

# Turbocharging DBMS Buffer Pool Using SSDs

Jaeyoung Do  
University of Wisconsin-Madison  
jae@cs.wisc.edu

Donghui Zhang  
Microsoft Jim Gray Systems Lab  
dozhan@microsoft.com

Jignesh M. Patel  
University of Wisconsin-Madison  
jignesh@cs.wisc.edu

David J. DeWitt  
Microsoft Jim Gray Systems Lab  
dewitt@microsoft.com

Jeffrey F. Naughton  
University of Wisconsin-Madison  
naughton@cs.wisc.edu

Alan Halverson  
Microsoft Jim Gray Systems Lab  
alanhal@microsoft.com

## ABSTRACT

Flash solid-state drives (SSDs) are changing the I/O landscape, which has largely been dominated by traditional hard disk drives (HDDs) for the last 50 years. In this paper we propose and systematically explore designs for using an SSD to improve the performance of a DBMS buffer manager. We propose three alternatives that differ mainly in the way that they deal with the dirty pages evicted from the buffer pool. We implemented these alternatives, as well another recently proposed algorithm for this task (TAC), in SQL Server, and ran experiments using a variety of benchmarks (TPC-C, E and H) at multiple scale factors. Our empirical evaluation shows significant performance improvements of our methods over the default HDD configuration (up to 9.4X), and up to a 6.8X speedup over TAC.

**Classification:** H.2.4 [Database Management]: Systems

**Subjects:** Relational databases -- Microsoft SQL Server

**General Terms:** Algorithms, Design, Performance

**Keywords:** Buffer Pool, Flash SSD

## 1. INTRODUCTION

After three decades of incremental changes in the mass storage landscape, the memory hierarchy is experiencing a disruptive change driven by flash Solid State Drives (SSDs). SSDs have already made significant inroads in the laptop and desktop markets, and they are also making inroads in the server markets. In the server markets, their appeal is in improving I/O performance while also reducing energy consumption [8, 18]. The research community has taken note of this trend, and over the past year there has been substantial research on redesigning various DBMS components for SSDs [21, 23, 27, 33].

At first glance, Jim Gray's prediction of "flash is disk, disk is tape, and tape is dead" [12, 14] seems to have come true. However, SSDs are still in the early adoption phase, especially in large data centers [11]. One reason is that SSD storage today is at least an order of magnitude more expensive per byte than hard disk storage. Another related reason is that due to the excellent sequential read performance of modern hard disk drives, higher

sequential read performance can be achieved using striped disks more cost effectively than with SSDs [32]. These observations suggest that SSDs and disks are likely to coexist in large data centers for a long time [16].

SSDs can be used by a DBMS in a number of ways without modifying the DBMS, including storing some of the hot relations/indices [6], storing logs [24], storing temp storage [24], and so forth. These approaches are simple to deploy but have the drawback that decisions are made statically and at a coarse granularity.

Our focus in this paper is on improving DBMS performance by using SSDs as an extension to a traditional database buffer pool. The decision as to what data (evicted from the memory buffer pool) to store in the SSD is made at run time and dynamically changes as the working set migrates. In addition, decisions are made at the granularity of pages, rather than tables or files.

We present and explore three alternative designs: clean-write (*CW*), dual-write (*DW*), and lazy-cleaning (*LC*), respectively. These designs differ in the way they handle dirty pages that are evicted from the memory buffer pool. The CW design never writes dirty pages to the SSD. When a dirty page is evicted from the buffer pool, the DW design writes it "simultaneously" to both the database and the SSD (like a write-through cache). The LC design, on the other hand, writes dirty pages to the SSD first, and then lazily copies pages from the SSD to the database (like a write-back cache). In addition to these design alternatives, suitable replacement policies are employed to identify and retain the most beneficial pages cached in the SSD.

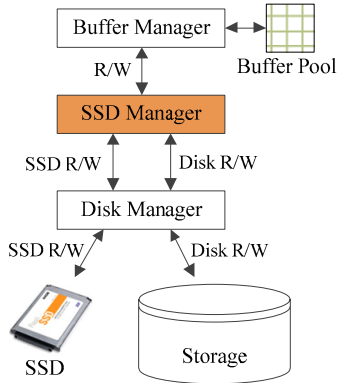
Recently, Canim *et al.* [7] proposed a way of using an SSD as a second-level write-through cache. They propose a new method called Temperature-Aware Caching (TAC) as their admission/replacement policy. TAC tracks the accesses to pages at a block level (in their implementation a block is a group of 32 contiguous pages) and records this as the temperature [5] of that block. TAC aims to cache the high temperature blocks in the SSD. Although our alternatives, like the TAC design, aim to use an SSD as a storage accelerator to bridge the performance gap between memory and disk, our paper extends their work in several ways. First, we describe three alternative approaches to use an SSD to extend the buffer pool. While we initially identified six alternatives, we selected three (CW/DW/LC) for implementation. In addition, we also implemented the TAC design [7]. In some cases, one of our designs (LC) outperformed TAC by 6.8X.

Second, our experiments used a hardware configuration that we feel better represents what a typical customer might deploy. The study in [7] used a 1 GB buffer pool, a 12 GB SSD, three disks, and a 48GB database to conduct the TPC-C experiment. We used

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...\$10.00.



**Figure 1: The relationship between the buffer, SSD, and disk managers along with the main-memory buffer pool, the SSD storage subsystem, and the disk storage subsystem.**

a system with a 20 GB buffer pool, a 140 GB SSD, eight disks, and a 400 GB database. We feel that it is important to investigate performance at this larger scale for several reasons. First, database systems frequently select different execution plans at different scales, so smaller-scale experiments may not be indicative of the results at a larger scale. Second, while running experiments at a larger scale can be extremely time consuming, our results revealed issues that would not have appeared at a smaller scale. These issues include a performance drop due to log-file growth, painfully long rampup/shutdown/checkpoint times, performance variations due to data placement in outer/inner tracks of the disks, and so forth.

Third, we experiment with a broader range of benchmark workloads. Canim *et al.* [7] considered only TPC-C (48 GB) and TPC-H (10 SF) scale factors as their experimental workloads. By contrast, we investigate: TPC-C (100 GB, 200 GB, 400 GB), TPC-E (~100 GB, ~200 GB, ~400 GB), and TPC-H (30 SF, 100 SF)<sup>1</sup>. Although TPC-E and TPC-C are both OLTP workloads, they differ from each other substantially in that TPC-E is read intensive while TPC-C is update intensive.

Fourth, re-implementing and evaluating TAC in a different DBMS using different hardware constitutes a contribution in its own right. Our community has recently identified the lack of experimental reproducibility as a weakness that detracts from the overall credibility of our field. By independently investigating TAC’s performance we hope to make a small contribution toward rectifying this weakness.

Our experimental results show that all the SSD designs provide substantial performance benefits for OLTP and DSS workloads, as illustrated by the sample of results shown in Figure 5 (refer to Section 4). For example, on a 2K warehouse (200 GB) TPC-C database, we observe about **9.4X** performance improvement over the no-SSD performance; for the 2K warehouse TPC-C database, the LC design provides **6.8X** better throughput than TAC.

<sup>1</sup> Disclaimer: While we have used the TPC-C, TPC-E, and TPC-H benchmark workloads as the basis of analysis presented in this paper, the results presented are not audited or official results and, in fact, were not run in a way that meets all of the benchmark requirements. The results are shown for the sole purpose of providing relative comparisons within the context of this paper and should not be compared to official TPC results.

The remainder of the paper is organized as follows: our design alternatives are presented in Section 2. Section 3 describes implementation details and lessons learned during the process of implementing and refining these designs. Experimental results are presented in Section 4. Related work is described in Section 5, and Section 6 contains our concluding remarks.

## 2. SSD DESIGN ALTERNATIVES

In this section we describe our three SSD designs: clean-write (CW), the dual-write (DW) and the lazy-cleaning (LC). We begin by reviewing key aspects of the buffer manager and disk manager in a traditional DBMS, and then explain how our designs fit in with these existing disk and buffer managers.

### 2.1 Storage Module Overview

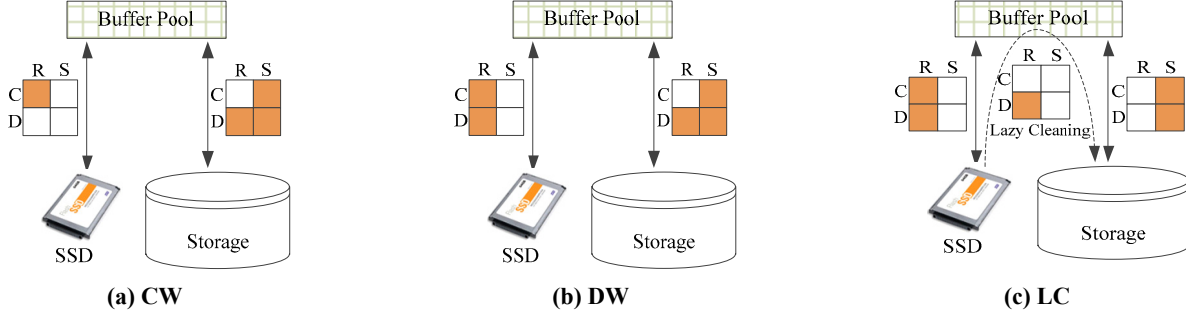
In a DBMS, the query execution engine makes calls to the buffer manager to request pages (for reading or writing), and the buffer manager, in turn, interacts with a disk manager that manages the interface to the physical storage devices. When a page is requested, the buffer manager first checks the buffer pool, and, if the requested page is found, the page is returned to the caller. Otherwise, if there is a free frame in the buffer pool, the buffer manager requests that the disk manager read the desired page (from disk) into the free buffer frame. If there is no free frame, the buffer manager picks a victim buffer page based on a replacement policy, and evicts that victim page. If the evicted page is dirty, then the evicted page is written to the disk before its frame is reused. The disk manager handles read and write requests passed to it from the buffer manager, by issuing asynchronous I/Os, to maximize both CPU and I/O resources.

### 2.2 SSD Manager

Each of our designs use a modified architecture that employs a new storage module component, called the *SSD Manager*. The SSD manager interacts with the buffer and disk managers, as illustrated in Figure 1. Like the (main-memory) buffer manager, the SSD manager uses an *SSD replacement policy* and an *SSD admission policy* to selectively cache pages in the SSD. The SSD replacement policy is used to determine which page to evict from the SSD when the SSD is full. The SSD admission policy plays a complimentary role and determines when a page should be considered for caching in the SSD. The admission of a page may then trigger an invocation of the replacement policy to evict a page if the SSD is full.

When the buffer manager gets a request for a page, it first examines the main-memory buffer pool for the page. If the page is found then a handle to the page is returned to the caller. If the page is not found, then the SSD manager is invoked to determine whether a copy of the page is cached on the SSD. If so, the page is read from the SSD into the main memory buffer pool and its usage information is updated (for use by the SSD replacement policy). Otherwise, the page is fetched from the disk. In the case that a clean page in the main-memory buffer pool is modified (dirty) and there is a copy of the page on the SSD, then the copy of the page in the SSD is invalidated by the SSD manager.

When the buffer manager decides to evict a page from the buffer pool it hands the page to the SSD manager, which, in turn, makes a decision about whether or not to cache it in the SSD based on an SSD admission policy (described below). If the page is selected for caching, then it is (asynchronously) written to the SSD. If, in turn, the SSD is full, the SSD manager chooses and evicts a victim



**Figure 2: Overview of the SSD designs: (a) the clean-write design, (b) the dual-write design, and (c) the lazy-cleaning design.**

based on the SSD replacement policy before writing the page to the SSD.

The *SSD admission policy* used in this study allows configurations in which the SSD can deliver much higher random performance than a disk (100X in our experimental setup), but provides smaller performance gains for sequential reads I/Os. As mentioned in the introduction, higher sequential read performance is easily and more economically achieved with a small number of striped disks. The SSD manager, therefore, differentiates pages that are read from disk using random accesses (such as through a non-clustered index lookup) versus sequential accesses (such as through table scans), and only pages fetched using random I/Os are considered for caching in the SSD.

In general, precisely classifying I/Os into the random and sequential categories is not easy. It becomes more difficult in a multi-user system where concurrent sequential I/Os can interleave [26], and with sophisticated I/O subsystems that optimize the order in which received I/Os are executed [9]. In this study, we take a simple approach that leverages the “read-ahead” mechanism that is often already found in a modern DBMS. Typical read-ahead mechanisms are already finely tuned to incorporate the complexities mentioned above, and will automatically be triggered during a sequential scan. In our designs, we simply exploit the existing read-ahead mechanism in SQL Server 2008 R2 [28]. Thus, when reading a page from disk, the SSD manager marks the page as “sequential” if it is read via the read-ahead method. Otherwise, the page read is marked as “random.” As an alternative, we could classify a page as sequential if the page is within 64 pages (512 KB) of the preceding page [29]. Our results indicate that leveraging the read-ahead mechanism is much more effective. For example, when we issued a sequential-read query, we observed that while the read-ahead mechanism was 82% accurate in identifying sequential reads, the method proposed in [29] was only 51% accurate. LRU-2 [30], which is widely used by commercial DBMSs, was used as the SSD replacement policy.

Note that we have chosen a specific admission policy above that reflects the current tradeoffs between random v/s sequential IO accesses between the SSD and the disks in our IO configuration. This admission policy can be generalized to adapt to other IO configuration, by simply caching in the SSD only those pages that are estimated to be faster to (re)access from the SSD compared to the disk.

## 2.3 The Three Design Alternatives

Now, we explain the three SSD design alternatives. These designs mainly differ in the way they deal with dirty pages that are evicted from the main-memory buffer pool. Specifically, if the dirty pages

are cached in the SSD using a write-back policy, the checkpoint and recovery logic must be modified. To graphically represent the target storage devices for pages evicted from the memory buffer pool, we use a quad chart in the C/D-R/S space, where C, D, R and S stand for “clean”, “dirty”, “random”, and “sequential”, respectively. Each page in the main-memory buffer pool is mapped to one quadrant of the chart. Below we use the terms “random” and “sequential” to refer to pages that are accessed from the database stored on disk using random and sequential I/Os respectively.

### 2.3.1 The Clean-Write (CW) Design

The CW design is illustrated in Figure 2(a). With this design, only clean pages are cached in the SSD. Whenever a dirty page is evicted from the main-memory buffer pool, it is only written to disk. Hence, the copy of each page in the SSD is identical to the copy on the disk. This design, consequently, requires no changes to the checkpoint and recovery logic.

Since dirty pages are never stored in the SSD with this design, CW mainly benefits applications whose working sets are composed of randomly accessed pages that are infrequently updated.

### 2.3.2 The Dual-Write (DW) Design

The DW design shown in Figure 2 (b) differs from the CW design in that dirty pages evicted from the buffer pool are written both to the SSD and to the database on disks; in effect, treating the SSD as a “write-through” cache for dirty pages. As with CW, clean pages that are selected for caching on the SSD are written only to the SSD. Since dirty pages evicted from the main-memory buffer pool are written to the SSD and the disk, the copy of each page in the SSD is identical to the copy on the disk (unless a crash occurs between the two operations), and therefore, no change to the checkpoint and recovery logic is required.

### 2.3.3 The Lazy-Cleaning (LC) Design

The LC design depicted in Figure 2(c) differs from the DW design in that dirty pages that are selected for caching on the SSD are written only to the SSD. A background “lazy-cleaning” thread is in charge of copying dirty SSD pages to disk when they are evicted from the SSD, thus implementing a “write-back” cache. As with CW and DW, clean pages that are selected for caching on the SSD are written only to the SSD. Since the SSD may have the latest copy of a page, there are checkpoint and recovery implications associated with this design.

Since SQL Server 2008 R2 uses a “sharp” check pointing mechanism [15], implementing the LC design required modifying the checkpoint logic to flush all the dirty pages in the SSD to disk

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Main-memory	$P$	$P'$			$P$	$P'$
SSD			$P$	$P'$	$P$	$P'$
Disk	$P$	$P$	$P$	$P$	$P$	$P$

**Figure 3: Six possible relationships amongst (up to) 3 copies of a page in memory, SSD, and disk.  $P$  denotes a copy of a page, and  $P'$  denotes a newer version of the page, so  $P \neq P'$ .**

when taking a checkpoint (in addition to all the dirty pages in the main memory buffer pool). With very large SSDs this can dramatically increase the time required to perform a checkpoint. An advantage of the sharp checkpoint (over the fuzzy checkpoint policy described below) is that the restart time (the time to recover from a crash) is fast.

To limit the time required for a checkpoint (in case of the sharp checkpoint policy) and for a system restart after a crash (in case of the fuzzy checkpoint policy), it is desirable to control the number of dirty pages in the SSD. The most “eager” LC method flushes dirty pages as soon as they are moved to the SSD, resulting in a design that is similar to the DW design. At the other extreme, in the “laziest” LC design, dirty pages are not flushed until the SSD is full. This latter approach has a performance advantage for workloads with repeated re-accesses to relatively “hot” pages that are dirtied frequently. However, delaying these writes to disk for too long can make the recovery time unacceptably long.

To mitigate this effect, the behavior of the background LC thread is controlled using a threshold parameter  $\lambda$  (expressed as a % of the SSD capacity). When the number of dirty pages in the SSD is above this threshold, then the LC cleaner thread wakes up and flushes pages to the disk so that the number of remaining dirty pages is slightly below this threshold (in our implementation about 0.01% of the SSD space below the threshold). In the evaluation section (Section 4) we investigate the effect of this tunable parameter.

## 2.4 Discussion

Each of the three designs has unique characteristics. The CW design never writes dirty pages to the SSD, and therefore, its performance is generally worse than the DW and the LC designs, unless the workload is read-only, or the workload is such that updated pages are never re-referenced. It is, however, simpler to implement than the other two designs because there is no need to synchronize dirty page writes to the SSD and the disk (required for DW), or to have a separate LC thread and to change the checkpoint module (required for LC).

The DW and LC designs differ in how they handle dirty pages that get evicted from the main-memory buffer pool. The LC design is better than DW if dirty SSD pages are re-referenced and re-dirtied, because such pages are written to and read back from the SSD multiple times before they are finally written back to the disk, while DW writes such pages to the disk every time they are evicted from the buffer pool. The DW design, on the other hand, is better than LC if the dirty SSD pages are not re-dirtied on subsequent accesses. Since pages cannot be directly transferred from the SSD to the disk<sup>2</sup>, pages being moved from the SSD to disk must first be read into main-memory and then written to the

<sup>2</sup> Current hardware platforms do not allow pages to be directly transferred between storage devices.

disk. Thus, DW saves the cost of reading the dirty pages from the SSD into the main-memory if they are not dirtied subsequently. In addition, DW is easier to implement than LC since it does not require an additional cleaner thread and does not require changing the checkpointing and recovery modules.

Note that the DW and the LC designs obey the write-ahead logging (WAL) protocol, forcibly flushing the log records for that page to log storage before writing the page to the SSD.

With the data flow scheme explained in Section 2.2, there may exist up to three copies of a page: one in memory, one in the disk, and one in the SSD. Figure 3 illustrates the possible relationships among these copies. First, if a page is cached in main-memory and disk, but not in the SSD, then the memory version could be the same as the disk version (case 1), or newer (case 2). Similarly, if a page is cached in the SSD and disk, but not in memory, then the SSD version is either the same as the disk version (case 3), or newer (case 4). When a page is cached in all three places, the main-memory version and the SSD version must be the same because whenever the memory version becomes dirty, the SSD version, if any, is immediately invalidated. As a result, the two cases where the memory/SSD version is the same as the disk version (case 5) or newer (case 6) are possible. As might be noticed, while all the cases apply to the LC design, only cases 1, 2, 3, and 5 are possible for the CW and DW designs.

## 2.5 Comparison with TAC

In this section we present a brief comparison of our designs (CW, DW, and LC) and TAC [7].

The page flow with TAC is as follows. (i) Upon a page miss (in the main memory buffer pool), the SSD buffer pool is examined for the page. If found, it is read from the SSD, otherwise it is read from disk. (ii) After a page is read from the disk into the main memory buffer pool, if the page qualifies for caching in the SSD, it is immediately written to the SSD. (iii) When a buffer-pool page is updated, the corresponding SSD page, if any, is *logically* invalidated. That is, the SSD page is marked as invalid but not evicted. (iv) When a dirty page is evicted from the buffer pool, it is written to the disk as in a traditional DBMS; at the same time, if the page has an invalid version in the SSD, it is also written to the SSD. The SSD admission and replacement policies both utilize the “temperature” of each extent (32 consecutive disk pages), which is used to indicate how “hot” the extent is. Initially the temperature of every extent is set to 0. With every page miss, the temperature of the extent that contains the page is incremented by the number of milliseconds that could be saved by reading the page from the SSD instead of the disk. Before the SSD is full, all pages are admitted to the SSD. After the SSD is full, a page is admitted to the SSD if its extent’s temperature is hotter than that of the coldest page in the SSD. That coldest SSD page is then selected for replacement.

Our designs differ from TAC in the following ways: First, when a page that is cached both in main-memory and on the SSD is modified, TAC does not immediately reclaim the SSD space that contains the copy of the page, while our designs do. Hence, with update-intensive workloads, TAC can waste a possibly significant fraction of the SSD. As an example, with the 1K, 2K and 4K warehouse TPC-C databases, TAC wastes about 7.4GB, 10.4GB, and 8.9GB out of 140GB SSD space to store invalid pages.

Second, for each of our designs clean pages are written to the SSD only after they have been evicted from the main-memory buffer

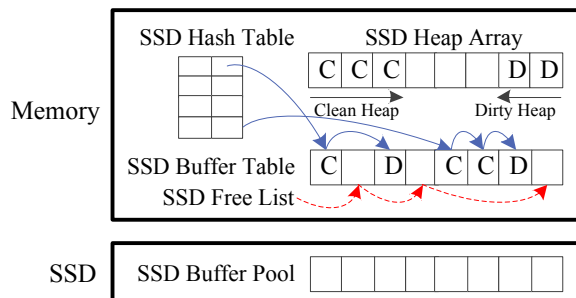


Figure 4: The data structures used by the SSD Manager.

pool. TAC, on the other hand, attempts to write pages to the SSD immediately after they are read from disk. We have observed that this approach (at least in our implementation of TAC) can create latch contention between the thread that is trying to write the page to the SSD and a second thread that wants to use the page to process transactions. Once the page has been read into main-memory, if the former thread grabs the latch on the page, forward processing (aka the second thread) can be delayed since pages cannot be used while they are being read/written. Consequently, the page latch wait times are higher with TAC. With the TPC-E workloads we have observed that TAC has page latch times that are about 25% longer on the average.

### 3. Implementation Details

In this section, we describe the implementation details of the SSD manager (the shaded component in Figure 1). We first introduce the data structures required for the three design alternatives, and then present some of the optimization techniques and lessons learned during the course of designing, implementing, and evaluating these designs.

#### 3.1 Data Structures

The SSD manager uses the following five data structures as illustrated in Figure 4.

- The *SSD buffer pool* is an array of  $S$  frames that are page-sized regions in which the database pages are cached. It resides on the SSD as a single file. This file is created on the SSD when the DBMS is started.
- The *SSD buffer table* is an array of  $S$  records corresponding to the frames in the SSD buffer pool. Each record has a page id, a dirty bit that describes the state of the matching page stored in the SSD, the last two access times, a latch, and some pointers such as the “next-free” pointer used in the *SSD free list* (to be described below). The size of each record is 88 bytes.
- To facilitate fast lookups, the records in the SSD buffer table that have the same hash index (transformed from their corresponding page IDs) are organized into a linked list (see the solid blue arrows in Figure 4). A hash table, called the *SSD hash table*, is used to point to the heads of such lists.
- The records in the SSD buffer table that correspond to free frames in the SSD buffer pool are organized into a linked list called the *SSD free list* (see the dashed red arrows.)
- The SSD manager must be able to find a victim page when the SSD is full or when cleaning dirty pages (in LC). To quickly identify a victim page, an *SSD heap array* is used. This heap

has nodes that point to entries in the SSD buffer table, and is used to quickly determine the oldest page(s) in the SSD, as defined by some SSD replacement policy (LRU-2 in our case).

This SSD heap array is divided into clean and dirty heaps. The clean heap stores the root (the oldest page that will be chosen for replacement) at the first element of the array, and grows to the right. The dirty heap stores the root (the oldest page that will be first “cleaned” by the LC thread) at the last element of the array, and grows to the left. This dirty heap is used only by LC for its cleaner thread and for checkpointing (CW and DW only use the clean heap).

#### 3.2 Checkpoint Operations

SQL Server 2008 R2 employs sharp checkpoints. Thus, in the case of the LC design, all the dirty pages in the SSD are flushed to the disk during a checkpoint. Note that during a checkpoint, LC stops caching new dirty pages evicted from the main-memory buffer pool in order to simplify the implementation.

In the DW design, during a checkpoint operation, dirty pages in the main-memory buffer pool that are marked as “random” are written both to the disk and to the SSD (as opposed to writing only to the disk). Although this technique extends the DW policy, in which only evicted pages, and not checkpointed pages, are flushed to the SSD, this modification has the advantage of potentially filling up the SSD faster with useful data.

#### 3.3 Optimizations

Our designs also incorporate a number of optimizations to the basic design, as described below.

##### 3.3.1 Aggressive Filling

To prime the SSD with useful data when SQL Server 2008 R2 starts from a cold start, all pages that are evicted from the main-memory buffer pool are cached in the SSD until the SSD is filled up to a certain threshold ( $\tau$ ). After this threshold is reached, only pages that are “qualified” for caching – as dictated by the admission policy – are stored in the SSD. The same technique with a different filling threshold is used in [7] and also incorporated in our TAC implementation.

##### 3.3.2 SSD Throttle Control

The SSD can become a bottleneck under high load. So we continuously monitor the SSD queue length to limit the load on the SSD. With this technique, no additional I/Os are issued to the SSD when the number of pending I/Os to the SSD exceeds a certain threshold ( $\mu$ ). However, when the SSD version of a requested page is newer than the disk version (only possible with the LC design), the page is read from the SSD for correctness. The same technique is incorporated in our TAC implementation.

##### 3.3.3 Multi-page I/O

When reading multiple data pages that are stored consecutively on disk, the I/O manager can optimize these requests and issue only one I/O call. In our designs, the individual pages in a read request are examined to see which pages reside in the SSD. In the initial implementations of our designs, we split the request into pieces. Consider, for example, a request to read six consecutive pages. If the 3<sup>rd</sup> and the 5<sup>th</sup> pages were in the SSD, five separate reads were performed (3 to the disk and 2 to the SSD). Unfortunately this approach actually reduced performance since the disk can handle a single large I/O request more efficiently than multiple small I/O requests.



**Table 1: Maximum sustainable IOPS for each device when using page-sized (8KB) I/Os. Disk write caching is turned off.**

READ			WRITE		
	Ran.	Seq.		Ran.	Seq.
8 HDDs	1,015	26,370	8 HDDs	895	946 <sup>3</sup>
SSD	12,182	15,980	SSD	12,374	14,965

Thus, to avoid breaking a single multi-page read into multiple smaller reads, only the leading and trailing pages from the requested list of pages (in the I/O request call) are trimmed if they are in the SSD, and the remaining pages are read consecutively from the disk in a single I/O operation. In the case that some of the pages in the middle of the read request have newer versions on the SSD, additional read I/Os are issued to read them these pages from the SSD (the older disk versions are removed from the main memory buffer pool immediately after they are read from disk).

### 3.3.4 SSD Partitioning

When one thread modifies an SSD data structure, other threads that try to access the same data structure must be blocked (via a latch wait). To increase concurrency, the SSD buffer pool is partitioned into  $N$  pieces, where  $N$  is the number of CPU hardware contexts. Different partitions share the SSD hash table, but each partition has its own segment of the SSD buffer table and SSD heap array.

### 3.3.5 Group Cleaning in LC

As mentioned in Section 2.2.3, the LC thread is invoked when the number of dirty pages in the SSD is higher than a threshold ( $\lambda$ ). In our implementation of the LC thread, it gathers up to  $\alpha$  dirty SSD pages with consecutive disk addresses, and writes them to the SSD using a single I/O operation.

## 4. EVALUATION

In this section, we compare and evaluate the performance of the three design alternatives using various TPC benchmarks.

### 4.1 Experimental Setup

#### 4.1.1 H/W and S/W Specifications

We implemented our three SSD design alternatives (*CW*, *DW*, and *LC*) and TAC (*TAC*) [7] in SQL Server 2008 R2, and compared the performance of these designs with each other and the standard version of SQL Server 2008 R2 (*noSSD*).

Our experiments were performed on a 2.27 GHz Intel dual CPU quad-core Xeon processor (Nehalem) running 64-bit Windows Server 2008 R2 with 24GB of DRAM. The databases, including TempDB, were created on file groups striped across eight 1 TB 7,200 RPM SATA hard disk drives (HDDs). We used two additional HDDs – one was dedicated to the OS and the other was used as the DBMS log disk. For the SSD buffer pool, we used a 160GB SLC Fusion ioDrive. In all experiments, 20GB of DRAM was dedicated to the DBMS, and 140GB of the SSD was used for the SSD buffer pool.

To aid the analysis of the results that are presented below, the I/O characteristics of the SSD and the collection of the 8 HDDs are given in Table 1. All IOPS numbers were obtained using Iometer

<sup>3</sup> The IOPS numbers were measured using Iometer while setting #outstanding I/Os = 1 for each disk. If #outstanding I/Os is set to 16 instead, the sequential-write IOPS changes to 12,246.

**Table 2: Parameter values used in our evaluations.**

Symbol	Description	Values
$\tau$	Aggressive filling threshold	95%
$\mu$	Throttle control threshold	100
$N$	Number of SSD partitions	16
$S$	Number of SSD frames	18,350,080 $\square$ (140GB)
$\alpha$	Maximum allowable number of dirty SSD pages gathered in an LC write request	32
$\lambda$	Dirty fraction of SSD space	1% (E, H), 50% (C)

[19]. In all the experiments, we used the optimization techniques described in Section 3.3 (for both our designs and TAC). Table 2 shows the parameter values used in our evaluations.

In all the experiments, the CW design performs worse than the DW and LC designs. For example, for the 20K customer TPC-E database, CW was 21.6% and 23.3% slower than DW and LC, respectively. Thus, in the interest of space, we omit additional CW results in the remainder of this section.

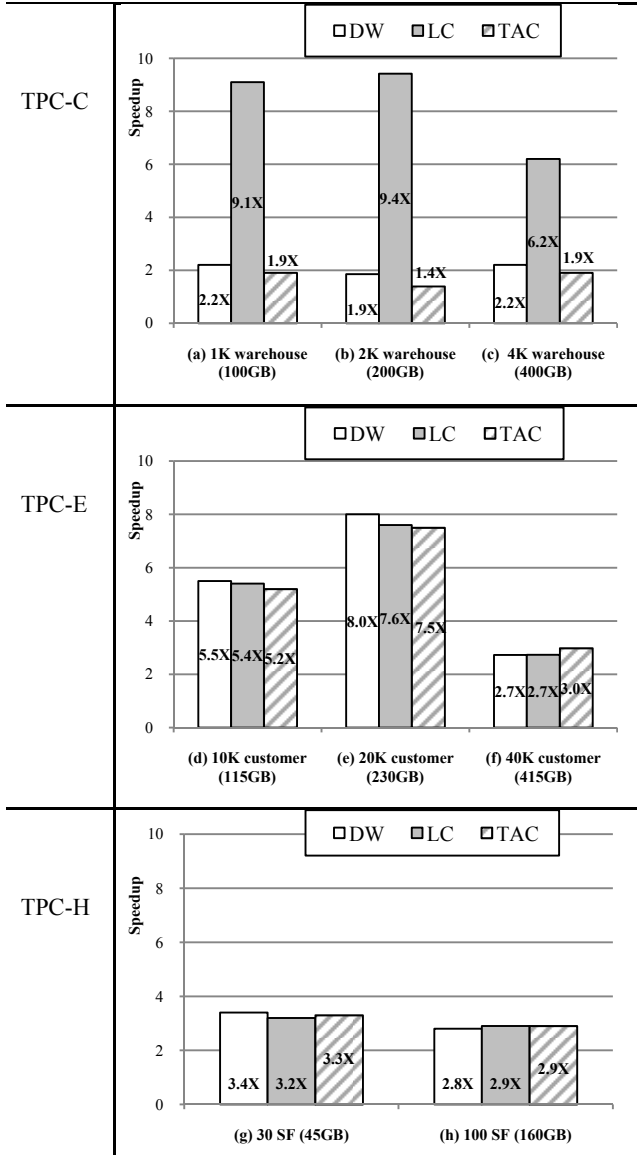
#### 4.1.2 Datasets

To investigate and evaluate the effectiveness of the SSD design alternatives, we used various OLTP and DSS TPC benchmark workloads, including TPC-C [35], TPC-E [36], and TPC-H [37].

One observation is that (as expected) the overall performance heavily depends on the relationship between the working set size of the workload, the DBMS buffer pool size, and the SSD buffer pool size. Thus, we designed the experiments to cover the following interesting cases: (a) when the database size is smaller than the SSD buffer pool size, (b) when the working set size is close to the SSD buffer pool size, and (c) when the working set size is larger than the SSD buffer pool size. For TPC-E we chose the 10k, 20k and 40k customer databases (corresponding to 115GB, 230GB and 415GB, database sizes respectively). For TPC-C, we selected the 1K, 2K and 4K warehouse databases with database sizes of 100GB, 200GB and 400GB, respectively. For TPC-H, we only used two scale factors: 30 and 100, corresponding to database sizes of 45GB and 160GB, respectively. Note these scale factors are “officially” allowed by the TPC-H specification. The next allowed scale factor (300SF) requires more than a week of run time for some data points with our hardware, and we omitted this run; as shown below, the trend with TPC-H is fairly clear with the two scale factors that we ran.

For each workload, we used different performance metrics according to the TPC specifications. For TPC-C and TPC-E, we measured the number of new orders that can be fully processed per minute (tpmC), and number of (Trade-Result) transactions executed within a second (tpsE), respectively. For TPC-H, we measured elapsed wall-clock time to complete the Power and the Throughput Tests, and transformed it to the TPC-H Composite Query-per-Hour Performance Metric (QphH). After each test run, we restarted the DBMS to clean up the main-memory and SSD buffer pools.

Since SQL Server 2008 R2 uses “sharp checkpoints” it flushes the dirty pages in the buffer pool to the disk during a checkpoint. To isolate the effects of checkpointing from the main goal of this empirical evaluation (namely, to quantify the benefits of the



**Figure 5: Speedups of DW, LC and TAC over the case without SSDs. We omit the CW results since it always performs worse than DW and LC in our experiments.**

various SSD caching designs), we set the interval between checkpoints to a large number for the update heavy TPC-C benchmark. Effectively, this setting turns off checkpointing.

In general we saw a significant reduction in performance – about 2X or more – when checkpointing is turned for TPC-C. This observation also highlights an important direction for future work: using the contents of the SSD during the recovery task, thereby requiring a sharp checkpoint to only flush the dirty pages in the main-memory to disk and not also those on the SSD. To improve performance even further one could conceptually use the SSD to hold the flushed pages. In principle this modification can be achieved by adding the SSD buffer table data structure, shown in Figure 4, along with a few other pieces of metadata information, to the checkpoint record. Exploring the implication of this with respect to fuzzy and sharp checkpoint performance is beyond the scope of this paper.

While checkpointing was effectively turned off for the TPC-C experiments, for the TPC-E and TPC-H experiments the recovery interval was set to 40 minutes. With this setting, the DBMS issues a checkpoint roughly every 40 minutes.

## 4.2 TPC-C Evaluation

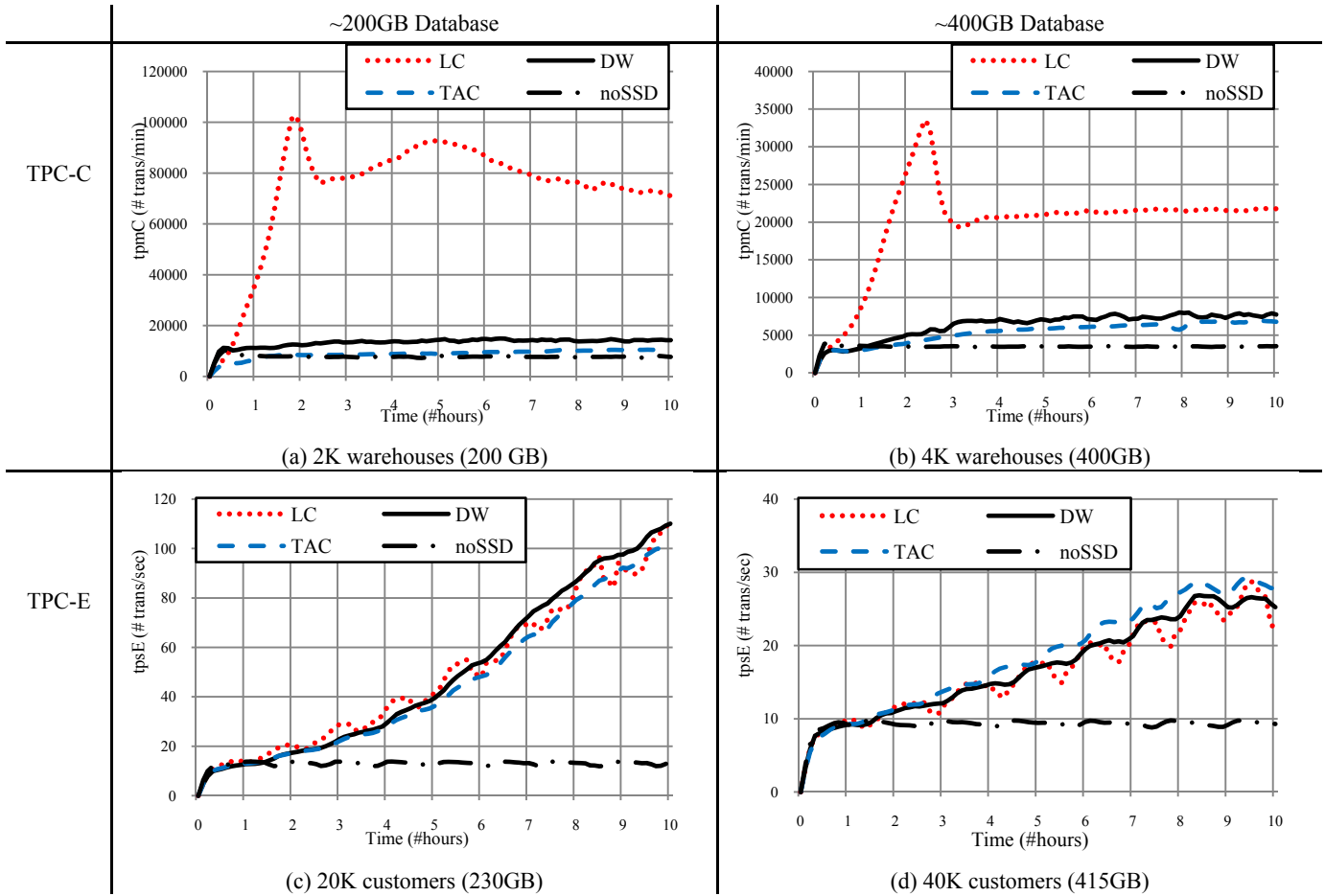
In this section, we evaluate the effectiveness of our SSD designs when running the TPC-C benchmark. Each design was run for 10 hours and we measured the tpmC as required by the benchmark. For LC, the parameter  $\lambda$  was set to 50%, meaning that the LC thread starts working once the number of dirty SSD pages exceeds 50% of the SSD buffer pool.

Figures 5 (a) – (c) present the steady-state throughput of the SSD designs (DW, LC and TAC) relative to the noSSD case for 1K, 2K and 4K warehouse databases, respectively. The speedups are calculated by using the average throughput achieved by each design over the last hour of execution (similar to the method that is specified by the TPC-C benchmark).

As can be seen in Figures 5 (a) – (c), LC clearly outperforms the other designs for each TPC-C configuration. LC is 9.4X faster for the 2K warehouse database than the noSSD case (while LC reads pages from the SSD at the speed of 74.2MB/s with an 89% SSD hit rate, the read speed of the 8 disks was only 6.7MB/s in the noSSD case). LC provided 5.1X and 6.8X better throughput relative to DW and TAC, respectively. For all the databases, DW also performed better than TAC, but worse than LC.

The write-back approach (LC) has an advantage over the write-through approaches (DW and TAC) if dirty pages in the SSD are frequently re-referenced and re-dirtied. The TPC-C workload is highly skewed (75% of the accesses are to about 20% of the pages) [25], and is update intensive (every two read accesses are accompanied by a write access). Consequently, dirty SSD pages are very likely to be re-referenced, and as a result, the cost of writing these pages to disk and reading them back is saved with the LC design. For example, with a 2K warehouse database, about 83% of the total SSD references are to dirty SSD pages. Hence a design like LC that employs write-back policy works far better than a write-through policy (such as DW and TAC).

While DW and TAC are both write-through approaches, DW performs better than TAC for TPC-C. We suspect that the main reason lies in the difference in their page flows. In TAC, a page is written to the SSD under two circumstances: (a) immediately after being loaded from the disk; and (b) upon eviction from the buffer pool, if the page is dirty, and if an invalid version of the page exists in the SSD. In a DBMS like SQL Server 2008 R2 that uses asynchronous I/Os, there may exist a gap between the time a page has been loaded from the disk and the time the page starts to be written to the SSD. If a forward-processing transaction grabs the latch to the page and dirties it first, TAC cannot write the page to the SSD because that would make the SSD version of the page newer than the disk version (which conflicts with its write-through policy). As a result, when this dirty page is later evicted from the main-memory buffer pool, TAC does not write it to the SSD because an invalid version of the page does not exist in the SSD. A similar problem occurs when a dirty page is generated on-the-fly, e.g. during a B+-tree page split. TAC does not write such pages to the SSD because they were not read from the disk in the first place. The DW design does not have this problem as it writes dirty pages to both the SSD and the disk when they are evicted from the buffer pool. This difference has a bigger impact for TPC-C than for TPC-E or TPC-H, because TPC-C is update intensive.



**Figure 6: 10-hour test run graphs with the TPC-C databases: (a) 2K warehouses and (b) 4K warehouses, and TPC-E databases: (c) 20K customers and (d) 40K customers.**

Another reason why DW may have better performance than TAC is that TAC wastes some SSD space (see Section 2.5). When a page in the main-memory buffer pool is modified, DW uses a *physical invalidation* mechanism (the SSD frame is appended to the free list). TAC, on the other hand, uses a *logical invalidation* mechanism (the SSD frame is marked as invalid, but is not released). Consequently, after the SSD is full, TAC may choose a valid SSD page for replacement, even though the SSD contains some pages that have been invalidated.

#### 4.2.1 Ramp-Up Period

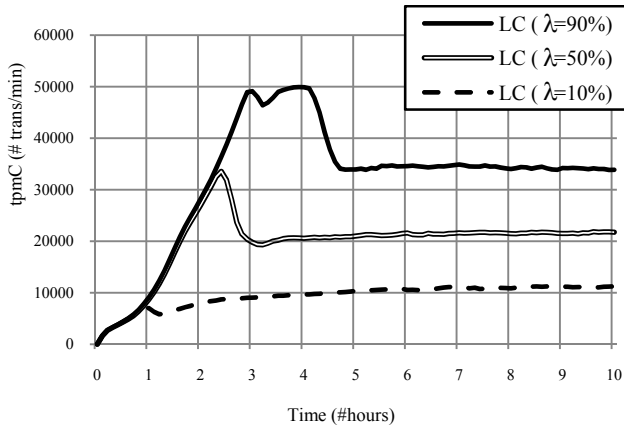
To analyze the behavior of each SSD design during the full 10-hour execution (recall Figures 5 (a) – (c) are the average throughput over the last hour), we present the six-minute average throughput versus elapsed wall clock time in Figures 6 (a) and (b). To conserve space, we only present the results for the 2K and 4K warehouse databases. To make the graphs more readable, each curve is smoothed using the moving average of three adjacent data points.

As can be observed in Figures 6 (a) and (b), the performance of the LC design drops significantly after 1:50 hours and 2:30 hours with the 2K (Figure 6 (a)) and 4K (Figure 6 (b)) warehouse databases, respectively. Before these drops, the number of dirty SSD pages was lower than the 50% threshold ( $\lambda$ ), so the SSD could completely absorb the dirty pages being evicted from the

main memory to the disk. Once the number of dirty pages in the SSD is greater than  $\lambda$ , the LC thread kicks in and starts to evict pages from the SSD to the disk (via a main-memory transfer as there is no direct I/O path from the SSD to disk). As a result, a significant fraction of the SSD and disk bandwidth is consumed by the LC thread, which in turn reduces the I/O bandwidth that is available for normal transaction processing.

To evaluate the impact of the parameter  $\lambda$  on the LC design, we tested LC with three different dirty fractions ( $\lambda = 10\%$ ,  $50\%$ ,  $90\%$ ). As illustrated in Figure 7, the speedup of LC increases as more dirty pages are allowed in the SSD. For example, with  $\lambda = 90\%$ , LC improves performance by 3.1X and 1.6X (steady-state throughput) over the 10% and 50% cases, respectively. Larger values of  $\lambda$  allow LC to cache more dirty pages, and thereby issue fewer I/Os to evict dirty SSD pages. Since the TPC-C workload has high re-reference and update rates, by keeping an evicted (from the main-memory buffer pool) dirty page in the SSD for a longer period of time, the LC design is likely to absorb more read/write requests associated with the dirty pages. For example, at  $\lambda = 90\%$  the LC thread issued about 521 IOPS to the disks, while at  $\lambda = 50\%$  and at  $\lambda = 10\%$ , 769 and 950 IOPSs were issued, respectively. Thus, with higher  $\lambda$  settings, LC can use more of the capacity of the SSD and disk subsystem to process transactions, resulting in higher overall transaction throughput.





**Figure 7: The effect of allowing more dirty pages in the SSD buffer pool with the TPC-C 4k warehouse database.**

### 4.3 TPC-E Evaluation

This section presents the results of running the TPC-E benchmarks for 10 hours. For these experiments checkpointing was turned on (triggered roughly every 40 minutes), and the LC thread was set to start working when  $\lambda$  was greater than 1% (varying  $\lambda$  did not have a significant impact for this benchmark). The results for TPC-E are summarized in Figures 5 (d) – (f).

As the results indicate each the four designs provide significant (and similar) performance gains over the noSSD case, about 5.5X, 8.0X, and 3X for the 10K, 20K and 40K customer databases, respectively.

Compared to the TPC-C results, the advantage of LC over DW and TAC is gone. This is because, while the TPC-C workload is update-intensive and highly skewed, updates constitute a lower fraction of the TPC-E workload. The performance advantage of DW over TAC is relatively lower due to the fact that there are fewer updates and hence the probability that a page is dirtied before being written to the SSD is smaller.

Also notice in Figures 5 (d) – (f) that the performance gains are the highest with the 20K customer database. In this case, the working set nearly fits in the SSD. Consequently, most of the random I/Os are offloaded to the SSD, and the performance is gated by the rate at which the SSD can perform random I/Os. As the database size decreases/increases, the performance gain obtained for each of the four designs is lower. As the working set becomes smaller, more and more of it fits in the memory buffer pool, so the absolute performance of the noSSD increases. On the other hand, as the working set gets larger, a smaller fraction of it fits in the SSD. Thus, although there is a lot of I/O traffic to the SSD, it can only capture a smaller fraction of the total traffic. Consequently, the SSD has a smaller impact on the overall performance, and the performance is gated by the collective random I/O speed of the disks.

#### 4.3.1 Ramp-Up Period

As with TPC-C (see Section 4.2.1) we now show the full 10-hour run for TPC-E. Again in the interest of space we only show the results with the 20K and the 40K customer databases. These results are shown in Figures 6 (c) and (d) respectively.

As can be seen in Figures 6 (c) and (d), the “ramp-up” time taken for the SSD designs is fairly long (and much longer than with

TPC-C). In the case of the DW design, steady state was achieved only after about 10 and 8.5 hours with the 20K and 40K customer databases, respectively. The reason for this long ramp-up time stems from the poor random-read bandwidth of the disk subsystem we used for these experiments. During most of the ramp-up period, the SSD buffer pool is continually getting populated with pages that are evicted from the memory buffer pool because of the aggressive filling technique (Section 3.3.1) that is used. This means that the speed with which the SSD can be populated directly depends on the rate at which pages are evicted from the main-memory buffer pool, which in turn is constrained by the speed of reading pages (not in the SSD) from the disks using random accesses (as most of the misses tend to result in a random disk I/O). Note that the ramp-up time with the 40K customer database is shorter than the 20K customer database. With the 20K database the working set nearly fits in the SSD, which results in repeated re-references with updates to the SSD pages during the ramp-up period. When this happens, first the page in the SSD must be invalidated. Later, when that page is evicted from the main-memory buffer pool it is written back to the SSD. Thus, during the ramp-up period a sizable chunk of the disk I/O bandwidth is consumed by these re-references. Consequently, the SSD is filled at a much slower rate.

#### 4.3.2 I/O Traffic

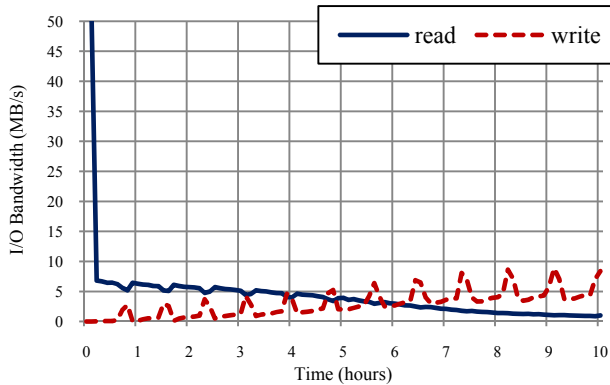
Next we dig deeper into the dynamic I/O characteristics during the entire run of the TPC-E benchmark to better understand the traffic that both the disks and the SSD see during the benchmark run. In the interest of space, we only focus on the 20K customer database with DW as the insights from this case are fairly representative of what happens with the other datasets and the other SSD designs.

Figures 8 (a) and (b) show the read and write traffic to the disks and SSD, respectively. First, we notice that, in Figure 8 (a), initially the disks started reading data at 50MB/s but drastically dropped to 6MB/s. This is a consequence of a SQL Server 2008 R2 feature that expands every single-page read request to an 8 page request until the buffer pool is filled. Once filled, the disk write traffic starts to increase as pages are evicted to accommodate new pages that are being read. Notice also that in Figure 8 (b), the SSD read traffic steadily goes up until steady state is achieved (at 10 hours) – this is the time it takes to fill the SSD. In both figures, notice the spikes in the write traffic, which is the I/O traffic generated by the checkpoints.

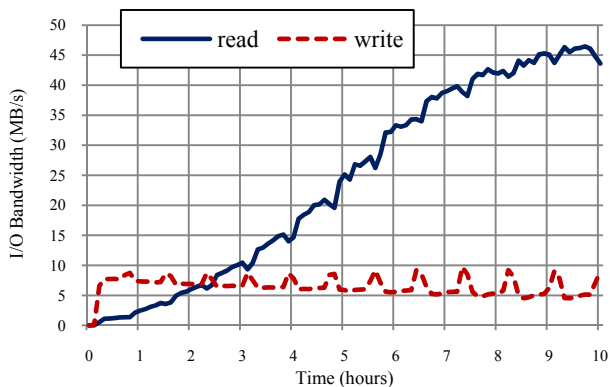
In steady state (after 10 hours), the bottleneck is the aggregated (random) traffic to the disks, which is 6.5MB/s, or 832 IOPS, close to the maximum that the disks can collectively support (See Table 1). Also notice that the SSD bandwidth is not saturated in the steady state. At most, the SSD sees 46MB/s of read traffic and 9.7MB/s of write traffic, but it can support 12,182 IOPS (= 95MB/s) with 1 page reads (similar for 1 page writes, see Table 1). This observation indicates that a very high performance SSD like the Fusion I/O card may not be required to obtain the maximum possible performance if the disk subsystem is the bottleneck.

#### 4.3.3 Effects of Checkpointing

In Figures 6 (c) and (d), we observed that the performance of each design (including noSSD) drops periodically when a checkpoint is taken (roughly every 40 minutes for the setting used). Currently when SQL Server 2008 R2 takes a checkpoint, the DBMS flushes all the dirty pages in the main-memory buffer pool to disk (i.e., a sharp checkpoint), which results in a degradation in throughput.



(a) disks



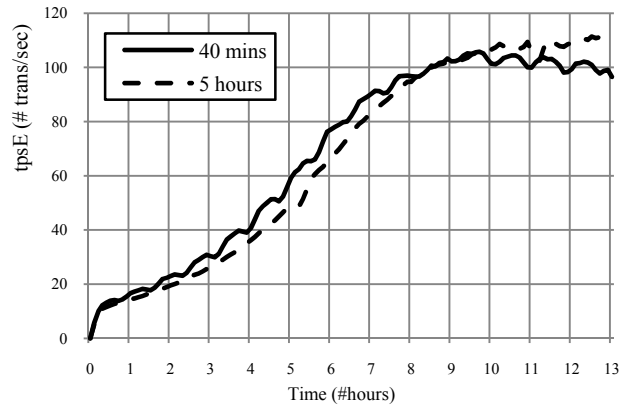
(b) SSD

**Figure 8: I/O traffic to the disks and SSD on the TPC-E 20K customer database with the DW design.**

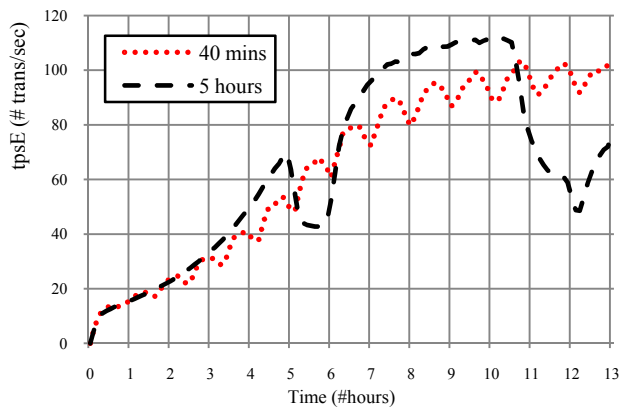
Amongst the SSD designs, checkpointing has the largest impact on LC (the dips are sharper and longer) because LC must also write all the dirty pages in the SSD buffer pool to disk as part of the checkpoint (in order to comply with SQL Server 2008 R2’s sharp checkpointing policy).

Next, we wanted to observe what would happen if we used a very large checkpoint interval (5 hours) in an attempt to maximize performance. Indirectly, this experiment is also interesting as it provides insight into the performance of the system if one were to change the actual checkpointing code to leverage the contents of the SSD during recovery. In this experiment, we only consider the DW and LC designs, since the previous results show that DW and TAC have similar checkpointing behaviors. In addition, the “dirty fraction” for LC ( $\lambda$ ) is increased from 1% to 50% to allow the SSD to accumulate more dirty pages, thereby not incurring additional disk I/Os in order to stay below the 1% threshold.

The results for this experiment are shown in Figures 9 (a) and (b) along with the results obtained using a 40 minute checkpoint interval. Comparing the DW curves (Figure 9 (a)), we notice that for the first 10 hours a shorter recovery interval improves the overall performance as the SSD does not get filled to its capacity during this period. Once the SSD is filled, the five hour



(a) DW



(b) LC

**Figure 9: Effect of increasing the checkpoint interval on the TPC-E 20K customer database for (a) DW and (b) LC.**

checkpoint interval provides better performance. The reason for this behavior is that, the dirty pages that are flushed from the main memory buffer pool to the SSD and disk during the checkpoint bump pages out of the SSD that are accessed subsequently.

In the case of LC (Figure 9 (b)), using the larger recovery interval initially (for the first 5 hours) improves performance until the first checkpoint is performed at which point its performance drops dramatically (until the checkpoint is finished at about 6.5 hours). With a 5 hour interval many dirty pages have accumulated in the SSD and the checkpoint takes much longer to complete than with a 40 minute interval.

#### 4.4 TPC-H Evaluation

Finally, we evaluated the alternative SSD designs using the TPC-H benchmark. As required by the benchmark specification [37], we first ran a power test that measures the raw query execution power by running the 22 TPC-H queries serially. Then, we ran a throughput test by executing four (@30SF) or five (@100SF) query streams concurrently. Before/after each query stream, we inserted/deleted records from the database as specified in the specification. The checkpoint and LC thread configurations are the same as for the TPC-E evaluation (see Section 4.3).

**Table 3: TPC-H Power and Throughput test results.**

30SF TPC-H	LC	DW	TAC	noSSD
Power Test	5978	5917	6386	2733
Throughput Test	5601	6643	5639	1229
QphH@30SF	5787	6269	6001	1832
100SF TPC-H	LC	DW	TAC	noSSD
Power Test	3836	3204	3705	1536
Throughput Test	3228	3691	3235	953
QphH@100SF	3519	3439	3462	1210

Figures 5 (g) and (h) show the performance speedups of the SSD designs over the noSSD case for the 30 and 100 SF databases, respectively. The three designs provide significant improvements in performance: up to 3.4X and 2.9X on the 30 and 100 SF databases, respectively. The reason why the SSD designs improve performance even for the TPC-H workload (which is dominated by sequential I/Os), is that some queries in the workload are dominated by index lookups in the LINEITEM table which are mostly random I/O accesses. The three SSD designs have similar performance, however, because the TPC-H workload is read intensive.

Table 3 shows the detailed results for the TPC-H evaluation. The SSD designs are more effective in improving the performance of the throughput test than the power test as the multiple streams of the throughput tests tend to create more random I/O accesses. With the 30SF database, for example, DW is 2.2X faster than noSSD for the power test, but 5.4X faster for the throughput test.

## 5. RELATED WORK

Using Flash SSDs to extend the buffer pool of a DBMS has been studied in [3, 7, 17, 22]. Recently, Canim *et al.* [7] proposed the TAC method and implemented it in IBM DB2. Simulation-based evaluations were conducted using TPC-H and TPC-C, and a TAC implementation in IBM DB2 was evaluated using a 5GB ERP database and a 0.5k-warehouse (48GB) TPC-C database.

Holloway [17] proposed another SSD buffer-pool extension scheme, that we call the “rotating-SSD design”, and implemented it in MySQL. The design was motivated by the relatively slow random write performance of non-enterprise class SSDs. The SSD buffer pool is organized as a circular queue and a logical pointer called *nextFrameNo* is used to indicate which SSD frame should be allocated next. This pointer rotates through all the frames in the SSD buffer pool sequentially. When a page (dirty or clean) is evicted from the buffer pool, it is written to the SSD frame that is pointed to by the *nextFrameNo* pointer. If the frame already contains a valid page, then that page is evicted from the SSD. In the case that the page to be evicted does not exist in memory and is newer than the disk version, it is written to the disk before being evicted. Note that the eviction of the page stored at *nextFrameNo* takes place even if the page in that frame is frequently accessed. The design sacrifices the quality of the SSD replacement policy in order to obtain truly sequential write behavior to the SSD. Since the random write performance of enterprise-class SSDs continues to improve, the premise on which this design is based is losing its validity.

Koltsidas and Viglas [22] proposed three SSD buffer-pool extension designs: the *inclusive* approach (which is somewhat

similar to TAC’s page flow in that a page in the memory buffer pool is allocated a frame in the SSD), the *exclusive* approach (where a page never exists in both the buffer pool and the SSD), and the *lazy* approach (somewhat similar to our LC design). In the *exclusive* approach, whenever a page is read from the SSD to memory, the SSD version of the page is removed. Later when the page is evicted from the buffer pool, it has to be written to the SSD again.

Also related is the HeteroDrive system [20], which uses the SSD as a write-back cache to convert random writes to the disk into sequential ones.

Instead of using the SSD as a cache, an alternative approach is to use the SSD to permanently store some of the data in the database. Koltsidas and Viglas [21] presented a family of on-line algorithms to decide the optimal placement of a page and studied their theoretical properties. Canim *et al.* [6] describes an *object placement advisor* that collects I/O statistics for a workload and uses a greedy algorithm and dynamic programming to calculate the optimal set of tables or indexes to migrate.

The effect of using the SSD for storing the transaction log or for temp space was studied in [24].

Since Jim Gray’s 2006 prediction [12] that “tape is dead, disk is tape, and flash is disk”, substantial research has been conducted to improve DBMS performance by using flash SSDs including revisiting the five-minute rule based [13], benchmarking the performance of various SSD devices [4], examining methods for improving various DBMS internals such as query processing [10, 32], index structures [1, 27, 38], and page layout [23].

Recently the industry has released several storage solutions integrated with SSD to accelerate the performance of traditional disk-based storage. Oracle Exadata [31] uses the SSD as a smart cache for frequently accessed disk pages. Teradata Virtual Storage System [34] continuously migrates hot data to the SSD. Solaris ZFS file system [2] uses the SSD as a second-level cache that extends the DRAM cache.

## 6. CONCLUSIONS AND FUTURE WORK

This paper evaluates four alternative designs that use an SSD to extend SQL Server 2008 R2’s buffer pool across a broad range of benchmarks. One of these designs includes a recently proposed design called TAC [7]. Our results indicate that these designs offer the potential of up to a 9.4X speedup for OLTP workloads and up to a 3.4X speedup for decision support workloads. A number of factors impact the overall speedup that can be obtained, including the difference in the random I/O performance between the SSD and the disk subsystem and the ratio of the working set size of the workload to the SSD buffer pool size. If the random I/O performance gap between SSDs and moving-head disks continues to increase, the improvement in performance will continue to increase as well. The best speedup can be achieved when the working set size is close to the SSD buffer pool size.

For our experiments and settings LC (a write-back cache design) offered the best overall performance for update-intensive workloads, as the LC design benefits from allowing dirty pages to be cached in the SSD. For read-intensive workloads, DW, TAC and LC have similar performance, as they all cache the pages that are randomly accessed in SSD.

Finally, using the SSD as a buffer pool extension has a number of implications on other aspects of a database management system.

No design to-date leverages the data in the SSD during system restart, and as a result, it takes a very long time to warm-up the SSD with useful pages (i.e. the “ramp-up” time before reaching peak throughput is very long). Another observation from our study is that unless a large fraction of the random I/O accesses are offloaded to the SSD, it is likely that the disk subsystem will remain a bottleneck. In such a scenario, using mid-range SSDs may provide similar performance benefits to those provided by expensive high-end enterprise SSDs. Exploring these issues is an interesting direction for future work.

## 7. REFERENCES

- [1] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. PVLDB, 2009.
- [2] R. Bitar. Deploying Hybrid Storage Pools with Sun Flash Technology and the Solaris ZFS File System. In Sun BluePrints Online, 2008.
- [3] B. Bhattacharjee, M. Canim, C. A. Lang, G. Mihaila, and K. A. Ross. Storage Class Memory Aware Data Management. In IEEE Data Engineering Bulletin, 33(4), 2010.
- [4] L. Bouganim, B. T. Jonsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In CIDR, 2009.
- [5] G. P. Copeland, W. Alexander, E. E. Boughter and T. W. Keller. Data Placement in Bubba. In SIGMOD, 1988.
- [6] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. PVLDB, 2009.
- [7] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. In VLDB, 2010.
- [8] Computer World. HP offers SSDs in ProLiant server line, 2008.  
[http://www.computerworld.com/s/article/9138209/HP\\_offers\\_SSDs\\_in\\_ProLiant\\_server\\_line](http://www.computerworld.com/s/article/9138209/HP_offers_SSDs_in_ProLiant_server_line)
- [9] B. Dees. Native Command Queuing – Advanced Performance in Desktop Storage. IEEE Potentials, 2007.
- [10] J. Do and J. M. Patel. Join Processing for Flash SSDs: Remembering Past Lessons. In DaMoN, 2009.
- [11] EE Times. SSDs: Still not a ‘Solid State’ Business, 2010.  
<http://www.eetimes.com/electronics-news/4206361/SSDs--Still-not-a--solid-state--business>
- [12] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King, 2006.  
[http://research.microsoft.com/en-us/um/people/gray/talks/Flash\\_is\\_Good.ppt](http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt)
- [13] G. Graefe. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In DaMoN, 2007.
- [14] J. Gray and B. Fitzgerald. Flash Disk Opportunity for Server Applications. ACM Queue, 2008.
- [15] J. Gray and A. Reuter. Chapter 12: Advanced Transaction Manager Topics. In Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.
- [16] S. Hetzler. The Storage Chasm: Implications for the Future of HDD and Solid State Storage. IBM, 2009.
- [17] A. L. Holloway. Chapter 4: Extending the Buffer Pool with a Solid State Disk. In Adapting Database Storage for New Hardware, University of Wisconsin-Madison Ph.D. thesis, 2009.
- [18] Information Week. Google Plans to Use Intel SSD Storage in Servers, 2008.  
<http://www.informationweek.com/news/storage/systems/showArticle.jhtml?articleID=207602745>
- [19] Iometer. <http://www.iometer.org>
- [20] S.-H. Kim, D. Jung, J.-S. Kim, and S. Maeng. HeteroDrive: Re-shaping the Storage Access Pattern of OLTP workload using SSD. In IWSSPS, 2009.
- [21] I. Koltsidas and S. Viglas. Flashing Up the Storage Layer. PVLDB, 2008.
- [22] I. Koltsidas and S. Viglas. The Case for Flash-Aware Multi-Level Caching. Technical Report, University of Edinburgh, 2009.  
<http://homepages.inf.ed.ac.uk/s0679010/mfcache-TR.pdf>
- [23] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In SIGMOD, 2007.
- [24] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In SIGMOD, 2008.
- [25] S. T. Leutenegger and D. M. Dias. A Modeling Study of the TPC-C Benchmark. In SIGMOD, 1993.
- [26] C. Li, K. Shen, and A. E. Papatthanasious. Competitive Prefetching for Concurrent Sequential I/O. In EuroSys, 2007.
- [27] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree Indexing on Solid State Drives. PVLDB, 2010.
- [28] Microsoft SQL Server 2008 R2 Editions.  
<http://www.microsoft.com/sqlserver/2008/en/us/R2-editions.aspx>
- [29] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. I. T. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In EuroSys, 2009.
- [30] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In SIGMOD, 1993.
- [31] Oracle Exadata.  
<http://www.oracle.com/us/products/database/exadata>.
- [32] M. Polte, J. Simsa, and G. Gibson. Enabling Enterprise Solid State Disks Performance. In WISH, 2009.
- [33] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In SIGMOD, 2009.
- [34] Teradata. Virtual Storage.  
<http://www.teradata.com/t/brochures/Teradata-Virtual-Storage-eb5944>.
- [35] TPC Benchmark C (TPC-C). <http://www.tpc.org/tpcc>.
- [36] TPC Benchmark E (TPC-E). <http://www.tpc.org/tpce>.
- [37] TPC Benchmark H (TPC-H). <http://www.tpc.org/tpch>.
- [38] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems. ACM TECS, 2007.