimpy architectures. Therefore, we are interested in workloads with the following properties:

- Partitionable across a large number of nodes.
- Minimal network transfer.

Figure 4: Energy consumption per GB (lower is better), for a wide-tuple scan on four different configurations (Nexus S always operates on disk data).

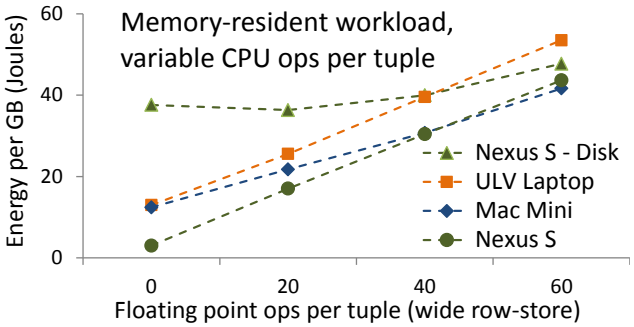


Figure 5: Energy consumption per GB (lower is better), for increasingly compute-intensive scans.

- Favor increased CPU processing cycles per byte read from disk.
- Do not require large in-memory structures.

Workloads that have the above properties include database scans that may be followed by additional (partitionable) operations (such as filtering, projection, aggregation) and certain types of joins that can run in a single-pass and involve small network transfers (e.g., when the inner table shuffled across all nodes also includes a highly selective predicate). In the rest of this paper, we focus on database scans without network transfers, with varying tuple widths (covering row/column-stores with lightweight compression), and with varying degree of CPU processing per tuple.

5.2 Experimental setup and results

We used the database storage manager developed in [8] to run a series of database scans. This is a block-iterator engine that can operate on both row- and column-oriented data. We ran the same C++ code on all three platforms. On Android we used the Native Development Kit (NDK, release R5b). We experimented with the `LINEITEM` table from TPC-H, using the same simplifications as in [8].

We compare Nexus S to two low-power systems: a 2010 Apple Mac Mini and an HP Compaq 2710p Tablet laptop. The Mac Mini features an Intel Core 2 Duo processor, whereas the HP Laptop features an Ultra Low Voltage version of the same processor. Both systems have 2GB RAM and relatively slow disks: a 40MB/sec SATAII HDD in the Mac Mini and a 80 MB/sec PATA SSD in the HP Laptop (we also report projected results based on faster disks). Table 3 summarizes configuration and power consumption details for all three systems. "Idle" is the power consumption at zero load. For the Mac Mini and the Laptop we used a Brand Electronics 20-1850 CI to measure total system power. This power meter has ±1.5% accuracy and collects readings once a second. Each experiment was repeated multiple times to get stable power measurements.

System	CPU (cores)	Power	Disk
Mac Mini	2.4GHz (2)	7.1W – 26.8W	41MB/sec
Laptop	1.2GHz (2)	11.1W – 23.7W	78MB/sec
Nexus S	1GHz (1)	0.2W – 1.17W	25MB/sec

Table 3: CPU specs and idle/peak power consumption of tested systems.

In our first experiment, we used all systems in an out-of-the-box configuration, to scan `LINEITEM` from disk, using a row-store representation (tuple width is fixed at 144 bytes). The results are in Figure 4 (leftmost part). This figure shows energy per GByte of data read (lower is better) for four different configurations of the two Intel systems. The rightmost configuration corresponds to the same scan when the table fits entirely in memory, whereas for the two middle configurations we recompute the consumed energy assuming two faster disks: one with 250MB/sec bandwidth and one with 500MB/sec. For the Nexus S we always show the energy consumption for disk-resident data.

Figure 4 shows that Nexus S consumes significantly less energy than the wimpy platforms for a row-store wide-tuple scan in an out-of-the-box configuration. The Mac Mini and ULV Laptop perform very poorly because they have fairly slow disks (thus running time is high) while paying a substantial up-front penalty for idle power (for example, the Mac Mini operates at 9.9–11.4W out of a 7.1–26.8W range). However, under more realistic assumptions about typical disk configurations, the wimpy nodes become competitive. When the wimpy platforms operate on memory-resident data, they consume less energy than Nexus S reading from disk.

For the remaining experiments we always show main-memory measurements for the two Intel systems, as the best-case scenario for those systems.

Next, we keep the tuple width the same, but experiment with increased number of computations per tuple. Figure 5 shows energy per GByte of data read (lower is better) for varying computation intensity. For this experiment we modified the code, by injecting a number of floating point operations to emulate worst-case processing for each tuple (e.g., complex analytic workloads). Data set size and tuple width are fixed for all runs. For the Nexus S we show both disk- and memory-resident performance.

As Figure 5 shows, the Nexus S consumes less energy than the other two systems when operating on memory-resident data. However, the gap closes when there are tens of floating operations executed in-between tuple reads. The energy consumption of Nexus S on disk-resident data converges slowly to the consumption with memory-resident data. In this experiment we wanted to show a large range of possible operations (typical per-tuple transformations correspond to a few floating operations).

In our last experiment, we compare energy efficiency when evaluating a single predicate using four different table storage representations: wide row-store (144 bytes), narrow row-store (32 bytes), wide column-store (8 bytes), and narrow column-store (1 byte). In all cases we use data from `LINEITEM` to create various projections of different width. The narrow

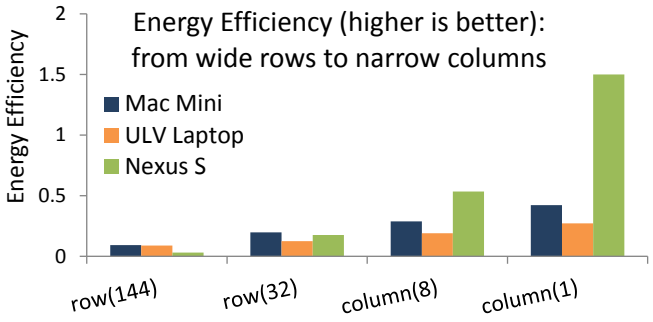


Figure 6: Energy efficiency (higher is better) for varying tuple widths and storage formats.

versions of the row and column tuples are also representative of compressed versions of the original tuples, as the additional overhead of lightweight compression is small [8]. Because we do not know how memory capacities will evolve, we compare the best-case scenario for the two Intel systems (memory-resident data) with the worst-case scenario for the Nexus S (disk-resident data); for memory-resident data, the efficiency of the Nexus S increases further by 1.2–2.6×.

Figure 6 shows the results. This time we compute the overall energy efficiency (higher is better) on the y-axis. While the two Intel systems always operate close to maximum CPU utilization, the Nexus S starts as disk-bound (for row-144) and becomes cpu-bound after row-32. For column-store scans, the Nexus S is significantly more energy-efficient than the other two systems—up to 6×, despite operating on disk-resident data. In accordance to [18], all systems become more efficient as they ran faster.

6. CONCLUSIONS

In this paper, we introduced the concept of a Micro-Cell-store (MCS) unit, a data appliance consisting of recycled smartphones. Through detailed power and performance measurements on a Linux-based current-generation smartphone, we assesed the potential of modern smartphones as a building unit for energy-efficient database appliances. Our results confirm that smartphones are overall a more energy efficient alternative, and further show that the gains become more significant for narrow tuples (i.e., column-oriented stores, or compressed row stores), achieving up to 6× improvement even when compared against other low-power options.

Our intention with the ideas presented in this paper is to motivate environmentally sustainable approaches, based on reusing and repurposing computing equipment. Towards this goal, there are several open questions around architecting MCS units and developing purpose-built data management software: How can total cost be competitive to that of traditional data centers? What other workloads could possibly run on this platform? What are the limits of scaling out data management tasks on wimpy or ultra-wimpy node architectures? Is there any benefit in combining MCS units with traditional servers to form hybrid data centers? If yes, what software changes would be needed? We hope these questions will motivate the research community to intensify their efforts on energy-efficient solutions.

7. REFERENCES

[1] IBM Simon. `http://en.wikipedia.org/wiki/IBM_Simon`.

[2] Report To Congress on Server and Data Center Energy Efficiency. In *U.S. EPA Tech. Report*, 2007.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *SOSP '09*, 2009.

[4] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12), 2007.

[5] C. Belady. In the Data Center, Power and Cooling Costs More than the IT Equipment it Supports. *Electronics Cooling*, 23(1), 2007.

[6] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos. Misco: A MapReduce framework for mobile systems. In *PETRA*, 2010.

[7] G. Graefe. Database Servers Tailored to Improve Energy Efficiency. In *Software Engineering for Tailor-made Data Management*, 2008.

[8] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[9] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy Efficiency: The New Holy Grail of Database Management Systems Research. In *CIDR*, 2009.

[10] W. Lang, S. Harizopoulos, M. A. Shah, J. M. Patel, and D. Tsirogiannis. Improving the Energy Efficiency of a DBMS Cluster. In *Submitted for publication*, 2011.

[11] W. Lang and J. M. Patel. Towards Eco-friendly Database Management Systems. In *CIDR*, 2009.

[12] W. Lang and J. M. Patel. Energy Management for MapReduce Clsuters. In *VLDB*, 2010.

[13] W. Lang, J. M. Patel, and J. F. Naughton. On Energy Management, Load Balancing and Replication. In *SIGMOD Record*, 2009.

[14] W. Lang, J. M. Patel, and S. Shankar. Wimpy Node Clusters: What About Non-Wimpy Workloads? In *DaMoN*, 2010.

[15] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower*, 2009.

[16] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "power" struggles: coordinated multi-level power management for the data center. *SIGOPS Oper. Syst. Rev.*, 2008.

[17] A. S. Szalay, G. C. Bell, H. H. Huang, A. Terzis, and A. White. Low-power amdahl-balanced blades for data intensive computing. *SIGOPS Oper. Syst. Rev.*, 2010.

[18] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD '10*, 2010.

[19] V. Vasudevan, D. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient cluster computing with FAWN: workloads and implications. In *e-Energy '10*, 2010.

[20] Z. Xu, Y.-C. Tu, and X. Wang. Exploring Power-Performance Tradeoffs in Database Systems. In *ICDE*, 2010.

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISS*, 2010.