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

Darius Šidlauskas
Aalborg University
darius@cs.aau.dk

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

Simonas Šaltenis
Aalborg University
simas@cs.aau.dk

ABSTRACT

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

1. INTRODUCTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. IMPLEMENTATION

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

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

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

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

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

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

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

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

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

2.2 Implementation Overview

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

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

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

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

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

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

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

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

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

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

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

3. EXPERIMENTS

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

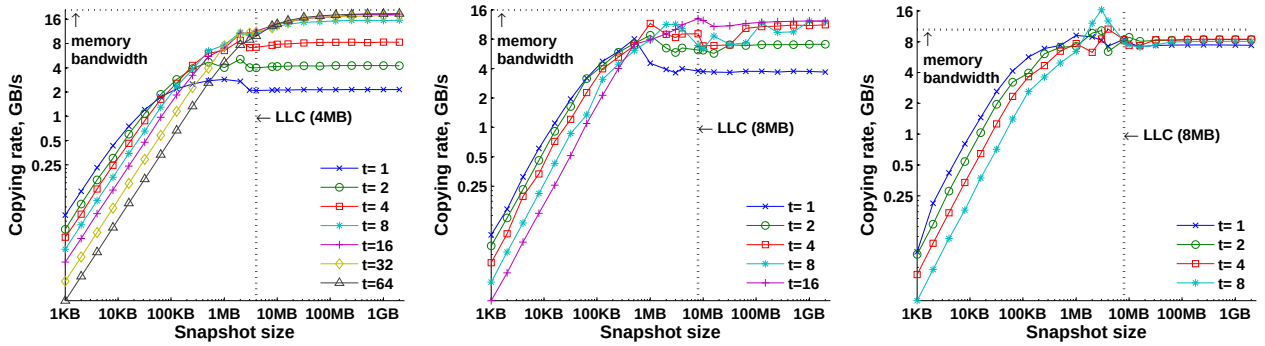
3.1 Snapshot Creation

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

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

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

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



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

Figure 1: Physical snapshotting performance.

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

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

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

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

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

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

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

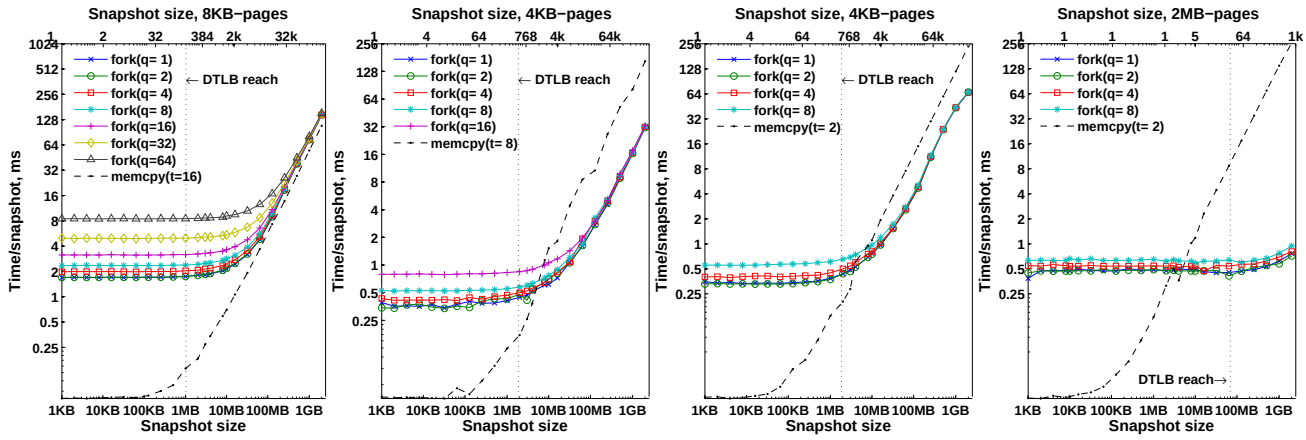
Since all q query threads must be recreated in a newly forked child process, we show how the cost increases with number of threads (different lines). To compare the performance of virtual and physical snapshotting, we additionally plot the physical copying time (the copying rate from Figure 1 converted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

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

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

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

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



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

Figure 2: Virtual snapshotting performance.

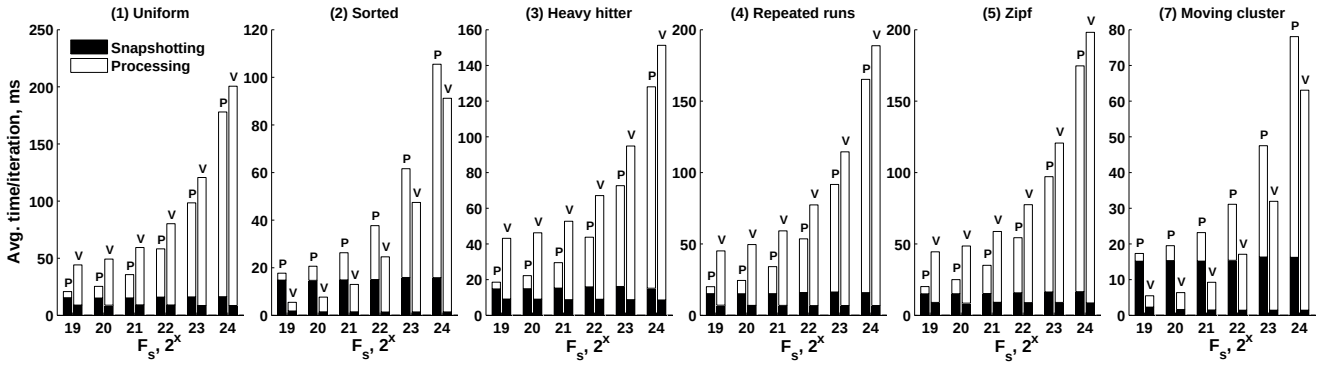


Figure 3: Snapshotting and update processing on Sandy Bridge (snapshot size = 2^{24} items = 128MB).

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

3.2 Virtual vs. Physical Snapshot Updating

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

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

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

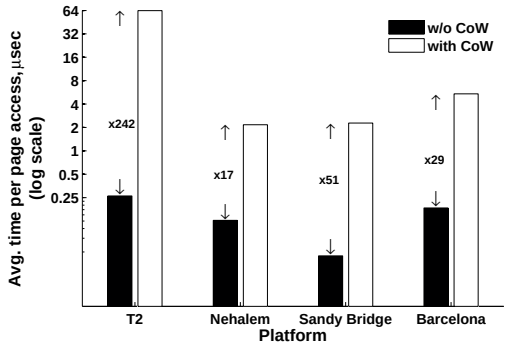


Figure 4: Copy-on-write cost.

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

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

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

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

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

3.3 Virtual vs. Physical Snapshot Querying

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

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

3.4 Memory Consumption

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

4. CONCLUSIONS

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

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

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

5. ACKNOWLEDGMENTS

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

6. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 2: Experimental platforms.

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

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

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

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

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

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

APPENDIX

A. EXPERIMENTAL SETTING

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

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

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

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

B. ADDITIONAL EXPERIMENTS

B.1 Update Performance

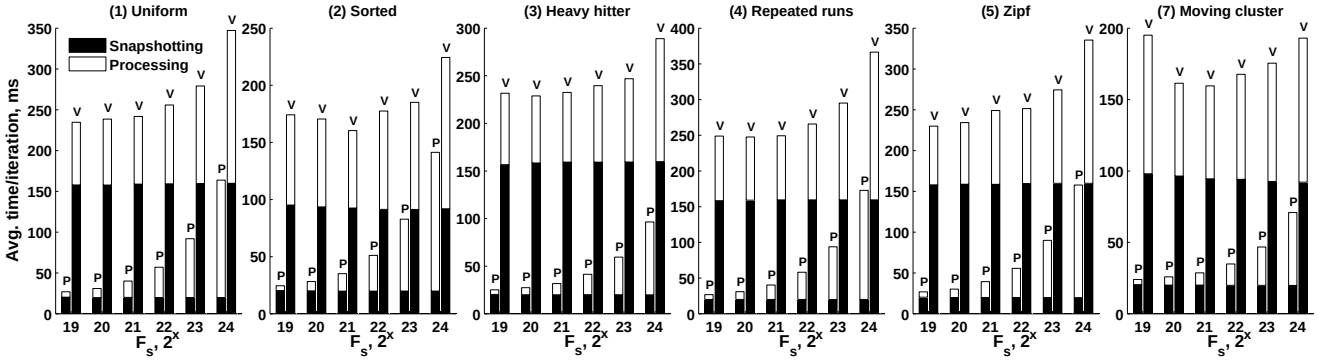
Figure 5 shows virtual and physical snapshot update performance on all our machines. The snapshot size is fixed to 128 MB (or 2^{24} items). On the x-axis, we plot the snapshotting frequency, F_s, expressed as the number of updates between snapshots. On the y-axis, we plot the average time per processing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

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

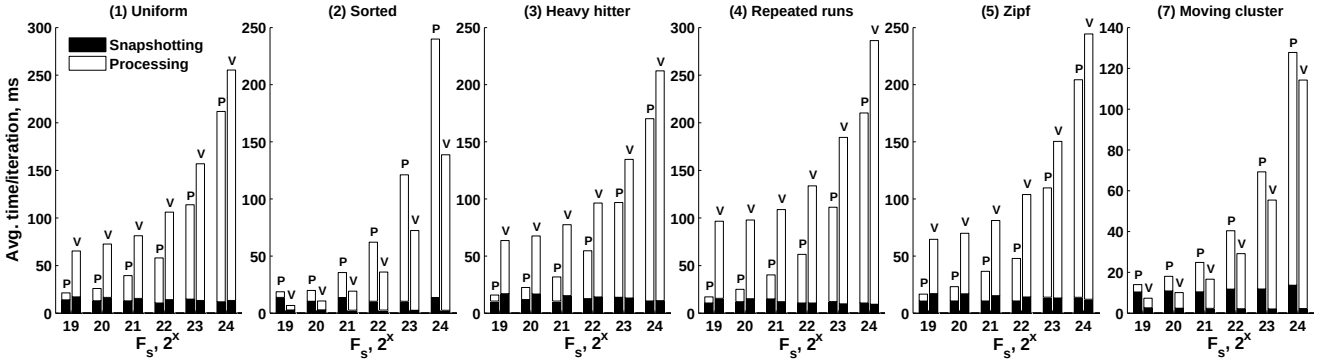
B.2 Poor `fork` Performance on T2

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

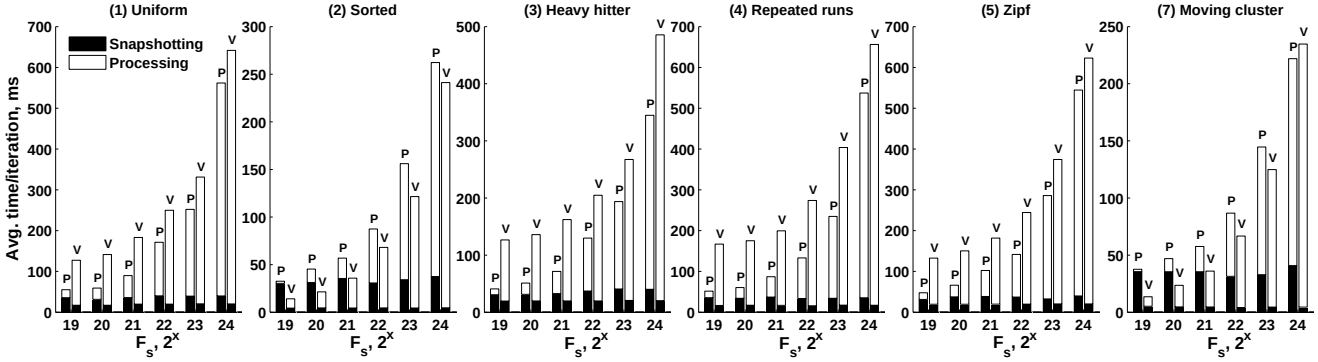
Second, the SPARC architecture has traditionally handled TLB misses in software using a software-managed, direct-mapped cache of translations, called a Translation Storage Buffer (TSB). To speed up look-ups in TSB, UltraSPARC T2 has hardware support for that. Nevertheless, other studies also show that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



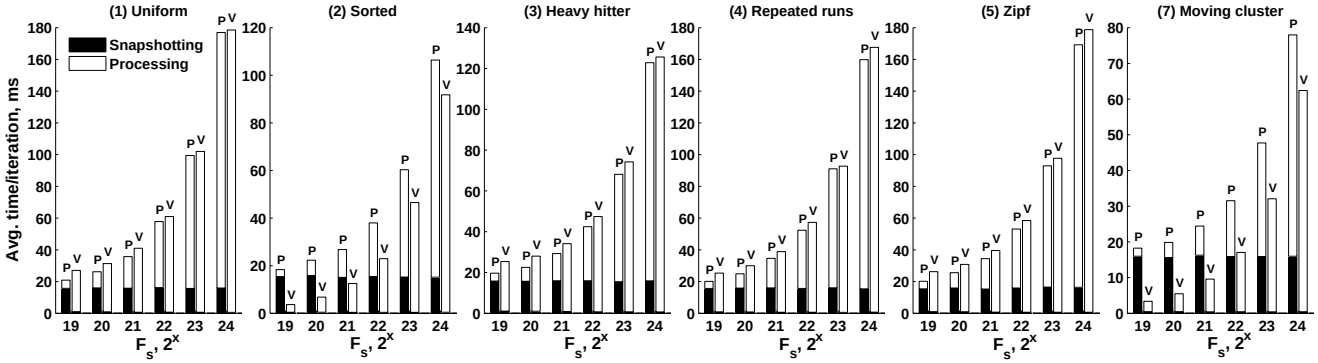
(a) Using 8KB-pages on Sun UltraSPARC-T2 CPU @ 1.2GHz



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



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



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

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