Enhancing Recovery Using an SSD Buffer Pool Extension

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

Christian Lang*
Acelot Inc.
clang@acelot.com

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

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

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

ABSTRACT

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

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

General Terms

Measurement, Performance, Design, Experimentation.

Keywords

Persistence, Solid State Storage, Database Engines, Recovery

1. INTRODUCTION

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

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

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

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

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

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

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

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

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

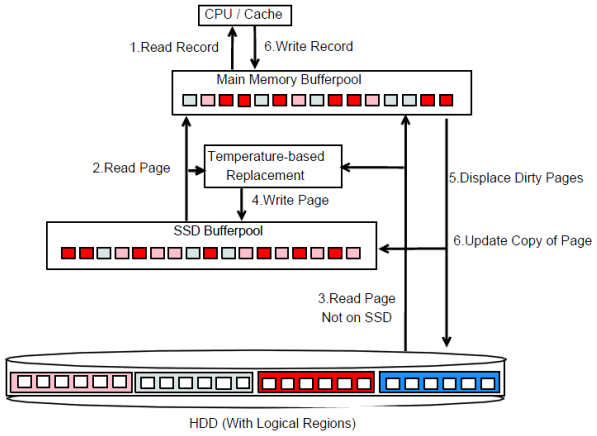

Figure 1: SSD Bufferpools without persistence exploitation

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

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

2. HOW RECOVERY WORKS

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

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

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

3. IMPLEMENTING RECOVERY

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

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

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

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

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

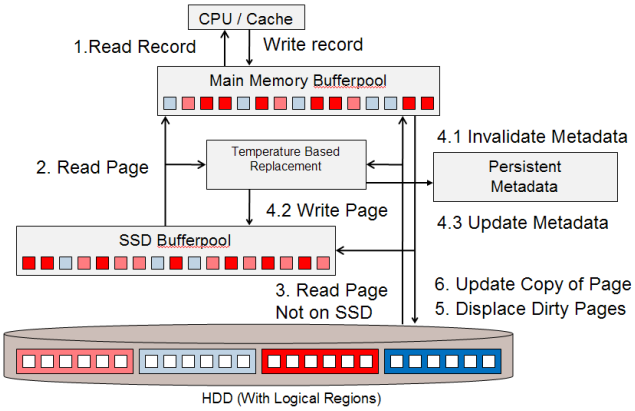

Figure 2: SSD Bufferpools with persistence exploitation

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

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

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

4. CORRECTNESS

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

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

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

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

5. EXPERIMENTAL RESULTS

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

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

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

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

5.1 Hardware Specifications

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

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

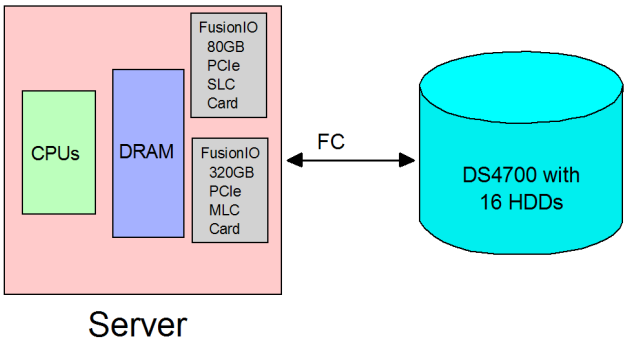

Figure 3: Hardware Experimental Setup

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

Table 1: Hardware specification of the SSDs

5.2 Software Specifications

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

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

5.3 Impact of metadata writes

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

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

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

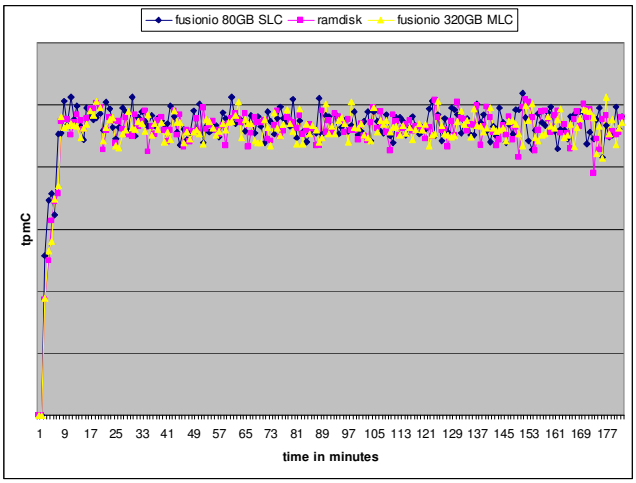

Figure 4: Impact of metadata writes on tpmC for various storage mediums

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

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

5.4 Impact on logging

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

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

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

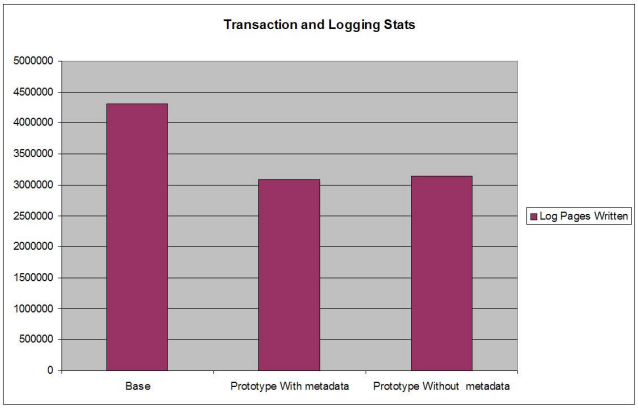

Figure 5: Amount of logging

5.5 Crash Recovery performance

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

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

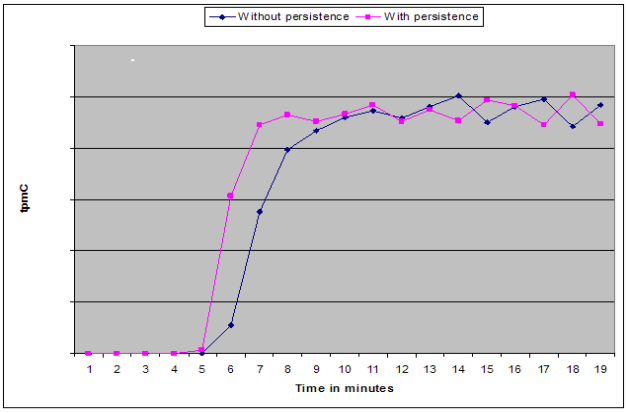

Figure 6: tpmC after crash with prototype

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

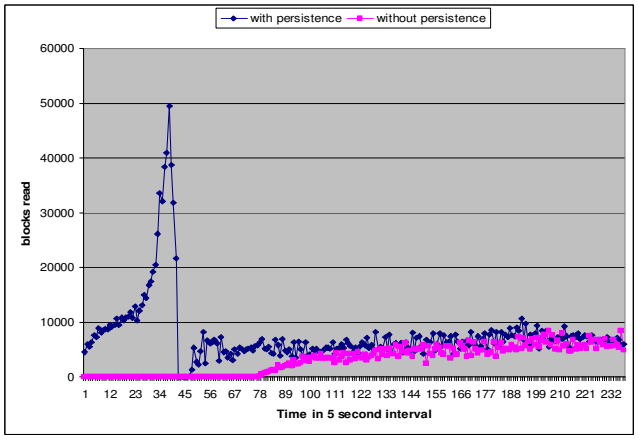

Figure 7: Reads from SSD Bufferpool after crash with prototype

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

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

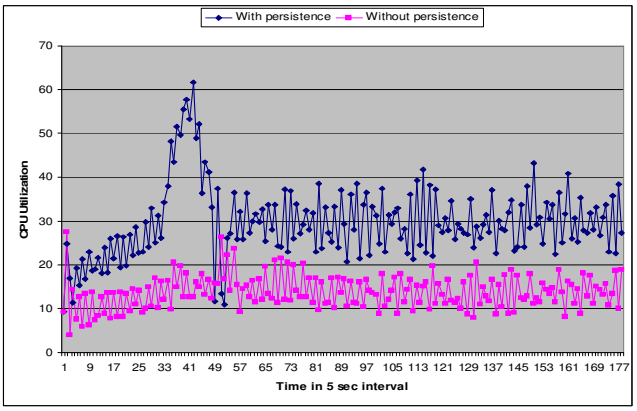

Figure 8: CPU utilization during crash recovery for prototype

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

5.6 Restart performance

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

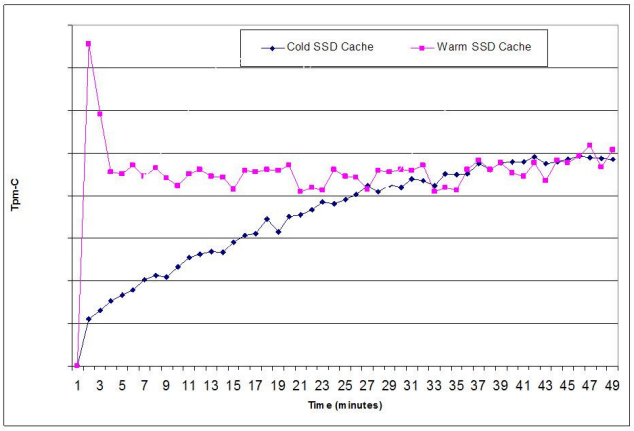

Figure 9: tpmC comparison for restart on prototype

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

6. CONCLUSION

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

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

7. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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