LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-635611

A Memory-mapped Approach to Checkpointing

D. Wong, G. S. Lloyd, M. B. Gokhale

April 26, 2013

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Memory-mapped Approach to Checkpointing

Daniel Wong, University of Southern California and LLNL
Scott Lloyd, Lawrence Livermore National Laboratory (lloyd23@llnl.gov)
Maya Gokhale, Lawrence Livermore National Laboratory (gokhale2@llnl.gov)

Abstract

We present a lightweight method of checkpointing application-specified persistent data structures to files. With this method, the programmer marks static variables that need to be persistent with a `PERM` qualifier, and uses a persistent variant of a dynamic memory allocator. Static and dynamic persistent data structures reside in a memory-mapped file. At checkpoint time, persistent variables and structures are synced to their associated files. Our method offers a new level of very fast, lightweight checkpoint particularly suitable for node local storage. The checkpoint files can be used by other internode or global checkpoint schemes such as Scalable Checkpoint and Restart (SCR). We describe the checkpoint mechanism and evaluate its performance. Source is available from http://computation.llnl.gov/casc/perm/.

1 Introduction

Long duration supercomputing applications periodically save program state to persistent storage, a process called checkpointing. Typically the programs running on the nodes of the supercomputer synchronize with each other and collectively agree to perform a checkpoint. They each copy program variables to one or more files in the storage area network, and when all checkpoint data has been sent to the global file system, they collectively resume. A compute cluster usually consists of highly interconnected nodes containing CPU and memory, but no local storage. The compute cluster is connected over lower bandwidth links to a storage area network holding the global file system. If nodes go down during a run, the application restarts by reading the last complete checkpoint file(s) from the global file system, restoring program state, and resuming computation from the restored state. The amount of program state saved to a checkpoint file can range from the entire program image [1] to a small selection of application-designated data structures.

As node counts of supercomputing clusters continue to grow, it is generally recognized that future systems will experience shorter mean time between failure (MTBF) than at present, making reliability a dominant concern. For exascale systems, the cost of checkpointing to a global file system will increase to the point that a full system checkpoint will take longer than the compute cluster's MTBF. System architecture level mitigation strategies include incorporating persistent memory into each node or into an I/O node, allowing checkpoints to be collected within the machine. Node level checkpoints are used by the Scalable Checkpoint and Restart (SCR) [2] library in which checkpoint files in memory or local storage are replicated and distributed among nodes in the supercomputer, allowing restart to occur even if nodes are lost to hardware failure. SCR also supports multi-level checkpoints in which every so often, the intra-machine checkpoints are written out to the global file system. Our checkpoint method is complementary to SCR. It produces a checkpoint file that can be distributed within the SCR framework.

The conventional method to do an application-specific checkpoint is to initially call a library function to define the

```
#include "jemalloc/jemalloc.h"

typedef struct {/* ... */} home_st;
PERM home_st *home; /* mark home persistent */

int main(int argc, char *argv[])
{
  int do_restore = argc > 1 &&
                   strcmp("-r", argv[1]) == 0;
  const char *mode = (do_restore) ? "r+" : "w+";

  perm(PERM_START, PERM_SIZE);
  mopen("app.mmap", mode, (size_t)1 << 30);
  bopen("app.back", mode);
  if (do_restore) {
    restore();
  } else {
    home = (home_st *)malloc(sizeof(home_st));
    /* initialize home struct... */
    backup();
  }

  for (;/* each step */;) {
    /* Application_Step(); */
    backup();
  }

  free(home);
  mclose();
  bclose();
  return(0);
}
```

Figure 1: Example checkpoint and restore program with full persistence. The perma allocator replaces the system allocator.

Technical Report LLNL-TR-635611, April 22, 2013

data types of variables to be checkpointed and then to perform the checkpoint by copying variables individually through library calls. To make checkpointing more transparent, we have designed and implemented a new method *perma* for very lightweight checkpointing using memory-mapped files. Using perma, the application programmer designates certain variables to be *persistent*, and further specifies a file to hold the variables. Our library maps the file into the address space of the application, and allocates the variables to addresses within the address range of the mapped file. The variables are used normally within the program code. A call to the perma backup function creates a checkpoint of the variables declared as persistent. The application doesn't have to write code to copy variables individually to the checkpoint file. Perma works with any file within the environment of the application: node local, I/O node file system, or global file system.

Two approaches can be used to introduce persistent variables into an application, namely full persistence and partial persistence. The full persistence approach makes all variables persistent while the partial persistence approach identifies a core set of variables as persistent. Remaining global variables must be reinitialized upon restart. Achieving full persistence is simple to implement; however, reaching partial persistence requires an intimate knowledge of the application source code and data flow within the application and requires application-specific restart code. An example program with full persistence is shown in Figure 1.

In Section 2, we describe the lightweight checkpoint method perma in greater detail and describe the implementation of the underlying dynamic memory allocator. In Section 4, we evaluate the method on the LULESH shock hydrodynamics compact application. Section 5 summarizes and sketches future directions.

2 Perma Method

The perma method uses a persistent heap management library that contains a dynamic-memory allocator (e.g. malloc, free). The perma memory allocator replaces the standard 'C' dynamic memory allocation functions with compatible versions that provide persistent memory to application programs. Memory allocated with the perma allocator will persist between program invocations after a call to a checkpoint function. This function essentially saves the state of the heap and registered global variables to a file which may reside in flash memory or other node local storage. A few other functions are also provided by the library to manage checkpoint files. Global variables in an application can be marked persistent and be included in a checkpoint by using a compiler attribute defined as PERM. The perma checkpoint method is not dependent on the programming model and works with distributed memory or shared memory programs.

2.1 API

The persistent form of the allocation functions can be prefixed with a string of choice when the library is configured. For instance, a user may choose a prefix of `perm_` to form `perm_malloc()`, `perm_free()`, etc. In addition to the allocation functions, the perma library consists of several functions to manage files associated with the persistent heap (see Figure 2). The functions `perm()`, `mopen()`, and `bopen()` should be called before any allocation functions. Global variables are registered as persistent by calling the `perm()` function at run time. By using the `PERM` attribute in a declaration, global variables will be combined by the linker into a contiguous block. Predefined macros give the start address and size of the `PERM` block. The `mopen()` function specifies the memory-mapped file that holds the persistent heap and the `bopen()` function specifies the backup file that will be written during checkpoint. Specifying these files separately allows independent operations on the files, which is particularly important when checkpointing with SCR. The `mflush()` function copies registered global variables to a reserved area in the persistent heap and then flushes any modified pages out to the memory-mapped file. A checkpoint is made with the `backup()` function, which saves registered global variables and a snapshot of the persistent heap to a backup file. Persistent variables and the heap can be restored from a backup file with a call to the `restore()` function.

```
/* Register a block as persistent memory */
int perm(void *ptr, size_t size);

/* Open and map file into core memory */
int mopen(const char *fname, const char *mode,
          size_t size);

/* Close memory-mapped file */
int mclose(void);

/* Flushes in-core data to memory-mapped file */
int mflush(void);

/* Open backup file */
int bopen(const char *fname, const char *mode);

/* Close backup file */
int bclose(void);

/* Backup globals and heap to backup file */
int backup(void);

/* Restore globals and heap from backup file */
int restore(void);
```

Figure 2: Programing Interface

2.1.1 C++ Support

Several macros and a custom C++ allocator are included in the library to support C++. Figure 3 shows the use of the custom allocator (`PERM_NS::allocator<type>`) with the standard template library (STL). The allocation macro `PERM_NEW` uses the placement syntax of `new` to allocate and construct objects. `PERM_FREE` is used for fundamental types that do not have a destructor, whereas `PERM_DELETE` calls the destructor

```
#include <list>
#include "jemalloc/pallocator.h"
using namespace std;

class globals {/* ... */};

PERM globals *home; /* mark home persistent */
PERM list<int, PERM_NS::allocator<int> > plist;

int main(int argc, char *argv[])
{
  ...
  home = PERM_NEW(globals);
  plist.push_back(/* ... */);
  ...
  PERM_DELETE(home, globals);
  return(0);
}
```

Figure 3: C++ example of the perma allocator

for an object and frees the memory. The new and delete operators for individual classes may also be overridden to use the persistent allocation functions.

2.2 Persistent Memory Allocation

Dynamic persistent variables are allocated in memory-mapped pages associated with a file. To create a persistent memory region, we use the `mmap` system call to establish a mapping between a process' address space and a file. We map a file containing the heap into memory using the `MAP_NORESERVE`, `MAP_SHARED`, and `MAP_FIXED` flags. `MAP_NORESERVE` tells mmap to not reserve swap space for the mapping. This allows us to map a persistent heap that is greater in size than the available physical memory in the system. With `MAP_SHARED`, storing to this region of memory is equivalent to writing to the memory-mapped file. `MAP_FIXED` allows us to specify the starting address for the mapping. This is essential for our checkpointing mechanism as we want to keep the address space deterministic across program runs.

2.2.1 Persistent Bookkeeping

An important feature of a persistent dynamic memory allocator is the ability to allocate and deallocate memory across program runs. This requires that the memory allocator's bookkeeping data structure and variables must survive across program runs.

Our persistent memory allocator is built from jemalloc [3]. Jemalloc is a scalable concurrent malloc implementation that manages memory in multiples of 4 MB chunks by default. Bookkeeping data for huge allocations, those larger than a chunk, are stored in a red-black tree. Smaller allocations are usually managed by "arenas," which carve chunks into smaller allocations called page runs. Information about page runs are stored at the beginning of each chunk. The number of arenas is equal to four times the number of processors so that multiple threads can each allocate from separate arenas without lock contention.

In jemalloc, bookkeeping data is held in global variables and in a separate internal heap. For example, the nodes of the red-black tree for huge allocations come from the internal heap and root pointers to these nodes are stored in global variables. In order to allocate and deallocate across program runs, all bookkeeping data must be persistent. To achieve this, the location of the allocator's bookkeeping data is modified to reside in the first chuck of memory-mapped address space.

2.3 Static Persistent Variables

Statically allocated persistent variables must also be preserved. Typically, static variables are pointers to dynamically allocated data structures. These persistent pointer variables allow a program to keep track of allocated memory across program runs. Preserving the state of static persistent variables is achieved though compiler support. Through the use of GCC's section attribute, persistent static variables will be stored in a section of the executable labeled "persistent". During checkpoint, this region of memory is copied to a reserved area in the memory-mapped address space.

2.3.1 ELF Executable Format

The Executable and Linkable Format (ELF) is a file format for executables under Linux. An ELF executable consists of two headers and several sections/segments. The program header contains information about the segments used during run-time. The section header contains information about the sections for linking and relocation. Common sections include *.text* (executable code), *.data* (initialized data), and *.bss* (uninitialized global data).

When an executable is loaded into memory, these sections are loaded into the beginning of the virtual address space. For example, `void *p = perm_malloc(1)`, would normally create the pointer variable `p` in the stack. If `p` were declared as a global variable, it would be stored in the *.bss* section. The issue we have now is that the value of `p` would be lost if persistent state were recovered from a checkpoint. We cannot issue another `perm_malloc` because this would allocate a different region in the virtual address space. In order to recover `p`, we make use of GCC's section attribute. By associating a section attribute to a global pointer variable, we force the global pointer variable to be stored in the "persistent" section. By having all persistent pointer variables stored in the "persistent" section, we can easily backup and restore these addresses pointing to dynamic data structures in the persistent heap. We associate a variable with the persistent section attribute simply by using the `PERM` qualifier as shown in Figure 1. `PERM` is defined as `__attribute__((section("persistent")))`.

2.3.2 Restoring Static Persistent Variables

Static persistent variables are backed up during checkpoint operations. The beginning and end address of the persistent section is stored as a symbol in the executable and is accessible by declaring `extern void *__start_persistent` and `extern void *__stop_persistent`. During checkpoint operations, we copy the persistent section, which is registered with a call to `perm()`, to a reserved area in the memory-

mapped region. On a restore, we simply copy the saved area back to the persistent section in memory. This restores all static persistent state, including persistent pointer variable addresses, to the state of the last checkpoint.

2.4 Checkpointing Mechanism

When a program crashes, we cannot guarantee that the memory-mapped file is in a consistent state, as we can't control when the OS might flush dirty pages out to the file. Therefore, in a checkpoint, we copy the used portion of the mapped persistent region to a backup file. The backup consists of bookkeeping data, persistent variables, and the persistent heap. On restart, we restore these persistent components from the backup file.

2.5 Kernel Parameters

In order for the checkpoint mechanism to function and perform well, several kernel parameters need to be set. Modern Linux kernels features Address Space Layout Randomization (ASLR) techniques to improve computer security by randomly arranging key components of executables, such as the library, heap, and stack. Our checkpointing technique relies on a deterministic virtual address space. ASLR can be disabled by setting /proc/sys/kernel/randomize_va_space to 0.

The Linux kernel also tends to flush dirty pages in memory to their associated memory-mapped file. If the memory-mapped region is very large, this can lead to performance degradation, especially if the memory-mapped file is stored on disk. By default, the Linux kernel flushes dirty pages every 30 seconds and in the background when the fraction of dirty pages in main memory is above 10%. We do not require the kernel to flush dirty pages to the memory-mapped file because we can manually flush dirty pages to disk during our checkpoint call. Since we rollback to a previous checkpoint state, losing dirty pages during a program crash is not important. Because of this, any flush to disk in between checkpoints can degrade performance drastically. To disable periodic page flush we set /proc/sys/vm/dirty_writeback_centisecs to 0. To reduce the number of page flushes due to a high ratio of dirty pages, we set /proc/sys/vm/dirty_background_ratio and /proc/sys/vm/dirty_ratio to 100.

3 SCR Integration

Integration of the perma method with SCR is straightforward since SCR uses a similar file-based checkpoint model. As shown in Figure 4, the SCR routines are added around the perma `backup()` function. During a checkpoint, the persistent heap of each node is saved to the SCR file framework. Since nodes query SCR for a new filename before each checkpoint, `bopen()` is called each time.

4 Evaluation

Our evaluation uses a shock hydrodynamics compact application recently developed at LLNL to simulate the relative motion of materials when subjected to force. The LULESH code [4] is derived from the Lagrangian part of ALE3D, the Arbitrary Lagrangian Eulerian program. LULESH simulates the Sedov blast wave problem [5] in three spatial dimensions. We evaluate the OpenMP threaded version of this C++ code [6], which has 3117 lines of code. With the help of the code author, we added checkpointing to this program.

```
template<class T>
struct PersistentType {
  typedef std::vector<T,PERM_NS::allocator<T> >
          vector;
};

PERM struct Domain {
  ...
  /* coordinates */
  PersistentType<Real_t>::vector m_x;
  PersistentType<Real_t>::vector m_y;
  PersistentType<Real_t>::vector m_z;
  ...
} domain;
```

Figure 5: Persistent STL data structures in C++

While the bulk of the LULESH code remained unchanged, a few additions and modifications were necessary to use the perma checkpoint method. To designate persistent variables, we added the PERM attribute to the `domain` global data structure as in Figure 5. Persistent arrays within this structure were designated as `PersistentType<Real_t>::vector`. In the program main, we added the initialization calls for the perma library, and the backup call to perform the checkpoints periodically in the simulation loop, similar to the example shown in Figure 1. Inserting perma checkpointing into this program increased the code size to 3172, of which 13 lines actually do the checkpoint. The other lines are associated with adding the PERM attribute and adding the persistent allocator to extend the STL template.

For comparison purposes, we also wrote a checkpoint version that explicitly wrote each persistent variable to a file as raw binary data, using C file I/O. For restart, each variable was read back in the same order it had been checkpointed. This version represents the lower bound on checkpoint overhead: only the variables needed for restart are written to the file; the data is not encoded to make it readable in a general format (e.g. HDF5); and C file I/O is used as opposed to C++. This version takes 3435 lines of code. We label this the manual approach.

The experiments distinguish three categories of checkpointing (no checkpointing, checkpoint, and checkpoint followed by sync), two methods to do a checkpoint (manual and perma), and three problem sizes (45 element edges, 68, and 90). A checkpoint file was written every 30 seconds or so. For the size 45 case, four checkpoints were written, for 68 there were sixteen checkpoints, and for 90 there were 52 checkpoints. For all experiments, the checkpoint file was written to node local storage in /tmp. The experiments were run on a single node, an 8-core Intel with 8 GB memory running RedHat 5 (2.6.18) with the configuration options described in Section 2.5. For

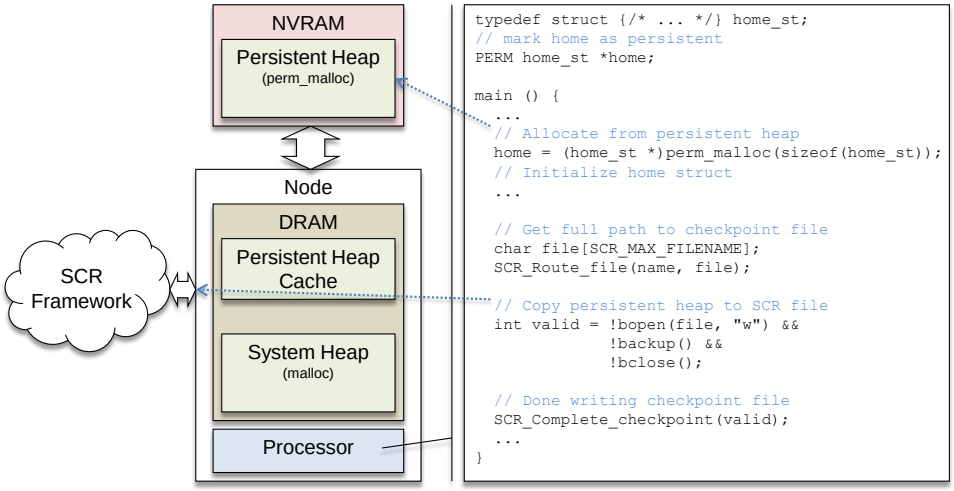

Figure 4: Checkpoint persistent heap with SCR

Table 1: Checkpoint size (MB)

Version	45	68	90
Manual	28.00	97.22	224.47
Perma	54.52	201.30	394.30

Table 2: Relative run times and lines of code

Description	45	68	90	LOC
Base: no checkpoint	1	1	1	1
jemalloc, no checkpoint	0.989	1.089	1.050	1.013
manual checkpoint	1.020	1.024	1.027	1.102
perma checkpoint	1.002	1.090	1.052	1.018
manual checkpoint, sync	1.040	1.086	1.175	1.102
perma checkpoint, sync	1.049	1.090	1.054	1.018

each problem size, the timing results are normalized to the run time of the base OpenMP version for that problem size. Table 1 shows the sizes of the checkpoint files, and Table 2 shows run times and lines of code.

Table 1 shows that the perma method requires more storage than the manual method. This is because static persistent data, the allocator's bookkeeping data structures, and unused heap space are written to the checkpoint file in addition to the actual data that is being checkpointed. Since our usage model assumes node local storage, the extra space is not significant.

In our evaluation, we measured the cost of using the perma allocator without actually doing a checkpoint. This compares the jemalloc allocator with the standard C library malloc. The results show that for the size 45 case, there is no performance penalty for using jemalloc. In fact, it is a bit faster than the base case. As the problem size grows, using jemalloc shows overhead of 8.9% for size 68 and 5% for size 90.

The checkpoint versions that don't sync out the checkpoint file are understandably faster than those that sync the file out to the storage controller. We measure both because of the use case in which this checkpoint method is combined with internode or global checkpoint methods. It is not necessary to sync out the file in our checkpoint code if SCR is going to distribute it.

Of the two variants that don't sync the checkpoint file, the perma checkpoint is faster than the manual approach for size 45, but has 3% more overhead for size 68 and 4% more overhead for size 90. When the file sync is done, perma has 0.9% more overhead for the 45 size and 6.6% more overhead for size 68 than the manual approach. However, for size 90, manual has 2.1% more overhead than perma. Overall, perma is competitive to the lower bound on checkpoint overhead for the measured problem and sizes.

4.1 Discussion

The checkpoint method discussed in this work is novel relative to the present state of practice in checkpointing HPC codes. In current practice, the checkpoint data is written often in a portable, self-describing format that is used for analysis and visualization as well as checkpoints. Often, a library such as silo [7] is used by the application programmer to describe the particular high level data structure to be checkpointed, assigning textual names to various components. This description is stored in the file along with the data that is passed to the library during a checkpoint. The data file is thus self-describing and usable through the library to many other programs.

In contrast, our checkpoint file is specific to a particular version of a program compiled with a specific compiler and run with specific libraries and OS. All those have to be the same to allow valid restart. It is intended for the exascale environment in which doing node local checkpoints might be the only scalable alternative, and the checkpoint files are needed only for a few hours. These lightweight checkpoints could augment a less frequent heavy weight checkpoint to be used for visualization and analysis.

In contrast to the practice of writing out persistent data explicitly, one data structure at a time, our library provides a generic method of checkpointing. The application programmer merely allocates variables from a persistent heap. It is not necessary to write code to copy each persistent variable to the

file. Similarly, on restart, the entire persistent environment is restored in a few memcpy operations. From a software engineering perspective, this greatly reduces the effort and potential bugs, as evidenced in the lines of code comparison. Perma shows a 1.8% LOC overhead over the base code, while the manual method incurs a 10.2% overhead.

5 Conclusions

This work describes a new method of checkpointing with memory-mapped files. It is complementary to inter-node checkpoint methods such as SCR, and could also be used with global methods such as PLFS [8]. Our method could be used in conjunction with future persistent memory technology such as 3D PCM directly attached to the memory bus [9]), Mnemosyne [10], or NV-Heaps [11]. Our approach is agnostic to the storage interface and can accommodate local disk, high performance SSD, or PCM attached to the memory bus. Like Mnemosyne, we take advantage of named linker sections and implement a persistent dynamic memory allocator. Our allocator is based on jemalloc rather than hoard or dlmalloc.

Our approach is orthogonal to the inter-node parallelization method, and therefore should work equally well with MPI, GA, and other methods. Future work is to test this premise by using perma in MPI codes and with other programming models.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We are grateful to Jeff Keasler for helping to insert checkpoint code into LULESH.

References

[1] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Proceedings of SciDAC 2006*, June 2006.

[2] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCan 2006: The Technical BSD Conference*, May 2006.

[4] Rich Hornung, Jeff Keasler, and Maya Gokhale. Hydrodynamics challenge problem. Technical Report LLNL-TR-490254, LLNL, 2011.

[5] L. I. Sedov. Similarity and dimensional methods in mechanics, 1959.

[6] Comp-LLNL. UHPC shock hydrodynamics challenge problem. `http://computation.llnl.gov/casc/ShockHydro/`, 2011.

[7] WCI-LLNL. Welcome to silo: A mesh and field i/o library and scientific database. `http://wci.llnl.gov/codes/silo/`, 2010.

[8] John Bent, Garth Gibson, Gary Grider, et al. PLFS: a checkpoint file system for parallel applications. In *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '09, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS '11*, pages 91–104. ACM, 2011.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS '11*, pages 105–118. ACM, 2011.