Software Persistent Memory

Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, Jinpeng Wei
Florida International University
{jguerra, lmarm001, dcamp020, ccres008, raju, weijp}@cs.fiu.edu

Abstract

Persistence of in-memory data is necessary for many classes of application and systems software. We propose Software Persistent Memory (SoftPM), a new memory abstraction which allows *malloc* style allocations to be selectively made persistent with relative ease. Particularly, SoftPM's persistent *containers* implement automatic, orthogonal persistence for all in-memory data reachable from a developer-defined *root* structure. Writing new applications or adapting existing applications to use SoftPM only requires identifying such root structures within the code. We evaluated the correctness, ease of use, and performance of SoftPM using a suite of microbenchmarks and real world applications including a distributed MPI application, SQLite (an in-memory database), and memcachedb (a distributed memory cache). In all cases, SoftPM was incorporated with minimal developer effort, was able to store and recover data successfully, and provide significant performance speedup (e.g., up to 10X for memcachedb and 83% for SQLite).

1 Introduction

Persistence of in-memory data is necessary for many classes of software including metadata persistence in systems software [21, 24, 32, 33, 35, 38, 40, 47, 48, 52, 59], application data persistence in in-memory databases and key-value stores [3, 5], and computational state persistence in high-performance computing (HPC) applications [19, 44]. Currently such software relies on the persistence primitives provided by the operating system (file or block I/O) or a database system. When using OS primitives, developers need to carefully track persistent data structures in their code and ensure the atomicity of persistent modifications. Additionally they are required to implement serialization/deserialization for their structures, potentially creating and managing additional metadata whose modifications must also be made consistent with the data they represent. On the other hand, using databases for persistent metadata is generally not an option within systems software, and when their use is possible, developers must deal with data impedance mismatch [34]. While some development complexity is alleviated by object-relation mapping libraries [10], these translators increase overhead along the data path. Most importantly, all of these solutions require substantial application involvement for making data persistent which ultimately increases code complexity affecting reliability, portability, and maintainability.

In this paper, we present *Software Persistent Memory* (SoftPM), a lightweight persistent memory abstraction for C. SoftPM provides a novel form of *orthogonal persistence* [8], whereby the persistence of data (the *how*) is seamless to the developer, while allowing effortless control over *when* and *what* data persists. To use SoftPM, developers create one or more persistent *containers* to house a subset of in-memory data that they wish to make persistent. They only need to ensure that a container's *root structure* houses pointers to the data structures they wish to make persistent (e.g. the head of a list or the root of a tree). SoftPM automatically discovers data reachable from a container's root structure (by recursively following pointers) and makes all new and modified data persistent. Restoring a container returns the container root structure from which all originally reachable data can be accessed. SoftPM thus obviates the need for explicitly managing persistent data and places no restrictions on persistent data locations in the process' address space. Finally, SoftPM improves I/O performance by eliminating the need to serialize data and by using a novel *chunk-remapping* technique which utilizes the property that all container data is memory resident and trades writing additional data for reducing overall I/O latency.

We evaluated a Linux prototype of SoftPM for correctness, ease of use, and performance using microbenchmarks and three real world applications including a recoverable distributed MPI application, SQLite [5] (a serverless database), and memcachedb (a distributed memory cache). In all cases, we could integrate SoftPM with little developer effort and store and recover application data successfully. In comparison to explicitly managing persistence within code, development complexity substantially reduced with SoftPM. Performance improvements were up to 10X for memcachedb and 83% for SQLite, when compared to their native, optimized, persistence implementations. Finally, for a HPC-class matrix multiplication application, SoftPM's asyn-

Function	Description
`int pCAlloc(int magic, int cSSize, void ** cStruct)`	*create a new container; returns a container identifier*
`int pCSetAttr(int cID, struct cattr * attr)`	*set container attributes; reports success or failure*
`struct cattr * pCGetAttr(int magic)`	*get attributes of an existing container; returns container attributes*
`void pPoint(int cID)`	*create a persistence point asynchronously*
`int pSync(int cID)`	*sync-commit outstanding persistence point I/Os; reports success or failure*
`int pCRestore(int magic, void ** cStruct)`	*restore a container; populates container struct, returns a container identifier*
`void pCFree(int cID)`	*free all in-memory container data*
`void pCDelete(int magic)`	*delete on-disk and in-memory container data*
`void pExclude(int cID, void * ptr)`	*do not follow pointer during container discovery*

Table 1: The SoftPM application programmer interface.

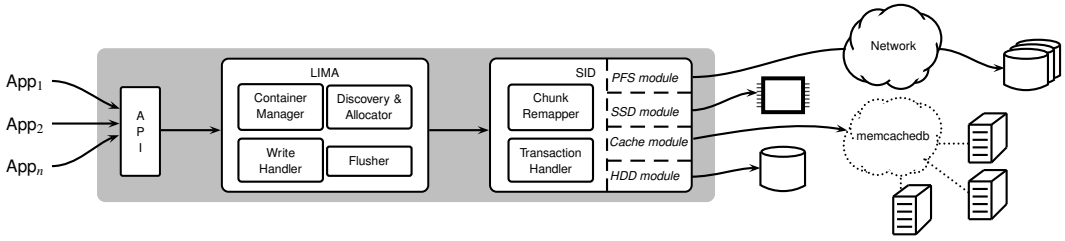

Figure 1: The SoftPM architecture.

Container Structure	Root Usage
`struct c_root { list_t *l; } *cr;`	`id = pCAlloc(m,sizeof(*cr),&cr); cr->l = list_head; pPoint(id);`

Figure 2: Implementing a persistent list. `pCAlloc` **allocates a container and** `pPoint` **makes it persistent.**

chronous persistence feature provided performance at close to memory speeds without compromising data consistency and recoverability.

2 SoftPM Overview

SoftPM implements a persistent memory abstraction called *container*. To use this abstraction, applications create one or more containers and associate a *root structure* with each. When the application requests a persistence point, SoftPM calculates a memory closure that contains all data reachable (recursively via pointers) from the container root, and writes it to storage atomically and (optionally) asynchronously.

The container root structure serves two purposes: *(i)* it frees developers from the burden of explicitly tracking persistent memory areas, and *(ii)* it provides a simple mechanism for accessing all persistent memory data after a restore operation. Table 1 summarizes the SoftPM API. In the simplest case, an application would create one container and create *persistence points* as necessary (Figure 2). Upon recovery, a pointer to a valid container root structure is returned.

2.1 System Architecture

The SoftPM API is implemented by two components: the *Location Independent Memory Allocator* (LIMA), and the *Storage-optimized I/O Driver* (SID) as depicted in Figure 1. LIMA's *container manager* handles container creation. LIMA manages the container's persistent data as a collection of memory pages *marked* for persistence. When creating a persistence point, the *discovery and allocator module* moves any data newly made reachable from the container root structure and located in volatile memory to these pages. Updates to these pages are tracked by the *write handler* at the granularity of multi-page *chunks*. When requested to do so, the *flusher* creates persistence points and sends the dirty chunks to the SID layer in an asynchronous manner. Restore requests are translated into chunks requests for SID.

The SID layer atomically commits container data to persistent storage and tunes I/O operations to the underlying storage mechanism. LIMA's flusher first notifies the *transaction handler* of a new persistence point and submits dirty chunks to SID. The *chunk remapper* implements a novel I/O technique which uses the property that all container data is memory resident and trades writing additional data for reducing overall I/O latency. We designed and evaluated SID implementations for hard drive, SSD, and memcached back-ends.

3 LIMA Design

Persistent containers build a foundation to provide seamless memory persistence. Container data is managed within a contiguous container virtual address space, a self-describing unit capable of being migrated across systems and applications running on the same hardware architecture. The container virtual address space is composed solely of pages marked for persistence including those containing application data and others used to store LIMA metadata. This virtual address space is mapped to

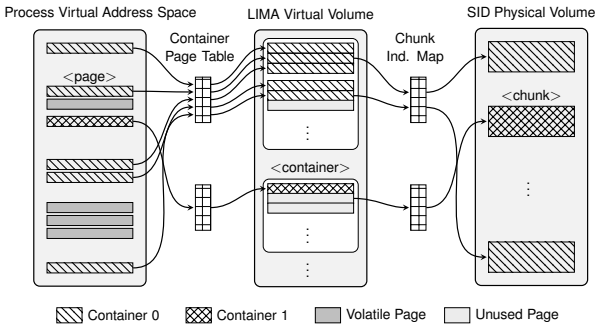

Figure 3: Container virtual address spaces in relation to process virtual address space and LIMA/SID volumes. The container virtual address space is chunked, containing a fixed number of pages (three in this case).

logically contiguous locations within the virtual volume managed by LIMA. SID remaps LIMA virtual (storage) volumes at the chunk granularity to the physical (storage) volume it manages. This organization is shown in Figure 3. The indirection mechanism implemented by SID simplifies persistent storage management for LIMA which can use a logically contiguous store for each container.

3.1 Container Manager

The container manager implements container allocation and restoration. To allocate a new container (`pCAlloc`), an in-memory container page table, that manages both application persistent data and LIMA metadata, is first initialized. Next, the container root structure and other internal LIMA metadata structures are initialized to be managed via the container page table.

To restore a container, an in-memory container instance is created and all container data and metadata loaded. Since container pages would likely be loaded into different portions of the process' address space, two classes of updates must be made to ensure consistency of the data. First, the container metadata must be updated to reflect the new in-memory data locations after the restore operation. Second, all memory pointers within data pages need to be updated to reflect the new memory addresses (pointer swizzling). To facilitate this, pointer locations are registered during process execution; we discuss automatic pointer detection in §5.

3.2 Discovery and Memory Allocation

A core feature of SoftPM is its ability to discover container data automatically. This allows substantial control over what data becomes persistent and frees the developer from the tedious and error-prone task of precisely specifying which portions of the address space must be allocated persistently. SoftPM implements automatic container discovery and persistent memory allocation by automatically detecting pointers in process memory, recursively moving data reachable from the container root to the container data pages, and fixing any *back references* (other pointers) to the data that was moved. In our implementation, this process is triggered each time a persistence point is requested by the application and is executed atomically by blocking all threads of a process only until the container discovery phase is completed; disk I/O is performed asynchronously (§3.4).

To make automatic container discovery possible, SoftPM uses static analysis and automatic source translation to register both *pointers* and *memory allocation* requests (detailed in §5). At runtime, pointers are added either to a *persistent pointer set* or a *volatile pointer set* as appropriate, and information about all memory allocations is gathered. Before creating a persistence point, if a pointer in the persistent pointer set (except those excluded using `pExclude`) references memory outside the container data pages, the allocation containing the address being referenced is moved to the persistent memory region. Forward pointers contained within the moved data are recursively followed to similarly move other new reachable data using an edge-marking approach [30]. Finally, back references to all the data moved are updated. This process is shown in Figure 4. There are two special cases for when the target is not within a recognized allocation region. If it points to the code segment (e.g. function pointers), the memory mapped code is registered so that we can "fix" the pointer on restoration. Otherwise, the pointer metadata is marked so that its value is set to NULL when the container gets restored; this allows SoftPM to correctly handle pointers to OS state dependent objects such as files and sockets within standard libraries. If allocations made by library code are required to be persistent, then the libraries must also be statically translated using SoftPM; the programmer is provided with circumstantial information to help with this. In many cases, simply reinitializing the library upon restoration is sufficient, for instance, we added one extra line in SQLite (see § 6.3.3) for library re-initialization.

3.3 Write Handler

To minimize disk I/O, SoftPM commits only modified data during a persistence point. The *write handler* is responsible for tracking such changes. First, sets of contiguous pages in the container virtual address space are grouped into fixed-size *chunks*. At the beginning of a persistence point, all container data and metadata pages are marked *read-only*. If any of these pages are subsequently written into, two alternatives arise when handing the fault: *(i)* there is no persistence point being created currently – in this case, we allow the write, mark the chunk *dirty*, and its pages *read-write*. This ensures at most one write page fault per chunk between two consec-

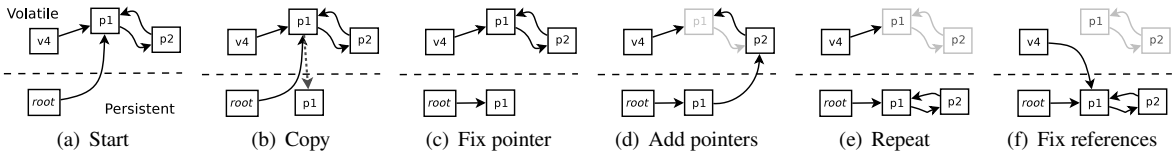

Figure 4: Container Discovery. Grey boxes indicate freed memory.

utive persistence points. *(ii)* there is a persistence point being created currently – then we check if the chunk has already been made persistent. If so, we simply proceed as in the first case. If it has not yet been made persistent, a copy of the chunk is first created to be written out as part of the ongoing persistence point, while write to the original chunk is handled as in the first case.

3.4 Flusher

Persistence points are created asynchronously (via `pPoint`) as follows. First, the flusher waits for previous persistence points for the same container to finish. It then temporarily suspends other threads of the process (if any) and marks all the pages of the container as read-only. If no chunks were modified since the previous persistence point, then no further action is taken. If modifications exist, the flusher spawns a new thread to handle the writing, sets the state of the container to *persistence point commit*, and returns to the caller after unblocking all threads. The handler thread first identifies all the dirty chunks within the container and issues write operations to SID. Once all the chunks are committed to the persistent store, SID notifies the flusher. The flusher then reverts the state of the container to indicate that persistence point has been committed.

4 SID Design

LIMA maps chunks and containers to its logical volume statically and writes out only the modified chunks during persistence points. If a mechanical disk drive is used directly to store this logical volume, I/O operations during a persistence point can result in large seek and rotational delay overheads due to fragmented chunk writes within a single container; if multiple containers are in use simultaneously, the problem compounds causing disk head movement across multiple container boundaries. If a solid-state drive (SSD) were used as the persistent store, the LIMA volume layout will result in undesirable random writes to the SSD that is detrimental to both I/O performance and wear-leveling [22, 31]. The complementary requirement of ensuring atomicity of all chunk writes during a persistence point must be addressed as well. The SID component of SoftPM is an indirection layer below LIMA and addresses the above concerns.

4.1 SID Basics

SID divides the physical volume into chunk-sized units and maps chunks in the LIMA logical volume to physical volume locations for I/O optimization. The *chunk remapper* utilizes the property that all container data is memory resident and trades writing additional data (chunk granularity writes) for reducing I/O latency using device-specific optimizations.

Each physical volume stores *volume-level SID metadata* at a fixed location. This metadata includes for each container the address of a single physical chunk which stores two of the most recent versions of metadata for the container to aid crash recovery (elaborated later). To support chunk indirection, SID maintains a *chunk indirection map* as part of the container metadata. Finally, SID also maintains both an in-memory and on-disk per-container *free chunk bitmap* to locate the chunks utilized by a container. We chose to store per-container free chunk bitmaps to make each container self-describing and as a simple measure to eliminate race conditions when persisting multiple containers simultaneously.

During SID initialization, the free chunk bitmaps for each container stored on the physical volume are read into memory. An in-memory *global* free chunk bitmap obtained by merging the per-container free chunk bitmaps is used to locate free chunks in the physical volume quickly during runtime.

Atomic Persistence. To ensure atomicity of all chunk writes within a persistence point, SID uses persistence *version* numbers. When SID receives a request to create a persistence point, it goes through several steps in sequence. First, it writes all the dirty data chunks; chunks are never updated in place to allow recovery of the previous version of the chunks in case the persistence operation cannot be completed. Once the data chunk writes have all been acknowledged, SID writes the updated free chunk bitmap. Finally, it writes the container's metadata. This metadata includes, the chunk indirection map, the location of the newly written free chunk bitmap, and a (monotonically increasing) version number to uniquely identify the persistence point. Writing the last block of the metadata (the version number) after an I/O barrier commits the persistence point to storage; we reasonably assume that this block gets written to the storage device atomically.

Recovery. SID recovers the same way after both normal shutdowns and crashes. In either case, it identifies the most recent metadata for each container by inspecting their version numbers. It then reads the per-container free chunk bitmaps, and builds the global free chunk bitmap by merging all per-container bitmaps. When the application requests to restore a container, the most recent version of the chunk indirection map is used to reconstruct the container data in memory.

4.2 Device-specific optimizations

Disk Drives. Since sequential access to disk drives is orders of magnitude more efficient than random, we designed a mechanical disk SID driver to employ mostly-sequential chunk layout. The design assumes that the storage device will be performance rather than capacity bound, justifying a fair degree of space over-provisioning for the SID physical volume. Every chunk is written to the nearest free location succeeding the previously written location, wrapping around in a circular fashion. The greater the over-provisioning of the SID physical volume, the higher the probability of finding an adjacent free chunk. For instance, a 1.5X over-provisioning of capacity will result in every third chunk being free on average. Given sufficient outstanding chunk requests in the disk queue at any time, chunks can be written with virtually no seek overhead and minimum rotational delay. Reclaiming free space is vastly simpler than a log-structured design [49] or that of other copy-on-write systems like WAFL [28] because *(i)* the design is not strictly log-structured and does not require multiple chunk writes to be sequential, and *(ii)* reclaiming obsolete chunks is as simple as updating a single bit in the freespace bitmap without the need for log cleaning or garbage collection that can affect performance.

Flash drives. An SSD's logical address space is organized into *erase units* which were hundreds of kilobytes to a few megabytes in size for the SSD units we tested. If entire erase units are written sequentially, free space can be garbage collected using inexpensive *switch merge* operations rather than more expensive *full merge* operations that require data copying [31]. SID writes to the SSD space one erase unit at a time by tuning its chunk size to a multiple of the erase unit size. The trade-off between the availability of free chunks and additional capacity provisioning follows the same arguments as those for disk drives above.

5 Pointer Detection

As discussed in §3, LIMA must track pointers in memory for automatic container discovery and updating pointer values during container restoration. The life-cycle of a pointer can be defined using the following stages: *(i) allocation*: when memory to store the pointer is allocated, *(ii) initialization*: when the value of the pointer is initialized, *(iii) use*: when the pointer value is read or written, and *(iv) deallocation*: when the memory used to store the pointer is freed. Note that, a pointer is always associated with an allocation. In SoftPM, we detect pointers at *initialization*, both explicitly (via assignment) or implicitly (via memory copying or reallocation). Hence, if programs make use of user-defined memory management mechanisms (e.g., allocation, deallocation, and copy), these must be registered with SoftPM to be correctly accounted for.

SoftPM's pointer detection works in two phases. At compile time, a static analyzer based on CIL (C Intermediate Language) [43] parses the program's code looking for instructions that allocate memory or initialize pointers. When such instructions are found, the analyzer inserts static hints so that these operations are registered by the SoftPM runtime. At runtime, SoftPM maintains an *allocation table* with one entry per active memory allocation. Each entry contains the address of the allocation in the process' address-space, size, and a list of pointers within the allocation. Pointers are added to this list upon initialization which can be done either *explicitly* or *implicitly*. A pointer can be initialized explicitly when it appears as an *l-value* of an assignment statement. Second, during memory copying or moving, any initialized pointers present in the source address range are also considered as implicitly initialized in the destination address range. Additionally, the source allocation and pointers are deregistered on memory moves. When memory gets deallocated, the entry is deleted from the allocation table and its pointers deregistered.

Notes. Since SoftPM relies on static type information to detect pointers, it cannot record integers that may be (cast and) used as pointers by itself. However, developers can insert static hints to the SoftPM runtime about the presence of additional "intentionally mistyped" pointers to handle such oddities. Additionally, SoftPM is agnostic to the application's semantics and it is not intended to detect arbitrary memory errors. However, SoftPM itself is immune to most invalid states. For example, SoftPM checks whether a pointer's target is a valid region as per the memory allocation table before following it when computing the memory closure during container discovery. This safeguard avoids bloating the memory closure due to "rogue" pointers. We discuss this further detail in § 8.

Related work. Pointer detection is an integral part of garbage collectors [58]. However, for languages that are not strongly typed, conservative pointer detection is used [12]. This approach is unsuitable for SoftPM since it is necessary to swizzle pointers. To the best of our knowledge, the static-dynamic hybrid approach to *exact*

pointer detection presented in this paper, is the first of its kind. Finally, although pointer detection seems similar to *points-to* analysis [27], these are quite different in scope. The former is concerned about if a given memory *location* contains a valid memory address, while the latter is concerned about *exactly which* memory addresses a memory location can contain.

6 Evaluation

Our evaluation seeks to address the correctness, ease of use, and performance implications of using SoftPM. We compare SoftPM to conventional solutions for persistence using a variety of different application benchmarks and microbenchmarks. In cases where the application had a built-in persistence routine, we compared SoftPM against it using the application's default configuration. Where such an implementation was not available, we used the TPL serialization library [6] v1.5 to implement the serialization of data structures. All experiments were done on one or more 4-Core AMD Opteron 1381 with 8 GB of RAM using WDC WD5002ABYS SATA and MTRON 64GB SSD drives running Linux 2.6.31.

6.1 Workloads

We discuss workloads that are used in the rest of this evaluation and how we validated the consistency of persistent containers stored using SoftPM in each case.

Data Structures. For our initial set of experiments we used the DragonFly BSD [1] v2.13.0 implementation of commonly used data structures including arrays, lists, trees, and hashtables. We populated these with large number of entries, queried, and modified them, creating persistence points after each operation.

Memcachedb [3]. A persistent distributed cache based on memcached [2] which uses Berkeley DB (BDB) [45] v4.7.25 to persistently store elements of the cache. Memcachedb v1.2.0 stores its key-value pairs in a BDB database, which provides a native persistent key value store by using either a btree or a hash table. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB, and compared its performance to the native version using default configurations of the software. To use SoftPM with memcachedb, we modified the file which interfaced with BDB, reducing the LOC from 205 to 40. The workload consisted of inserting a large number of key-value pairs into memcachedb and performing a number of lookups, inserts, and deletes of random entries, creating persistence points after each operation.

SQLite [5]. A popular serverless database system with more than 70K LOC. We modified it to use SoftPM for persistence and compared it against its own persistence routines. SQLite v3.7.5 uses a variety of complex data structures to optimize inserts and queries among other operations; it also implements and uses a custom slab-based memory allocator. A simple examination of the SQLite API revealed that all the database metadata and data is handled through one top-level data structure, called `db`. Thus, we created a container with just this structure and excluded an incorrectly detected pointer resulting from casting an `int` as a `void*`. In total, we added 9 LOC to make the database persistent using SoftPM which include a few more code to re-initialize a library.

MPI Matrix Multiplication. A recoverable parallel matrix multiplication that uses Open MPI v1.3.2 and checkpoints state across processes running on multiple machines.

6.2 Correctness Evaluation

To evaluate the correctness of SoftPM for each of the above applications, we crashed processes at random execution points and verified the integrity of the data when loaded from the SoftPM containers. We then compared what was restored from SoftPM to what was loaded from the native persistence method (e.g. BDB or file); in all cases, the contents were found to be equal. Finally, given that we were able to examine and correctly analyze complex applications such as SQLite with a large number of dynamically allocated structures, pointers, and a custom memory allocation implementation, we are confident that our static and dynamic analysis for pointer detection is sound.

6.3 Case Studies

In this section, we perform several case studies including *(i)* a set of SoftPM-based persistent data structures, *(ii)* an alternate implementation of memcachedb [3] which uses SoftPM for persistence, *(iii)* a persistent version of SQLite [5], a serverless database based on SoftPM, and *(iv)* a recoverable parallel matrix multiplication application that uses MPI.

6.3.1 Making Data Structures Persistent

We examined several systems that require persistence of in-memory data and realized that these systems largely used well-known data structures to store their persistent data such as arrays, lists, trees, and hashtables. A summary of this information is presented in Table 2. We constructed several microbenchmarks that create and modify several types of data structures using SoftPM and TPL [6], a data structure serialization library. To quantify the reduction in development complexity we compared the lines of code necessary to implement persistence for various data structures using both solutions. We report in Table 3 the lines of code (LOC) without any persistence and the additional LOC when implementing persistence using TPL and SoftPM respectively.

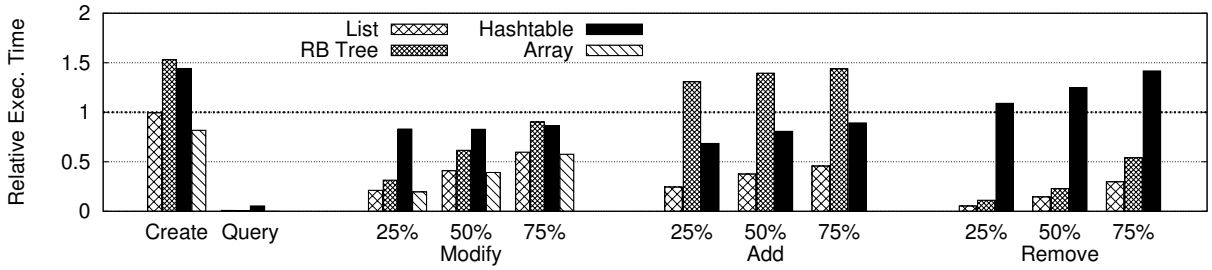

Figure 5: Performance of individual data structure operations. The bars represent the execution time of the SoftPM version relative to a version that uses the TPL serialization library for persistence. We used fixed size arrays which do not support add or remove operations.

Systems	Arrays	Lists	Hash Tables	Trees	C
BORG [11]	✓		✓	✓	✓
CDP [33]				✓	
Clotho [21]	✓	✓			
EXCES [54]	✓	✓	✓	✓	
Deduplication [59]	✓		✓		
FlaZ [38]	✓	✓	✓		
Foundation [48]	✓		✓		
GPAW [41]	✓				
I/O Shepherd [24]	✓			✓	
I/O Dedup [32]	✓			✓	✓
Venti [47]			✓		

Table 2: Persistent structures used in application and systems software. Arrays are multidimensional in some cases. *C* indicates other complex (graphs and/or hybrid) structures were used. This summary is created based on descriptions within respective articles and/or direct communication with the developers of these systems.

Data Structure	Original LOC	LOC for Persistence	LOC to use SoftPM
Array	102	17	3
Linked List	188	24	3
RB Tree	285	22	3
Hash Table	396	21	3
SQLite	73042	6696	9
memcachedb	1894	205	40

Table 3: Lines of code to make structures (or applications) persistent and recover them from disk. We used TPL for Array, Linked List, RB Tree, and Hash Table; SQLite and memcachedb implement custom persistence.

For each data structure we perform several operations (e.g modify) and make the data structure persistent. Note that the TPL version writes entire structures to disk, whereas SoftPM writes only what was modified. For *create*, SoftPM calculates the memory closure, move the discovered data to persistent memory, and write to disk and overhead is proportional to this work. The *query* operation doesn't modify any data and SoftPM clearly outperforms TPL in this case. *modify* only changes existing data values, *remove* reduces the amount of data written by TPL and involves only metadata updates in SoftPM, and *add* increases the size of the data structure increasing both the amount of data and metadata writes with SoftPM. Figure 5 presents the execution times of the SoftPM version relative to the TPL version. Two interesting points are evidenced here. First, for add operations SoftPM outperforms TPL for all data structures except RB Tree, this is due to balancing of the tree modifying almost the entire data structure in the process requiring expensive re-discovery, data movement, and writing. Second, the remove operations for Hashtable are expensive for SoftPM since its implementation uses the largest number of pointer; removing involves a linear search in one of our internal data structures and we are currently working on optimizing this.

6.3.2 Comparing with Berkeley DB

memcachedb is an implementation of memcached which periodically makes the key value store persistent by writing to a Berkeley DB (BDB) [45] database. BDB provides a persistent key value store using a btree (BDB-Btree) or hash table (BDB-HT), as well as incremental persistence by writing only dirty objects, either synchronously or asynchronously. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB. In Figure 6 we compare the operations per second achieved while changing the persistence back-end. SoftPM outperforms both variants of BDB by upto 2.5X for the asynchronous versions and by 10X for the synchronous.

6.3.3 Making an in Memory Database Persistent

SQLite is a production-quality highly optimized serverless database, it is embedded within many popular software such as Firefox, iOS, Solaris, and PHP. We implemented a benchmark which creates a database and performs random insert, select, update, and delete transactions. We compare the native SQLite persistence to that using SoftPM; transactions are synchronous in both cases. Figure 7 shows that SoftPM is able to achieve 55%

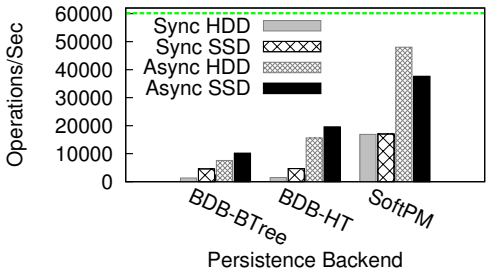

Figure 6: Performance of memcachedb using different persistent back-ends. The workload randomly adds, queries, and deletes 512 byte elements with 16 byte keys. The dashed line represents a memory only solution.

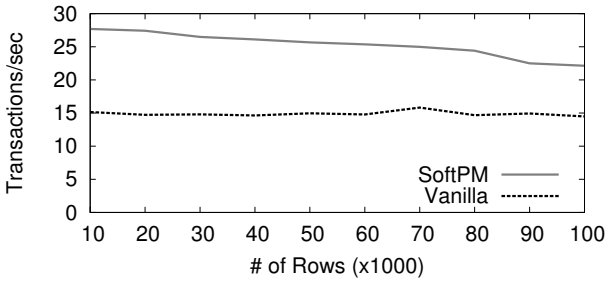

Figure 7: SQLite transactions per second comparison when using SoftPM and the native file persistence.

to 83% higher transactions rate depending on the size of the database. We believe this is a significant achievement for SoftPM given two facts. First, SQLite is a large and complex code base which includes a complete stand alone database application and second, SQLite's file transactions are heavily optimized and account for more than 6K LOC. Further analysis revealed that most of SoftPM's savings arise from its ability to optimize I/O operations relative to SQLite. The reduction in performance improvement with a larger number of rows in the database is largely attributable to a sub-optimal container discovery implementation; by implementing incremental discovery to include only those pointers within dirty pages, we expect to scale performance better with database size in future versions of SoftPM. Figure 8 shows a breakdown of the total overhead including I/O time incurred by SoftPM which are smaller than the time taken by the native version of SQLite. Finally, all of this improvement was obtained with only 9 additional LOC within SQLite to use SoftPM, a significant reduction relative to its native persistence implementation (6696 LOC).

6.3.4 Recoverable Parallel Matrix Multiplication

To compare SoftPM's performance to conventional checkpointing methods, we implemented a parallel matrix multiplication application using Cannon's algorithm [25]. We evaluated multiple solutions, including a no checkpoint non-recoverable implementation, a serialization-based implementation which serializes the matrices to files, and *sync* and *async* versions of SoftPM, in all cases a checkpoint is made after calculating each sub-matrix. For the file-based checkpointing version we added 79 LOC to serialize, write the matrix to a file, and recover from the file. In the SoftPM version, we added 44 LOC, half of them for synchronization across processes to make sure all processes restored the same version after a crash.

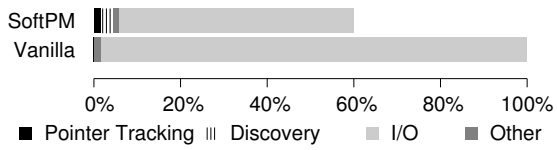

Figure 8: Breakdown of time spent in the SQLite benchmark for 100K rows.

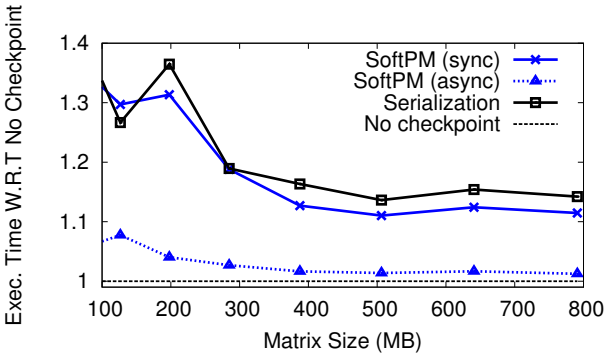

Figure 9: Contrasting application execution times for MPI matrix multiplication using 9 processes.

Figure 9 compares the total execution time across these solutions. Synchronous SoftPM and the serialization solution have similar performance. Interestingly, because of unique ability of overlapping checkpoints with computation, the asynchronous version of SoftPM performs significantly better than either of the above, in fact, within a 1% difference (for large matrices) relative to the memory-only solution.

6.4 Microbenchmarks

In this section, we evaluate the sensitivity of SoftPM performance to its configuration parameters using a series of microbenchmarks. For these experiments, we used a persistent linked list as the in-memory data structure. Where discussed, *SoftPM* represents a version which uses a SoftPM container for persistence; TPL represents an alternate implementation using the TPL serialization library. Each result is averaged over 10 runs, and except when studying its impact, the size of a chunk in the SID

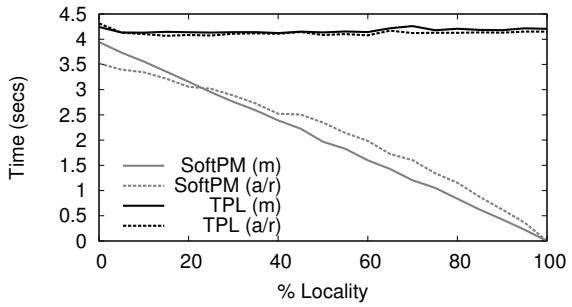

Figure 10: Impact of locality on incremental persistence. Two different sets of experiments are performed: *(m)* where only the contents of the nodes are modified, and *(a/r)* where nodes are added and removed from the list. In both cases the size of the list is always 500MB.

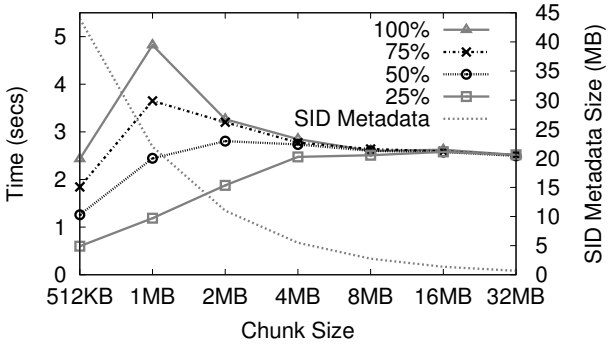

Figure 11: Impact of chunk size on persistence point time. A list of size 500MB is made persistent and individual lines depict for a specific fraction of the list modified.

layer is set to 512KB. To make the linked list persistent, SoftPM and TPL add 5 and 28 LOC, respectively.

Incremental Persistence. Usually, applications modify only a subset of the in-memory data between persistence points. SoftPM implements incremental persistence by writing only the modified chunks, which we evaluated by varying the locality of updates to a persistent linked list, shown in Figure 10. As expected, TPL requires approximately the same amount of time regardless of how much data is modified; it always writes the entire data structure. The SoftPM version requires less time to create persistence points as update locality increases.

Chunk Size. SoftPM tracks changes and writes container data at the granularity of a chunk to create persistence points. When locality is high, smaller chunks lead to lesser data written but greater SID metadata overhead because of a bigger chunk indirection map and free chunk bitmap. On the other hand, larger chunks imply more data written but less SID metadata. Figure 11 shows the time taken to create persistence points and the size of the SID metadata at different chunk sizes.

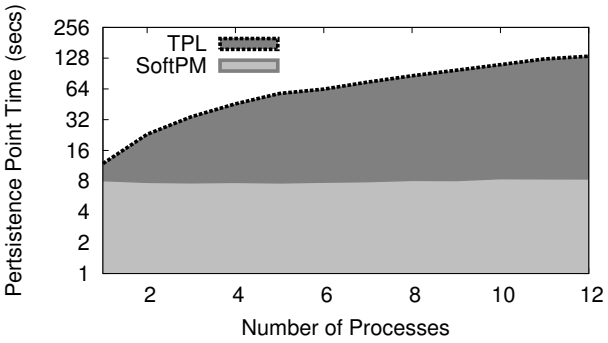

Figure 12: Time to create persistence points for multiple parallel processes. Every process persists a list of size (1GB/number-of-processes).

Parallel Persistence Points. The SID layer optimizes the way writes are performed to the underlying store, e.g. writing to disk drives semi-sequentially. Figure 12 depicts the performance of SoftPM in relation to TPL when multiple processes create persistence points to different containers at the same time. We vary the number of processes, but keep the total amount of data persisted by all the processes a constant. The total time to persist using SoftPM is a constant given that the same amount of data is written. On the other hand, the time for TPL increases with the number of threads, because of lack of optimization of the interleaving writes to the different container files at the storage level.

Percentage of Pointers in Data. Creating a persistence point requires computing a transitive memory closure, an operation whose time complexity is a function of the number of pointers in container data. We varied the fraction of the memory (used by the linked list) that is used to store pointers (quantified as "percentage pointers in data") and measured the time to create a full (non-incremental) persistence point.

We compare performance with a TPL version of the benchmark that writes only the contents of the elements of the list to a file in sequence without having to store pointers. A linked list of total size 500MB was used. Figure 13 shows the persistence point creation times when varying the percentage pointers in data. SoftPM is not always more efficient in creating persistence points than TPL, due to the need to track and store all the pointers and the additional pointer data and SoftPM metadata that needs to be written to storage. The linked list represents one of the best case scenarios for the TPL version since the serialization of an entire linked list is very simple and performs very well due to sequential writing. We also point out here that we are measuring times for registering pointers in the entire list, a full discovery and (non-incremental) persistence, a likely worst case for SoftPM; in practice, SoftPM will track pointers incrementally and

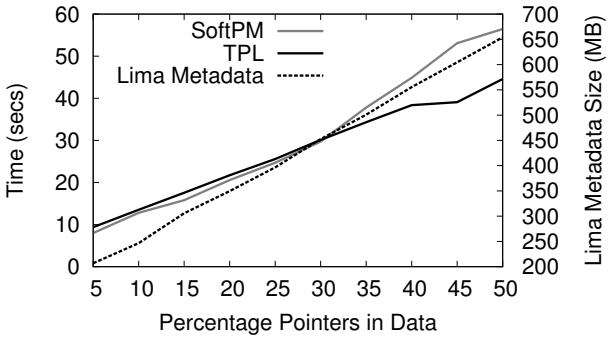

Figure 13: Time to persist a linked list and LIMA metadata size, varying percentage of pointers in data. The total size is fixed at 500MB and node sizes are varied accordingly.

persist incrementally as the list gets modified over time. Further, for the complex programs we studied the percentage pointers in data is significantly lower; in SQLite this ratio was 4.44% and for an MPI-based matrix multiplication this ratio was less than 0.04%. Finally, the amount of SoftPM metadata per pointer can be further optimized; instead of 64 bit pointer locations (as we currently do), we can store a single page address and multiple 16 bit offsets.

7 Related Work

Persistence techniques can be classified into *system-managed*, *application-managed*, and *application-directed*. System-managed persistence is usually handled by a library with optional OS support. In some solutions, it involves writing a process's entire execution state to persistent storage [23, 26, 13]. Other solutions implement persistently mapped memories for programs with pointer swizzling at page fault time [51]. While transparent to developers, this approach lacks the flexibility of separating persistent and non-persistent data required by many applications and systems software. With application-managed persistence [19, 44], application developers identify and track changes to persistent data and build serialization-based persistence and restoration routines. Some hybrid techniques implemented either as persistent memory libraries and persistent object stores have combined reliance on extensive developer input about persistent data with system-managed persistence [14, 20, 36, 34, 46, 50]. However, these solutions involve substantial development complexity, are prone to developer error, and in some cases demand extensive tuning of persistence implementations to the storage system making them less portable. For instance, ObjectStore[34] requires developers to specify which allocations are persistent and their type by overloading the new operator in C++ [4].

Application-directed persistence provides a middle ground. The application chooses what data needs to be persistent, but a library implements the persistence. The earliest instances were persistent object storage systems [16] based on Atkinson's seminal *orthogonal persistence* proposal [8]. Applications create objects and explicit inter-object references, and the object storage system (de)serializes entire objects and (un)swizzles reference pointers [42]. Some persistent object systems (e.g., Versant Persistent Objects [7], SSDAlloc [9], Dali [29]) eliminate object serialization but they require (varying degrees of) careful development that includes identifying and explicitly tagging persistent objects, identifying and (un)swizzling pointers, converting strings and arrays in the code to custom persistent counterpart types, and tagging functions that modify persistent objects.

Recoverable Virtual Memory (RVM) [50] was one of the first to demonstrate the potential for memory-like interfaces to storage. However, its approach has some key limitations when compared to SoftPM. First, RVM's interface still requires substantial developer involvement. Developers must track all persistent data, allocate these within RVM's persistent region, and ensure that dependence relations among persistent data are satisfied (e.g., if persistent structure a points to b, then b must also be made persistent). Manually tracking such relations is tedious and error-prone. Further, developers must specify the address ranges to be modified ahead of time to optimize performance. These requirements were reported to be the source of most programmer bugs when using RVM [39]. Second, RVM's static mapping of persistent memory segments makes it too rigid for contemporary systems that demand flexibility in managing address spaces [37, 53]. In particular, this approach is not encouraged in today's commodity operating systems that employ address-space layout randomization for security [53]. Finally, RVM is also restrictive in dynamically growing and shrinking persistent segments and limits the portability of a persistent segment due to its address range restrictions.

The recent Mnemosyne [56] and NV-Heaps [15] projects also provide persistent memory abstractions similar to SoftPM. However, there are at least two key differences. First, both of the solutions are explicitly designed for non-volatile memories or NVM (e.g., phase-change memory) that are not yet commercially available. Most significantly, these devices are intended to be CPU accessible and byte addressable which eliminates copying data in/out of DRAM [17]. Thus, the focus of these systems is on providing consistent updates to NVM-resident persistent memory via transactions. On the other hand, SoftPM targets currently available commodity technology. Second, neither of these

systems provide the *orthogonal persistence* that SoftPM enables; rather, they require the developer to explicitly identify individual allocations as persistent or not and track and manage changes to these within transactions. For instance, the NV-Heaps work argues that explicit tracking and notification of persistent data ensures that the developer does not inadvertently include more data than she intends [15]. We take the converse position that besides making persistence vastly simpler to use, automatic discovery ensures that the developer will not inadvertently exclude data that does need to be persistent for correctness of recovery, while simultaneously retaining the ability to explicitly exclude portions of data when unnecessary. Further, SoftPM's design, which relies on interposing on application memory allocations, ensures that pointers to library structures (e.g., files or sockets) are reset to NULL upon container restoration by default, thus relieving the developer of explicitly excluding such OS dependent data; such OS specific data is typically re-initialized upon application restart. Finally, feedback about automatically discovered persistent containers from SoftPM can help the developer in reasoning about and eliminating inadvertently included data.

Single level persistent stores as used in the Grasshopper operating system [18] employ pointer swizzling to convert persistent store references to in-memory addresses at the page granularity [55, 57] by consulting an object table within the object store or OS. Updates to persistent pointers are batch-updated (swizzled) when writing pages out. SoftPM fixes pointer addresses when persistent containers get loaded into memory but is free of swizzling during container writing time.

Finally, Java objects can be serialized and saved to persistent storage, from where it can be later loaded and recreated. Further, the Java runtime uses its access to the object's specification, unavailable in other lower-level imperative languages that SoftPM targets.

7.1 SoftPM: A New Approach

SoftPM implements application-directed persistence and differs from the above body of work in providing a solution that: requires little developer effort, works with currently available commodity storage, is flexible enough to apply to modern systems, and enables memory to be ported easily across different address space configurations and applications. Unlike previous solutions in the literature, SoftPM automatically discovers all the persistent data starting from a simple user-defined root structure to implement orthogonal persistence. SoftPM's modular design explicitly optimizes I/O using chunk remapping and tuning I/Os for specific storage devices. Further, SoftPM's asynchronous persistence allows overlapping computation with persistence I/O operations. Finally, unlike most previous solutions, SoftPM implements persistence for the weakly typed C language, typically used for developing systems code using a novel approach that combines both static and dynamic analysis techniques.

8 Discussion and Future Work

Several issues related to the assumptions, scope, and current limitations of SoftPM warrant further discussion and also give us direction for future work.

Programmer errors. SoftPM's automatic discovery of updated container data depends on the programmer having correctly defined pointers to the data. One concern might be that if the programmer incorrectly assigned a pointer value, that could result in corrupt data propagating to disk or losing portions of the container. This is a form of programmer error to which SoftPM seems more susceptible to. However, such programmer errors would also affect other classes of persistence solutions including those based on data structure serialization since these also require navigating hierarchies of structures. Nevertheless, SoftPM does provide a straightforward resolution when such errors get exercised. While not discussed in this paper, the version of SoftPM that was evaluated in this paper implements container versioning whereby previously committed un-corrupted versions of containers can be recovered when such errors are detected. Additionally, we are currently implementing simple checks to warn the developer of unexpected states which could be indicators of such errors; e.g., a persistent pointer points to a non-heap location.

Container sharing. Sharing container data across threads within a single address-space is supported in SoftPM. Threads sharing the container would have to synchronize updates as necessary using conventional locking mechanisms. Sharing memory data across containers within a single address-space is also supported in SoftPM. These containers can be independently checkpointed and each container would store a persistent copy of its data. However, sharing container data persistently is not supported. Further, in our current implementation, containers cannot be simultaneously shared across process address-spaces. In the future, such sharing can be facilitated by implementing the SoftPM interface as library system calls so that container operations can be centrally managed.

Non-trivial types. SoftPM currently does not handle pointers that are either untyped or ambiguously typed. This can occur if a programmer uses a *special* integer type to store a pointer value or if a pointer type is part of a union. These can be resolved in the future with additional hints to SoftPM's static translator from the programmer. Additionally, the runtime could hint to SoftPM

about when a union type resolves to a pointer and when it is no longer so.

Unstructured data. The utility of SoftPM in simplifying development depends on the type of the data that must be made persistent. Unstructured data (e.g., audio or video streams) are largely byte streams and do not stand to benefit as much from SoftPM as data that has structure containing a number of distinct elements and pointers between them. Specifically, unstructured data tends not to get modified in place as much as structured data and consequently they may not benefit from the incremental change tracking that SoftPM implements.

9 Conclusion

For applications and systems that rely on a portion of their state being persistent to ensure correctness for continued operation, the availability of a lightweight and simple solution for memory persistence is valuable. SoftPM addresses this need by providing a solution that is both simple and effective. Developers use the existing memory interfaces as-is, needing only to instantiate persistent containers and container root structures besides requesting persistence points. They thus entirely bypass the complex requirements of identifying all persistent data in code, tracking modifications to them, and writing serialization and optimized persistence routines specific to a storage system. SoftPM automates persistence by automatically discovering data that must be made persistent for correct recovery and ensures the atomic persistence of all modifications to the container; storage I/O optimizations are modularized within SoftPM making it conveniently portable. Recovery of persistent memory is equally simple; SoftPM returns a pointer to the container root via which the entire container can be accessed. We evaluated SoftPM using a range of microbenchmarks, an MPI application, SQLite database, and a distributed memcachedb application. Development complexity as measured using lines of code was substantially reduced when using SoftPM relative to custom-built persistence of the application itself as well as persistence using an off-the-shelf serialization library. Performance results were also very encouraging with improvements of up to 10X, with SoftPM's asynchronous persistence feature demonstrating the potential for performing at close to memory speeds.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Paul Barham, whose feedback substantially improved our work. We are also grateful to Michail Flouris, Haryadi Gunawi, and Guy Laden for sharing the details of the persistent data structures they used in their systems. This work was supported by NSF grants CNS-0747038 and CCF-093796. Jorge Guerra was supported in part by an IBM PhD Fellowship.

References

[1] DragonFlyBSD. http://www.dragonflybsd.org/.

[2] memcached. http://memcached.org/.

[3] memcachedb. http://memcachedb.org/.

[4] ObjectStore Release 7.3 Documentation. http://documentation.progress.com/output/ostore/7.3.0/.

[5] SQLite. http://www.sqlite.org/.

[6] tpl. http://tpl.sourceforge.net/.

[7] Versant. http://www.versant.com/.

[8] M. P. Atkinson. Programming Languages and Databases. In *VLDB*, 1978.

[9] A. Badam and V. S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proc. of NSDR*, 2009.

[10] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, 1998.

[11] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization and Self-optimization in Storage Systems. In *Proc. of USENIX FAST*, 2009.

[12] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience* , 18(9):807–921, September 1988.

[13] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level Checkpointing for Shared Memory Programs. *SIGARCH Comput. Archit. News*, 32(5):235–247, 2004.

[14] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of ACM SIGMOD*, 1994.

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

[16] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. POMS - A Persistent Object Management System. *Software Practice and Experience*, 14(1):49– 71, 1984.

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of SOSP*, 2009.

[18] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computer Systems*, 7(3):289–312, 1994.

[19] E. N. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE TDSC*, 1(2):97–108, 2004.

[20] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, 1989.

[21] M. D. Flouris and A. Bilas. Clotho: Transparent Data Ver-

sioning at the Block I/O Level. In *Proc. of IEEE MSST*, 2004.

[22] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[23] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proc. of the ACM/IEEE SC*, 2005.

[24] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proc. of ACM SOSP*, 2007.

[25] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. *Proc. of the ACM SPAA*, 1994.

[26] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Proc.of SciDAC Conference*, 2006.

[27] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[28] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proc. of the USENIX Technical Conference*, 1994.

[29] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proc. of VLDB*, 1994.

[30] S. V. Kakkad and P. R. Wilson. Address translation strategies in the texas persistent store. In *Proceedings of theUSENIX Conference on Object-Oriented Technologies & Systems*, 1999.

[31] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-memory based File System. In *USENIX Technical*, 1995.

[32] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve i/o performance. In *Proc. of USENIX FAST*, 2010.

[33] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for controller based cdp. In *Proc. of USENIX FAST*, 2007.

[34] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34:50–63, October 1991.

[35] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proc. of USENIX FAST*, 2004.

[36] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of ACM SIGMOD*, 1996.

[37] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. *SIGARCH Comput. Archit. News*, 36(1):115–124, 2008.

[38] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proc. of EuroSys*, 2010.

[39] H. M. Mashburn, M. Satyanarayanan, D. Steere, and Y. W. Lee. RVM: Recoverable Virtual Memory, Release 1.3. 1997.

[40] C. Morrey and D. Grunwald. Peabody: the time travelling disk. In *Proc. of IEEE MSST*, 2003.

[41] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, Jan 2005.

[42] E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE TSE*, 18(8):657–673, 1992.

[43] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, Lecture Notes in Computer Science, 2002.

[44] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. *IEEE MSST*, 2007.

[45] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[46] J. S. Plank, M. Beck, and G. Kingsley. Libckpt: transparent checkpointing under Unix. In *Proc. of the USENIX ATC*, January 1995.

[47] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proc. of USENIX FAST*, 2002.

[48] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. of USENIX ATC*, 2008.

[49] M. Rosenblum and J. Ousterhout. The Design And Implementation of a Log-Structured File System. In *Proc. of ACM SOSP*, 1991.

[50] M. Satyanarayanan, H. Mashburn, P. Kumar, D. C. Steer, and J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of the ACM SOSP*, 1993.

[51] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Intl Workshop on Persistent Object Systems*, September 1992.

[52] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration in Self-managing Storage Systems. In *Proc. of ICAC*, 2005.

[53] The PaX Team. PaX Address Space Layout Randomization (ASLR). *Available online at: http://pax.grsecurity.net/docs/aslr.txt*.

[54] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. In *Proc. of IEEE HPCA*, 2008.

[55] F. Vaughan and A. Dearle. Supporting large persistent stores using conventional hardware. In *In Proc. International Workshop on POS*, 1992.

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

[57] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc of VLDB*, 1992.

[58] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of ISMM*, 1992.

[59] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of USENIX FAST*, Feb 2008.