RICE UNIVERSITY

General-purpose Programming Techniques for Emerging Systems with Non-volatile Byte-addressable Random Access Memory

by

Kumud Bhandari

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
Research Professor in Computer Science

Keith D. Cooper
L. John and Ann H. Doerr Professor in
Computational Engineering

Peter J. Varman
Professor in Electrical and
Computer Engineering
Professor in Computer Science

Houston, Texas

January, 2018

ABSTRACT

General-purpose Programming Techniques for Emerging Systems with Non-volatile Byte-addressable Random Access Memory

by

Kumud Bhandari

Emerging byte-addressable non-volatile random access memory (NVRAM) technologies allow persistent data to be accessed and manipulated using CPU load and store instructions. The persistent data in NVRAM can be persisted in the same format as it is manipulated and re-used across execution cycles. In order to achieve this persist-reuse programming idiom, persistent data need to be kept consistent across restarts and tolerated failures, i.e. certain data invariants need to be maintained. As persistent data may be cached in volatile structures such as CPU cache and memory instructions may be reordered, failures or power cycle could violate these invariants and leave persistent data in an inconsistent state. This thesis work explores general-purpose software techniques that minimize the burden on programmers to maintain persistent data consistency under this persist-reuse model. It further focuses on persistent memory management techniques to simplify NVRAM programming. In this aspect, this thesis work identifies and addresses challenges associated with designing a failure-safe and leak-free persistent memory allocator that is also interoperable with various persistent programming libraries. Additionally, it presents software techniques to minimize de-/allocation overhead, avoid persistent leaks, and discusses a memory allocator design approach that enhances the programmability of NVRAM programming libraries. This thesis work shows that these software techniques can

enable the failure-safe use of NVRAM without placing an undue burden on programmers or incurring significant performance overheads.

Acknowledgments

I thank my Ph.D. thesis advisor Prof. Vivek Sarkar for his invaluable support throughout my Ph.D. studies. I would also like to thank my Ph.D. thesis committee members for their feedback on my thesis to make this thesis better.

I extend my gratitude to the current and past members of the Habanero Extreme-scale Software team for helping me in numerous ways to conduct my research during my graduate studies. I also thank many student and staff members of the Rice Computer Science Department who have helped and supported me throughout my time at Rice University.

A large portion of this thesis work was carried out at Hewlett-Packard Labs over a number of extended internships during my graduate studies. I would like to thank my mentors at HP Labs Dhruva Chakrabarti, and Hans-J. Boehm for providing me with guidance and support during these internships and beyond. I would also like to thank Haris Volos at HP Labs for helping me evaluate my work against Mnemosyne. I would also like to thank him for sharing his experience and wisdom related to his work on NVRAM. Additionally, I would like to thank Harumi Kuno, Rob Schreiber, Terence Kelly, and Brad Morrey at HP Labs for guidance, support, and fruitful discussions.

This thesis was partially supported by the US Department of Energy under Cooperative Agreement no. DESC0012199 and the Go Fellowship from Oakridge National Lab. I would like to extend my gratitude to Joel Denny, and Jeffrey Vetter for many fruitful meetings and discussions we had related to persistent memory allocation and programming.

Contents

List of Figures

Chapter 1

Introduction

Limitations in current DRAM technology scaling [1–3] have prompted research in alternate memory technologies. Almost all alternatives being explored such as Memristor [4], Phase Change Memory (PCM) [5, 6], and 3D XPoint [7] are non-volatile in nature. Such non-volatile memory (NVRAM) combines the byte-addressability of DRAM with the persistence of hard disks. In the near future, NVRAM may at least partially replace DRAM, making persistent data accessible through CPU load and store instructions. With NVRAM, persistent data can be manipulated in the same format as stored, thus requiring no expensive and cumbersome conversion. Consider an example where a linked list has to be saved in a traditional filesystem in the absence of NVRAM. Such a linked-list has to be serialized into a stream of bytes representing its state to save to a file and deserialized later on before the CPU can again access the data in some later time, possibly after a system restart. With NVRAM, a programmer can write code that can directly allocate persistent linked-list nodes, and manipulate this single copy of the linked-list. Thus, NVRAM opens up an opportunity to have in-memory object persistence so that program states that outlive the creating process can be preserved, shared, and reused. Using this persist-and-reuse model, a quick restart of an application from an intermediate state appears a reality. While starting, an application looks for existing data that it can reuse. If present, the application adjusts its context and instead of computing from scratch, merely reuses the existing data for the rest of its computation.

Although using NVRAM as traditional volatile memory is trivial, taking advantage of in-memory durability is challenging in the presence of system failures. Updates to persistent data may not become visible to NVRAM in the order intended due to volatile CPU caches

or store re-ordering allowed by the memory model. As a result, an interruption such as a power failure may introduce inconsistencies (e.g. dangling pointer) to persistent data in NVRAM. This thesis work addresses several aspects of NVRAM programming such as failure resilient storage of user data in NVRAM, and persistent memory management with the central goal of enabling programmers to take advantage of persistence provided by NVRAM while minimizing the direct programming burden on them.

1.1 Thesis Scope

System experts may choose to exploit persistence provided by NVRAM in a highly specialized setting such as to improve database performance [8] or to design in-memory filesystems [9]. In contrast to such specific use cases of NVRAM, this thesis discusses techniques to make persistence provided by NVRAM accessible to programmers for general purpose programming. Using software techniques described in this thesis work, programmers can allocate and store everyday data structures such as a linked-list or a queue in NVRAM, and reuse them across system restarts by adjusting the programming context based on the state of the persistent data. In this direction, this thesis work discusses low-overhead techniques for developing applications to take advantage of NVRAM persistence, addresses challenges in managing persistent memory and identifies the interplay that exists between persistent memory approach and NVRAM programmability.

1.2 Thesis Statement

Advances in programming models and system software for NVRAM can enable fault-tolerant use of NVRAM without placing an undue burden on programmers or incurring significant performance overheads.

1.3 Contributions

This dissertation makes the following contributions:

- A novel approach for failure consistency that combines lock-based transactional semantics with copy-on-write. We show that copy-on-write mechanism can be combined with lock-based transactional semantics to reduce performance overhead while maintaining programmability
- A design and implementation of a real-world NVRAM application. We designed and implemented an NVRAM version of the commercially used MDB key-value store and characterized its performance overheads
- The first published detailed assessment of the challenges involved in designing an interoperable, leak-free persistent memory allocator.
- The first published leak-free persistent memory allocator that is interoperable with numerous NVRAM programming library. To the best of our knowledge, we present the design and the implementation of the first NVRAM memory allocator that is drop-in replaceable with C/C++ standard malloc/free, that is leak-free and can be used with various existing NVRAM libraries.
- A novel offline recovery and conservative garbage collection approach to prevent failure-induced leaks, and reduce the online cost of de-/allocation.
- A novel approach for precise offline garbage collection over relative heap addresses using persistent type information.

1.4 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 presents a survey of emerging `NVRAM` technologies, their performance characteristics and extensions in Instruction Set Architecture (ISA) to facilitate `NVRAM` programming
- Chapter 3 introduces persist-reuse model for `NVRAM` programming and compares it with other traditional approaches to persistence such as checkpoint-restart and orthogonal persistence

- Chapter 4 presents a novel approach to failure consistency that combines a lock-based transactional approach with a copy-on-write mechanism to reduce persistence overhead. It also presents the design and implementation of an NVRAM version of the MDB key-value store, a commercially used backend for OpenLDAP.
- Chapter 5 presents an assessment of challenges associated with designing a leak-free interoperable persistent memory allocator and the design of such a memory allocator for NVRAM
- Chapter 6 presents the design and the implementation of memory allocator that supports relative addressing and precise garbage collection based on persistent types
- Finally, chapter 7 presents future work and conclusion

Chapter 2

Survey of NVRAM Technologies and Hardware Support

2.1 NVRAM Device Technologies

Several different device technologies are being explored by hardware manufacturers to make NVRAM available in future systems. Some of these technologies which have been productized are described below [10]:

- *Ferroelectric RAM (FRAM):* uses ferroelectric capacitor formed from a common ferroelectric material to store bits represented by two reversible polarization states.
- *Magnetic RAM (MRAM):* uses a two-layer magneto-resistive structure for data storage and measures the resultant difference in resistance between the layers to readout information.
- *Spin Transfer Torque RAM (STTRAM):* uses a spin-polarized current to modify the orientation of a magnetic layer in a magnetic tunnel junction (two ferromagnetic layers separated by a thin insulator) to store bits.
- *Phase Change Memory (PCRAM):* uses two reversible phases – amorphous and the crystalline states – of chalcogenide glass.

Given the differences in underlying technologies, future NVRAM devices may vary in their access times. Table 2.1 lists access times – both write (W) and read (R) – for various

Device Types	FRAM	MRAM	STTRAM	PCRAM
Access Time (W/R)	50/75ns	12/12ns	10/10ns	100/20ns

Table 2.1: Access times for various NVRAM device technologies [10]

.

Device Types	NAND Fash	HDD	DRAM
Access Time (W/R)	200/25 us	9.5/8.5ms	10/10ns

Table 2.2: Access times for various traditional device technologies [10]

.

NVRAM device technologies. Table 2.2 lists access times for DRAM, magnetic hard disk drive (HDD) and NAND Flash block devices for comparison. As seen from these tables, many technologies such as STTRAM and MRAM have access latencies comparable to the current DRAM technology. Others such as PCRAM have higher write latency but are still orders of magnitude better than NAND Flash or HDD. The endurance of these devices also vary. PCRAM, for instance, have endurance similar to NAND Flash (10^5 cycles) whereas FRAM, MRAM, and STRAM have endurance similar to DRAM (10^{16} cycles) [10]. Unlike DRAM which has retention of 64ms (without refresh), NVRAM devices listed in table 2.1 have retention same as NAND Flash devices (10 years). Addressing device endurance and wearability is beyond the scope of this thesis.

2.2 Hardware Support

Traditionally, CPU cache, in general, is functionally invisible to a program that manipulates data in DRAM. The stores from a CPU can become visible in DRAM out of program order as long as memory consistency model along with cache coherence protocol ensures that it becomes visible to other CPUs in a correct order. This may not be true for data stored in NVRAM. In future systems, data stored in NVRAM is expected to be accessible using CPU load and store instructions. Consider an example below where some data is being written before the valid bit is being set.

```
1: store data "moon"
2: store valid "1"
```

If store 2 becomes visible in persistent domain (e.g. NVRAM) and power recycle occurs while store 1 is only present in a transient domain, e.g. CPU cache, the persistent data

invariant is violated as a valid bit is set when the data field in NVRAM contains garbage. Therefore, being able to order stores visibility to NVRAM is crucial to maintaining persistent data consistency. Given volatile write-back CPU caches may continue to exist in future systems with NVRAM and may cache persistent data updates, a persistent memory application needs to be able to make these cache lines visible in NVRAM in a correct order.

2.3 ISA Extensions:

Intel x86-64 ISA traditionally provides CLFLUSH instruction to invalidate a cache line by its virtual address such that it gets eventually written back to the memory system. This instruction is supported by most Intel x86-64 bit CPUs. This instruction has several drawbacks. First, it stalls the CPU pipeline and thus serializes the execution. Second, it invalidates the cache line, thus next read/write to the cache line is a miss. Lastly, it only evicts data to the memory subsystem and does not guarantee that the data has reached the persistent domain.

In anticipation of NVRAM, Intel announced an extension to its x86-64 bit ISA [11], which includes CLFLUSHOPT, an optimized version of CLFLUSH. Unlike CLFLUSH, it does not stall CPU (unordered) and thus the execution of one or more CLFLUSHOPT may overlap. A CLFLUSH operation is only ordered by a full memory fence (e.g. MFENCE), whereas CLFLUSHOPT is ordered by store fence (e.g SFENCE). Intel also introduced another instruction, namely CLWB, which is ordered by SFENCE and writes back a cache line without invalidating it.

All the instructions mentioned above ensures that a cache line reaches a write pending queue (WPQ) in the memory controller. In Intel systems with NVRAM, WPQ is expected to be in the persistent domain. In this light, Intel deprecated the instruction PCOMMIT which was introduced briefly as a means to flush WPQ in systems where WPQ in a memory controller was not incorporated into the persistent domain. Combination of CLWB and SFENCE is enough to order the visibility of persistent stores in Intel architecture where WPQ is in the persistent domain [12]. This is expected to be the case for all future systems

with NVRAM.

Likewise, ARM has introduced an instruction DC CVAP to clean virtual address to the point of persistency in Armv8.2 [13].

2.4 Research Ideas/Prototypes

Several ideas have been proposed to make persistent stores visible in NVRAM while exposing persistent stores concurrency as much as possible at the hardware level. We discuss some of them below.

Epoch Barrier: Condit et al. proposed a hardware modification in CPU cache to separate persistent writes into epochs [14]. A set of writes are separated into epoch by epoch barrier. Each cache line is tagged with epoch id. Whenever a cache line belonging to a specific epoch is evicted, it triggers cascading writes of all cache lines belonging to earlier epochs. This approach requires tagging cache line with epoch id and tracking oldest in-flight epoch per CPU.

Strand Persistency: Pelley et. al. suggest a mechanism where a set of stores from the same CPU are divided into strands by a strand barrier [15]. Stores from different strands are made visible concurrently. Within a strand, persist barrier can be used to order store visibility. This approach requires store dependence tracking at the hardware level for ordering requirements.

2.5 Summary

The access latency for most future NVRAM devices seems to be similar to DRAM. Furthermore, with the extension of persistence domain to the memory controller and write pending queue within it (as seems to be the case for Intel architecture), write latency of actual NVRAM devices may not be of a large concern to persistent memory programmers.

ISA architecture is continuously evolving to enable programmers to use NVRAM as a byte-addressable persistent medium without incurring a large performance penalty for keeping persistent data consistent across power recycles and failures. Systems with extended architecture and NVRAM are still not widely available to programmers and researchers, and the ISA itself appears to be continually evolving. Several other approaches requiring hardware modifications have been proposed but are limited to research ideas or prototypes. A large body of work on programming models and programming libraries predates these ISA extensions. Simulation-based studies are equally likely to be inaccurate in the face of evolving device technology and system architecture. Hence, in the absence of extended ISA and/or due to lack of wide availability of new hardware support, programming library and runtime developers/researchers have largely relied on existing instructions such as CLFLUSH and MOVNTQ (move with non-temporal hint) to design programming libraries for NVRAM and to roughly estimate their performance [16–19]. This may change in the future with better access to new hardware.

Chapter 3

NVRAM Persist-Reuse Programming Model

Persistent data stored in NVRAM are useful only if they are consistent; in particular, their invariants must be preserved. Even in the presence of NVRAM, there will be volatile buffers and caches in the memory hierarchy, simply because of the performance advantages they provide. This implies that during program execution, some of the states may reside in volatile structures and the rest in NVRAM. Figure 3.1 shows a simplified version of the architectural model that we assume; it is in part based on the Intel x86-64 architecture [20]. The CPU core may temporarily keep each store to memory in a store buffer. The store buffer improves performance by hiding the latency of cache and memory accesses. Certain instructions such as memory fences result in draining the store buffer. As shown in figure 3.1, we assume that DRAM and NVRAM may co-exist in the same memory system.

As figure 3.1 shows, even in the presence of NVRAM, volatility remains an important part of the memory hierarchy. If the program crashes because of a hardware or power failure, only the state present in NVRAM at the time of the crash will survive a restart. Any state that was present in volatile structures at the time of the crash is lost. Hence, the challenge is to ensure that at any point of program execution, there is enough information on NVRAM to reconstruct a consistent state of the program data structures.

Using low-level hardware primitives described in chapter 2 to maintain the consistency of persistent data stored in NVRAM diminishes programmability. Several NVRAM programming libraries have been developed which enable programmers to maintain persistent data in NVRAM and manipulate them in a fail-safe manner [16, 17, 21]. These independently developed libraries support the common persist-reuse model described below. Figure 3.2 shows pseudocode written using this persist-reuse model.

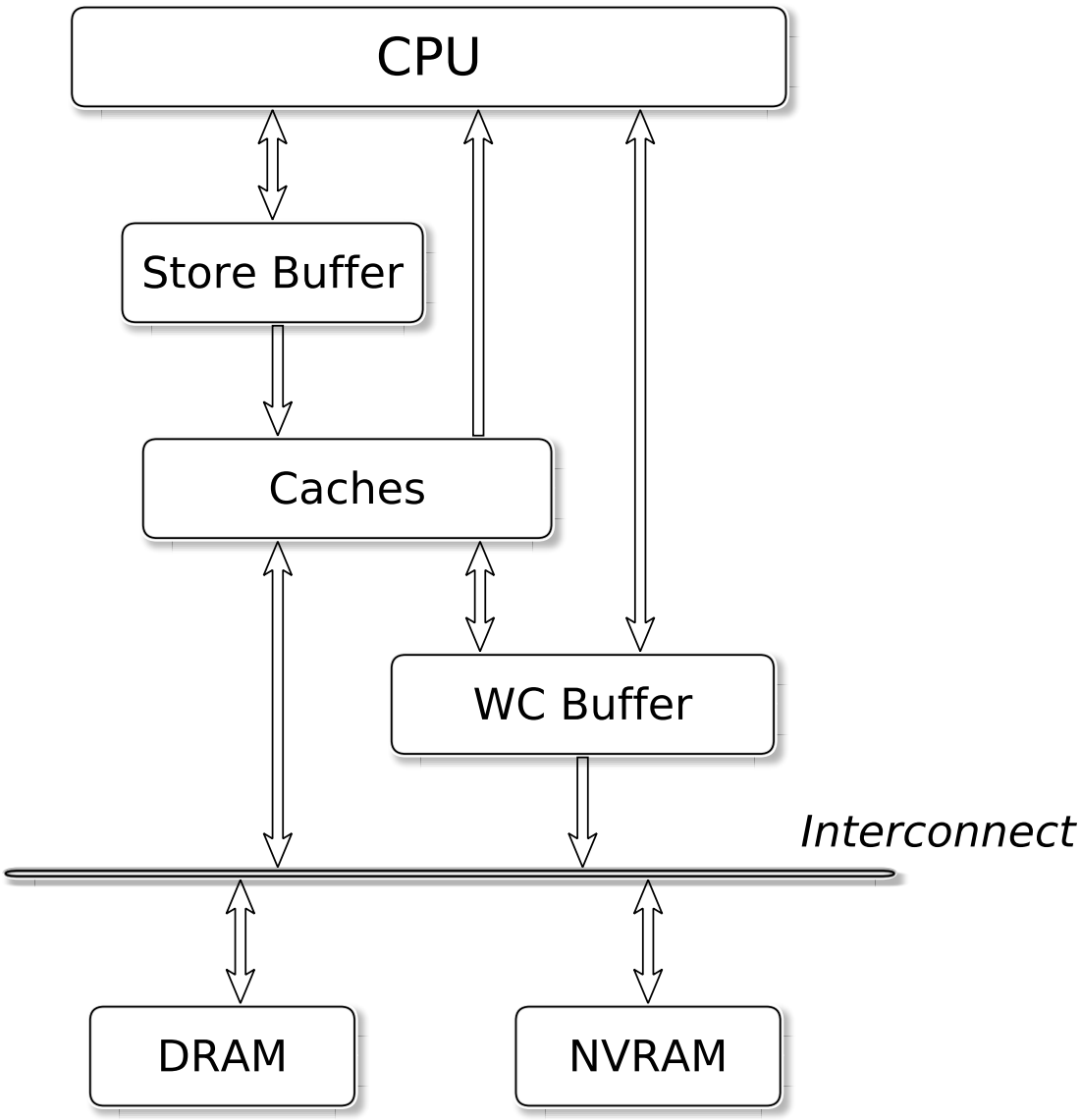


Figure 3.1: A Simplified Architectural Model

3.1 Persistence Designation

An NVRAM application may persist one or more data structures such as a queue, an array or other program variables in NVRAM. Such a data structure is designated as persistent by explicitly allocating it in NVRAM.

This approach is in contrast with earlier pre-NVRAM persistence programming models such as orthogonal persistence [22]. In orthogonal persistence, all objects were allocated the same way, and hence all allocated objects are potentially persistent. The persistence is based on the liveness of the variable and/or its reachability from other persistent objects. This required pointer reachability analysis at different program points incurring performance penalties. Furthermore, transient objects such as network sockets, and user passwords may undesirably become persistent under such a scheme. Such undesirable persistence of data also unnecessarily increases the size of persistent data. Under the persist-reuse model described here, only the user-desired data becomes persistent. However, this puts a burden on a programmer to correctly identify such persistent objects. In the example in figure 3.2, line 1 shows a node of a persistent queue being explicitly allocated.

3.2 Persistent Data Failure-safe Updates

Persistent data in NVRAM is stored in the same format as it is manipulated. This single format makes it possible to manipulate often times a single copy of persistent data in NVRAM directly using CPU load/store instructions. Such modifications need to take place in a failure-safe manner. In an example in figure 3.2, the node allocated in line 1 is being initialized in 2, and finally being attached to persistent queue in NVRAM in lines 4–7. In this example, stores associated with lines 1, 2 and 3 need to be visible in NVRAM before stores in lines 4–7. Additionally, stores from 4–7 need to be visible in NVRAM on-all-or-nothing basis w.r.t failures and power recycle.

A programming library implementing this programming model may provide tools to the programmer to identify a section of code that requires failure safety guarantees. The library may then enforce proper ordering and atomicity of stores based on programmer directives.

Mnemosyne, one such NVRAM programming library that implements this programming model, for instance, offers software transactional memory to programmers as a means to enforce failure atomicity. As the readers will observe in the following chapters, the failure atomicity and store ordering can also be largely inferred automatically for certain class of legacy code.

This approach is also different than the approach taken traditional orthogonal persistence. To uphold the design principle of "persistence independence", orthogonal persistence allows a programmer to write code independent of persistence or potential persistence of the data that the code manipulates [22].

3.3 Restart Code

The central goal of failure safe updates mentioned in the previous section to transition persistent data stored in NVRAM from one consistent state to another. The programmer is responsible for writing restart code that adjusts the programming context after failure or power recycle based on the consistent persistent data in NVRAM. A restart code is never expected to see an inconsistent version of the persistent data. Depending on the implementation strategy, the library implementation of the programming model may use some post-failure recovery logic (e.g persistent logs replay) to restore the state of the persistent data in NVRAM to a consistent state before user-written restart code accesses such data. In the example in figure 3.2, restart code in lines 8–9 re-initializes the head and tail pointers of a queue with values from NVRAM.

3.4 Persist-Reuse Model vs. Checkpoint-Restart

Persist-Reuse Model is much different that system level checkpoint- restart (SLC) and closer to application-level checkpoint-restart (ALC). In SLC, a programmer may be able to define the frequency of checkpoint, but since the checkpointing happens usually at the OS level, the whole state of execution is usually saved as a snapshot [23]. Such a snapshot may contain register states, call stack, program variables and so on. The persist-reuse model is

much more focused on the state of the certain application data rather than the state of the whole execution.

Some similarities exist between ALC [24] and persist-reuse model in the sense that programmer much more control over what program variables and data are important to be saved for later use. However, the emphasis of ALC to recreate the state of computation to last check-pointed state on restart. As a result, ALC may still need to save more information about transient program states, which is not typically done under the persist-reuse model. Persist-reuse model is much data-centric (as opposed to execution-centric) in the sense that persist-reuse model only concerns with advancing the state of persistent data in NVRAM. For example, 10 threads may be adding nodes to a persistent queue under persist-reuse model before a failure occurs. When the application restarts, it initializes from the last consistent state of the queue (i.e. queue with a certain number of nodes added to it) before the failure occurred and may resume with a single thread or 20 threads. Alternatively, unlike the ALC, persist-reuse model does not necessarily attempt to recreate the exact execution scenario before failure. Furthermore, the persist-reuse model allows a single copy of user data to be continually updated and the same copy becoming the basis of restart later. This is also not the case with checkpoint-restart where a snapshot is stored separately in a possibly different format and never manipulated directly.

3.5 System Assumptions

In the face of evolving hardware architecture, much of the contemporary work on the persist-reuse programming model for NVRAM (e.g. Mnemosyne, NV-Heaps, Atlas, Pmem.io) make following general assumptions about the underlying hardware. It is assumed that the access latency of durable data stored in NVRAM is comparable to transient data stored in DRAM. At the lowest level, initial NVRAM devices may make trade-offs resulting in write latencies appreciably longer than DRAMas discussed in chapter 2. But this increased latency may not be user visible. It appears likely that memory controllers will have enough capacitive power backup such that write requests, once accepted by the memory controller, can be

viewed as persistent. A recent decision by Intel to ensure that persistence domain extends to write pending queue in memory controller seems to confirm this case for the future (see chapter 2) [12]. Even if the power and the CPU die, the controller will ensure that accepted requests are written. Hence, the actual write latency to the NVRAM device may not matter for our purposes. Hence, unless otherwise noted, NVRAM is simulated by DRAM using Linux RAM disk [25] throughout this thesis work.

This thesis work, similar to other related work on persist-reuse programming model, also assumes that persistent data stored in DRAM and transient data stored in NVRAM are addressed using the same virtual address space.

A fail-stop model is assumed throughout this thesis and does not address partial failure scenarios. If a component of a system fails, the whole system is expected to fail. Process crash, power outage are all assumed as tolerated failures and once such failure happens, the system halts until it is restarted.

The work in this thesis, along with much of the related work, uses CLFLUSH and MFENCE instructions when estimating persistence programming performance overhead instead of new cache management instructions described in section 2.3. This approach is taken for the following two reasons. First, actual hardware with extended ISA is not widely available. Second, almost all of the related work (e.g. Mnemosyne, Atlas, nvm_malloc) against which we can compare our work use CLFLUSH instructions instead of the new instructions and thus doing so makes it a fair comparison. These performance estimates using CLFLUSH are rather pessimistic, and throughout this thesis, we describe why we expect the overhead estimate to decrease (not increase) and the performance to improve with new and better ISA.

3.6 Summary

The anticipation of NVRAM being widely available has inspired a persist-reuse programming model that is quite different than the traditional approaches to persistence such as orthogonal persistence and checkpoint-restart. Persist-reuse enables single-copy update

and checkpointing as opposed to snapshot based checkpointing. It also streamlines the persistent data needed to restart a program after a failure or power cycle. Unlike in the implementation of orthogonal persistence, it does not require expensive reachability-based analysis to determine what variables and states need to be persisted at various program points.

```
/* allocate */
1: Node* t_tmp = nvm_alloc(sizeof(Node));

/* initialize */
2: t_tmp->val = 10;
3: t_tmp->next = NULL;

/* publish */
4: BEGIN FAILURE ATOMIC SECTION
5: p_queue->tail->next = t_tmp;
6: p_queue->tail = t_tmp;
7: END FAILURE ATOMIC SECTION

/* user written restart code */
8: p_queue->head = NVM_root(0);
9: p_queue->tail = NVM_root(1);
```

Figure 3.2: A sample pseudocode showing persist-reuse model.

Chapter 4

Combining Lock-based Approach with Copy-on-Write for NVRAM Persistency

Persistent data stored in emerging NVRAM, can be accessed directly through CPU load and store instructions. This means that the persistent data is being stored in the same format as it is being manipulated, thus requiring only a single copy of data to be maintained, and avoiding expensive and cumbersome conversion.

NVRAM opens up an opportunity to have in-memory object persistence so that program states that outlive the creating process can be preserved, shared, and reused [16, 17, 19]. Using this persist-and-reuse model, a quick restart of an application from an intermediate state appears a reality. While starting, an application looks for existing data that it can reuse. If present, the application adjusts its context and instead of computing from scratch, merely reuses the existing data for the rest of its computation.

Persistent data must be updated with great care so as to be reusable. Otherwise, updates may become visible to NVRAM only partially, or not in the intended order. This may happen as a result of failures in the middle of critical sections, volatile CPU caches, or out-of-order cache evictions. As a result, an interruption such as a power failure may introduce inconsistencies (e.g. dangling pointers) to persistent data in NVRAM. Hence, some set of critical updates to persistent data must be applied on an all-or-nothing basis, and in the correct order with respect to other updates, to guarantee the consistency of in-memory persistent data.

4.1 Log-based Transactional Approach vs. Copy-on-Write

Prior work [16, 17, 19, 21] in this area has focused on developing non-volatile memory programming libraries (NVMPLs) which enable programmers to specify a failure-atomic section (FASE) of updates to persistent data using familiar programming languages such as C/C++. One such NVMPL Atlas automatically infers such a FASE from legacy lock-based code. All of NVMPLs mentioned above, including Atlas, use persistent transactional logs to enforce failure atomicity. The major drawback of this approach is the high cost of writing and flushing logs synchronously to NVRAM. Furthermore, automatic or manual code changes are necessary to generate such logs. On the other hand, copy-on-write approach to updating data is inherently failure atomic without additional code changes. Furthermore, it does not require expensive logging. The advantage of the log-based approach is the ability to easily add atomicity to an arbitrary set of writes at an arbitrary granularity. This chapter of the thesis work demonstrates how the log-based transactional approach can be combined with copy-on-write to achieve the best of both worlds.

4.2 Lack of NVRAM Applications

previous work has suffered from the lack of benchmarks and real-world data structures that are cognizant of NVRAM. This chapter also addresses this problem by developing an NVRAM version of the commercially used legacy disk-based key-value store. This chapter describes MDB-NVM, which is an NVRAM-based key-value store implemented using C programming language. This key-value store preserves the API and the functionality of MDB [26] such that it could be used instead of MDB-NVM out of the box. MDB is a disk-based key-value store that is designed to replace BerkeleyDB [27] in a commercial and non-commercial setting as an improved backend for OpenLDAP. MDB-NVM is implemented partially using Atlas NVMPL [19] and copy-on-write mechanism for consistency. MDB-NVM played a crucial role in the development and evaluation of Atlas.

4.3 Contributions

This work makes the following contributions:

1. demonstrated how copy-on-write can be combined with a lock-based transactional approach to improve programmability and lower persistence overhead.
2. developed a real-world data structure that is designed to utilize `NVRAM`,
3. a measurement of the cost of persisting a real-world data structures under workloads with varying characteristics.

4.4 Atlas Programming Library

Altlas Library adds durability semantics to the lock-based concurrent programs, typically with very little effort from a programmer [19]. In programs written using locks, shared data structures are modified only within a critical section, marked by lock acquire and release, to avoid data races and guarantee their consistency. Hence, Atlas assumes that these data structures only become inconsistent within critical sections and are consistent everywhere else. This assumption fails in cases of single-threaded programs, data structures that are isolated by design, or when a programmer writes non-blocking multithreaded code using primitives such as compare-and-swap. In such cases, a programmer can specify the section of code that requires atomicity w.r.t failure using programming construct `nvm_begin_durable` and a matching `nvm_end_durable`. Note that this construct does not guarantee thread-safety.

4.4.1 Failure Atomic Section and Consistency Guarantees

Locks can be acquired and released in a nested pattern. Data cannot be considered consistent when any of the locks are held. Therefore, the whole outermost critical section in a given thread has to be executed atomically w.r.t failure. This outermost critical section executed by a thread is called the Failure Atomic SEction (FASE). FASEs between two threads may establish a happen before relationship during the runtime through lock acquire and

release. Note that the start and the end of the outermost critical section may not be marked by the acquire and the release of the same lock, due to cases such as hand-over-hand locking.

In Atlas, an execution of a FASE becomes durable (i.e. visible in `NVRAM`even after a restart) only if all updates to persistent memory made within that phase are durable. This guarantees an all-or-nothing behavior of a FASE w.r.t failure. Furthermore, given FASEs f_1 and f_2 such that f_1 is sequenced before f_2 as established by lock acquire and release, the f_2 is durable only if f_1 is so. Lastly, Atlas ensures that if a FASE is made durable, all updates to persistent memory that are sequenced before the FASE as established by runtime happen-before relationships are also durable. These requirements enforced by Atlas library maintains the consistency of persistent data structure stored in NVRAM across failures and restarts. After failure or restart, the persistent data is in the state as though each thread stopped execution at some program point where no locks were being held.

4.4.2 Compiler Instrumentation and Runtime

Hidden from a programmer, Atlas uses a combination of write-ahead logs with undo information, cache line flushes and memory fences to ensure correct visibility ordering of updates to persistent memory and to enforce failure atomicity. A compilation pass instruments synchronization operations and store operations that appear directed to persistent memory. During the runtime, the instrumented stores call the Atlas runtime library which may create persistent logs, execute memory visibility barriers comprising of memory fences and cache line flushes. Atlas keeps the log structure and the log entries themselves consistent across failures. These log entries contain sufficient information to recreate a consistent state after a failure or a restart.

Atlas makes log entries with undo information visible in persistent memory before stores to the corresponding locations become visible. This enables Atlas recovery to replay the log entries to undo the effect of any stores that are not part of the last consistent state

achieved. Furthermore, each log entry becomes visible in NVRAM atomically and in program order.

Consistent State Computation A helper, separate from the worker thread computes a consistent state from the persistent log, advances the persistent data structure's consistent state, and removes the log entries associated with the latest consistent state achieved. To advance the consistent state, a helper thread creates a graph where each node corresponds to a set of log entries associated with each FASE executed by each worker thread. The ordered edges representing the happens-before relationship may join one or more nodes in the graph. First Atlas marks all nodes in this graph representing FASEs as durable. Then it marks FASEs with unmatched acquire(s) and release(s) in the FASE as incomplete and hence non-durable. Furthermore, all nodes reachable from incomplete nodes are also marked as non-durable. The effects of non-durable FASEs are not to be seen by user code after a failure or a restart and are not part of the newly computed consistent state. Therefore, undo information related to incomplete nodes are preserved to undo the memory effect in the case of a crash or restart. The rest of the log entries associated with complete nodes unreachable from any incomplete nodes are purged.

4.4.3 Failure, Restart, and Recovery

Under the persist-reuse model, Atlas follows, a programmer adds code that checks for the prior version of the data available upon restart (both normal and post-failure). If the prior version of the data is available, this "restart code", adjusts the programming context so as to resume the computation from this data.

In the case of a failure (e.g. sudden power loss), persistent data may not be in a consistent state. This inconsistent state is never visible to the application. Upon restart, Atlas first computes a consistent state from the persistent logs and prunes the log as helper thread may have lagged behind before the failure. Next, it replays the outstanding logs (after pruning) associated with incomplete FASEs and un-does any inconsistent updates

restoring the data structure to a consistent state. Once the persistent data is restored to a consistent state, user applications may resume.

4.5 MDB-NVM: NVRAM Key-value Store

MDB-NVM stores data in persistent memory and ensures that it is always consistent. It is thread-safe and uses a copy-on-write mechanism to avoid conflicts between read and write transactions. Therefore, multiple read transactions can proceed alongside the write transaction. MDB-NVM only supports one active write transaction at a time.

4.5.1 Programming Interface

Figure 4.1 lists the common method calls to interact with MDB-NVM. These interfaces are similar to ones provided by MDB.

Initialization, Setup, and Shutdown API: `mdb_nvm_env_create` (line 1) creates an environment object and stores in the location pointed by its argument. A `MDB_env` object holds contains several fields to hold information about the MDB database such as the size of the `NVRAM` region to be allocated for this database etc. An `NVRAM` persistent region is a named container, similar to a mmap-ed file, that stores persistent data [19].

`mdb_nvm_env_open` (line 2) takes the `MDB_env` object, created by the call to line 1, as an argument, initializes MDB-NVM and populates the argument object with current MDB-NVM settings. The three arguments `path`, `flags`, and `mode` are not necessary and are only preserved for compatibility with MDB. `mdb_nvm_txn_begin` (line 3) creates a transaction to be used with the given environment. The `parent` argument is for a nested transaction which is not currently supported in MDB-NVM, and as a result `NULL` can be passed as the argument. The `flag` argument indicates whether the transaction is read-only or read-write.

`mdb_nvm_dbi_open` (line 4) opens a database referred by the name in the given environment. If the `name` argument is null, it opens the default database. MDB-NVM currently only supports a single default database, and hence the `name` argument is ignored. MDB-NVM

```
1: int mdb_nvm_env_create (MDB_env ** env)
2: int mdb_nvm_env_open (MDB_env * env, const char *path,
     unsigned int flags, mdb_mode_t  mode)
3: int mdb_nvm_txn_begin (MDB_env *env, MDB_txn *parent,
     unsigned int flags, MDB_txn **txn)
4: int mdb_nvm_dbi_open (MDB_txn *txn, const char *name,
     unsigned int flags, MDB_dbi *dbi)
5: int  mdb_nvm_txn_commit (MDB_txn *txn)
6: void mdb_nvm_txn_abort (MDB_txn *txn)
7: void mdb_nvm_dbi_close (MDB_env *env, MDB_dbi dbi)
8: void mdb_nvm_env_close(MDB_env *env)
9: int  mdb_nvm_put (MDB_txn *txn, MDB_dbi dbi,
     MDB_val *key, MDB_val *data, unsigned int flags)
10:int mdb_nvm_get(MDB_txn *txn, MDB_dbi dbi,
     MDB_val *key, MDB_val *data)
11:int  mdb_nvm_cursor_open (MDB_txn *txn, MDB_dbi dbi,
     MDB_cursor **cursor)
12:int mdb_nvm_cursor_get(MDB_cursor * cursor,MDB_val *key,
     MDB_val *data, MDB_cursor_op op)
13:void mdb_nvm_cursor_close (MDB_cursor *cursor)
14:int mdb_nvm_del (MDB_txn *txn, MDB_dbi dbi,
     MDB_val *key, MDB_val *data)
...
```

Figure 4.1: A non-exhaustive list of MDB-NVM API. A complete list can be found in mdb.h

returns the database handler object by storing it in at the address provided as the 4^{th} argument `dbi`.

`mdb_nvm_txn_commit` (line 5) commits the given transaction whereas `mdb_nvm_txn_abort` (line 6) aborts the transaction. A read-only transaction can be aborted with no effect to the database as such a transaction does not modify the database.

`mdb_nvm_dbi_close` and `nvm_env_close` closes the database handle and destroys the environment object created respectively. Both of these calls should only be used at the end to cleanly shutdown the database. They should also only be called once by a single thread.

Read and Write API: `mdb_nvm_put` (line 9) stores a <key, data> pair into the database indicated by the `MDB_dbi` database handler object (returned by line 4). Note that there is only one default database supported in MDB-NVM at the moment. The key and data both have to be of type `MDB_val`, which has the following structure:

```
typedef struct MDB_val {
    /**< size of the data item */
    size_t mv_size;
    /**< pointer to the data item */
    void *mv_data;
} MDB_val;
```

`mdb_nvm_get` returns the value and the size of the data associated with the *key* argument if it exists in the database. It returns the value (of type `MDB_val`) by assigning its address to the field `mv_data` in the 4^{th} argument `data`. The address of the data belongs to MDB-NVM and therefore, the user application should make a copy instead of directly modifying it.

`mdb_nvm_cursor_open` (line 11) creates a cursor capable of traversing the entire database. It returns the cursor handle by storing the address of the handle in the `cursor` (3^{rd}) argument. `mdb_nvm_cursor_get` (line 12) returns the <key, value> pair at the current

cursor position. The 4^{th} argument `op` instructs MDB-NVM on the next position of the cursor. `mdb_nvm_cursor_close` (line 13) destroys the cursor handle.

Finally, `mdb_nvm_del` (line 14) deletes the <key,value> pair matching the given key from the database if it exists. This operation requires the transaction to be opened in the read-write mode just like the put operation. Figure 4.2

4.5.2 Internal Structure

MDB-NVM is thread-safe, and failure-safe. MDB-NVM stores user data in persistent memory maintain its consistency. It maintains only the essential metadata about the database in persistent memory to minimize the cost of persistence. Rest of the metadata is recreated from the persistent metadata and maintained in transient memory. We discuss and distinguish below transient vs. persistent metadata:

Persistent B-tree

MDB-NVM stores <key, value> pairs as a B-tree [28]. Each node in the B-tree is system page size, typically 4096 bytes. This B-tree is stored in an `NVRAM` persistent region (see section 4.5.1), and updated using copy-on-write mechanism. When a new environment is opened, MDB-NVM looks for the existing NVRAM region containing MDB-NVM database. If it does not find the already initialized persistent region, it creates a new one. Figure 4.3 shows the code snippets from the MDB-NVM initialization routine.

Persistent Metapages

At the beginning of the persistent region, MDB-NVM allocates two identical metapages. At any given time, only one metapage is active, and all transactions start at the active metapage. A write transaction toggles the active metapage between the two. An active metapage stores a pointer to the current version of the database, a current list of unallocated (free) pages belonging to the persistent region, the id of the last write transaction that toggled the metapage, a bump pointer keeping track of the last page (highest virtual

```
#include "mdb.h"
....
void update_mdb_nvm(){
  MDB_env *env; MDB_dbi dbi;
  MDB_val key, data;
  MDB_txn *txn;
  int key;
  char sval[32];
  int rc;

  rc |= mdb_nvm_env_create(&env);
  //sets the size of persistent region to be allocated
  rc |= mdb_nvm_env_set_mapsize(env, 10485760);
  rc |= mdb_nvm_env_open(env, NULL, NULL, NULL);
  rc |= mdb_nvm_txn_begin(env, NULL, 0, &txn);
  rc |= mdb_nvm_open(txn, NULL, 0, &dbi);

  key = 12;
  key.mv_size = sizeof(int);
  key.mv_data = &key;

  sprintf(sval, "foo bar soo boo");
  data.mv_size = sizeof(sval);
  data.mv_data = sval;

  rc |= mdb_nvm_put(txn, dbi, &key, &data, MDB_NOOVERWRITE);
  rc |= mdb_nvm_txn_commit(txn);
  ASSERT(!rc);
}
...
```

Figure 4.2: A simple prpgram that inserts a key-value pair into the MDB-NVM

```
#include "mdb.h"
....
int mdb_nvm_env_open(MDB_env *env, const char *path,
 unsigned int flags, mode_t mode)
{
   ....
   NVM_Initialize();
   //initialize a persistent region
   env->me_rgnid = NVM_FindOrCreateRegion("mdb", O_RDWR, NULL);
   ....
   if (newenv) {
      //allocate large chunk of memory within the region
      env->me_map = nvm_alloc(env->me_mapsize, env->me_rgnid);
      ...
      //initialize persistent metadata
      mdb_nvm_env_init_meta(env, &meta);
   }

   //initialize here env object from the existing metadata
   ...
}
...
```

Figure 4.3: A Code Snippet showing the creation of a MDB-NVM persistent region

address) in the persistent region currently in use by MDB-NVM. MDB-NVM attempts to reuse earlier allocated free pages before incrementing this bump pointer.

Transient Reader Table

MDB-NVM maintains a transient table of all threads that have opened a read-only transaction to the database during any given execution cycle. Each unique thread has a slot in this transient table. For each thread that opens the read-only transaction, MDB-NVM stores the current version of the database that the thread is using. The current version is indicated by the transaction id of the last write transaction that created the active metapage pointing to the most current database (which the read-only transaction will be using).

Each thread acquires a "read" lock to create a slot in this table when accessing database for the first time in read-only mode. Once the slot is created, subsequent read-only access does not require any lock acquisition. This table is not relevant from execution cycle to another and hence is not persisted. The transaction id of the last write transaction is already recorded in the metapage. By comparing the transaction id contained in two metapages, the restart code can determine which metapage contains the latest version of the database. Note, that there is one slot per thread in this table. Therefore, MDB-NVM cannot support more than one outstanding transaction per thread.

4.5.3 Persistent Memory Management

All persistent memory allocation in MDB-NVM happens at page-size. Recall that MDB-NVM modifies the B-tree that stores <key, value> pairs using the copy-on-write mechanism. Each time a node has to be modified, it allocates a page from the list of free pages maintained in the active metapage, copy the content from the old page to the newly allocated page, and apply the changes to the newly allocated page. The old page is added to the list of free pages and eventually reused.

The list of free pages is indexed by the transaction id of the write transaction that last wrote to these free pages. The list is sorted by the transaction id - the most recently written

pages are added to the end of the list. Each time a page has to be allocated, MDB-NVM starts at the beginning of the map, i.e entry with the lowest transaction id (the earliest written page). Each page has a transaction id, indicating the last transaction that wrote to the page. It tallies this transaction id with the lowest of all the transaction ids for all readers found in the transient reader table. If the lowest transaction id found in the reader table is higher than the free page transaction id, the page is safe to be reused as no reader is reading from this page or will read from this page in the future. Thus, the page is allocated for a new node. If there are no eligible pages in the free list, a new free page is obtained by incrementing the bump pointer stored in the active metapage.

Miscellaneous Transient Structures

All other data structures such as MDB-NVM environment handler object (`MDB_env`), database handler object (`MDB_dbi`), transaction handler (`MDB_txn`) or cursor handler (`MDB_cursor`) are allocated in transient memory (see section 4.5.1 for interface description). These structures either merely replicate the information stored in persistent metapages. All transient de-/allocations are performed using standard C malloc/free library calls.

4.5.4 Read-Only Transactions

A read-only transaction cannot modify the MDB-NVM database. Therefore, the operations are limited to get, and database traversal. Each thread initiating a read-only transaction for the first time registers itself in the reader table as described section 4.5.2. In each subsequent read-only access, the thread first copies the metadata fields such as the root of the current version of the database, the transaction id of the write transaction that created the current database version from the active metapage into the transient transaction handler object associated with the current read-only transaction. It also updates its entry in the reader table with transaction id from the metapage. Next, all gets and database traversal requests proceed from the database root stored in the transaction handler. The handler can be discarded at the end by aborting the transaction.

4.5.5 Read-Write Transactions

MDB-NVM only allows one read-write transaction to be active at a time. To prevent another thread from starting the read-write transaction, MDB-NVM requires a thread to acquire a write lock, set the flag to indicate a read-write transaction in progress, and release the lock. All other subsequent attempts to start read-write transaction wait conditionally on this flag to be unset. The flag is unset once the transaction is committed. A thread that sets the flag successfully copies the metadata in active metapage into the transaction handler. This includes the root (pointer) to the current version of the database and the list of free pages. Each time, it has to modify a node which is stamped with the current transaction id, the thread allocates a new page (as described in section 4.5.2), stamps the new page with the new incremented transaction id, copies the existing page (node) to the new page, modifies the new page. Each old version of the modified node is added to the new list of free pages as deallocated pages. Recall that deallocated pages are only used if they are safe to do so (i.e. no potential readers are active, (see 4.5.2).

To commit the transaction, the following steps occurs atomically with respect to failure:

1. A new root to B-tree is formed as a result of updating the database using copy-on-write semantics. This new currently stored in the transaction handler is copied to the non-active metapage.
2. The pointer to a new list of free pages is copied to the non-active metapage.
3. The new incremented transaction id to represent the ongoing read-write transaction is written to the non-active metapage.
4. Other miscellaneous persistent data is copied from the active metapage and from the transaction handler to the non-active metapage.

The above set of failure atomic steps ensures that transactions are published in persistent memory on an all-or-nothing basis. Once this completes, the updated metapage is toggled to active. It also ensures that persistent memory pages are not leaked permanently. If a failure were to occur before the transaction committed, the existing version of the

database continues to serve as the most recent version. Also, the existing free pages list is unaffected. Thus the persistent state of the database remains intact.

```
#include "mdb.h"
...
mdb_txn_commit(MDB_txn *txn)
{
  ...
  NVM_BEGIN_DURABLE();

  //free page list
  meta->mm_dbs[0] = txn->mt_dbs[0];

  //database root
  meta->mm_dbs[1] = txn->mt_dbs[1];

  //bump pointer, max page within PR
  meta->mm_last_pg = txn->mt_next_pgno - 1;

  //new transaction id stamp
  meta->mm_txnid = txn->mt_txnid;

  NVM_END_DURABLE();
  ...
}
...
```

Figure 4.4: A Code Snippet showing the creation of a MDB-NVM persistent region

Failure Consistency: The above failure atomicity is designated using Atlas programming construct `nvm_begin_durable` and `nvm_end_durable`. Figure 4.4 shows this failure code section in MDB-NVM. Note that this failure atomic section is outside the critical section. The write locks were released immediately after setting the flag. The write lock could have been held for the entire duration of the write transaction. This would cause Atlas to log each persistent write during the transaction, incurring high failure consistency

cost. To avoid the cost of logging and publishing the log, MDB-NVM uses copy on write mechanism to publish transactional updates on an all-or-nothing basis. This minimizes Atlas logging to the small section of the code above.

Although Atlas does not perform logging outside the failure atomic section described above, it tracks all persistent writes within the read-write transaction leading up to the failure atomic section (FASE) defined by `nvm_begin_durable` and `nvm_end_durable`. Recall that Atlas guarantees the visibility of all persistent writes sequenced before a FASE before the effect of executing a FASE is visible in `NVRAM`. Therefore, all updates to nodes become visible when the effects of the FASE are published in `NVRAM`.

Once the active metapage is toggled, the thread acquires the write lock again, unsets the flag indicating active write and releases the write lock. This completes the transaction commit. All new read-only and read-write transactions proceed from the root in the new metapage. Thus we achieve failure consistency at a much lower cost using copy-on-write for updating non-metadata pages.

4.6 MDB vs. MDB-NVM

MDB-NVM preserves the interface of MDB, but differs drastically in its internal structure and how a read-write transaction is committed. First, MDB performs all the modifications during a read-write transaction to transient nodes and uses file I/O to persist these nodes. Hence, they do not have to address the challenges of persistent memory leaks, and in-place updates of persistent memory regions. This simplifies the memory management aspect of MDB. Furthermore, MDB read-write transactions are oblivious of failure consistency performance implications. For instance, a writing thread holds a lock for the entire duration of the write transaction.

4.7 Methodology

Since real `NVRAM` devices are not yet available17, DRAM was used for simulating `NVRAM`. Linux tmpfs [25] was used for “persisting” data and logs. A file in tmpfs is backed entirely

by DRAM (instead of disks) and is memory-mapped into the address space of a process. Although data in tmpfs does not persist past a system shutdown, it provides a directly mapped, byte-addressable persistent memory across process shutdowns. We successfully performed crash-recovery testing of these programs but a more extensive testbed is required for full correctness testing. Unless otherwise stated, experiments were performed on a Red Hat Linux machine with 4 Intel quad-core Xeon E7330 processors running at 2.4GHz. Results are averages over 6 runs. We compiled the MDB-NVM library using Atlas compiler. The benchmarks do not require Atlas instrumentation MDB-NVM transactions provide failure consistency guarantees, hence they were compiled using regular GNU C compiler.

4.8 Benchmarks

We used the following benchmark to evaluate MDB-NVM implemented using Atlas:

mtest: The workload, Mtest, belongs to the MDB test suite. It first inserts 3000 key/value pairs within a single transaction, where each key is a 4-byte integer and each value is a 32-byte string. Next, it iterates through all the key/values inserted. Then, it deletes some entries where each deletion is a separate transaction, followed by two iterations - forward and reverse - of the entries remaining.

Fillseqsync: This workload is taken from the Google levelDB benchmark suite [29]. It inserts 1 million pairs of 16-byte key and 100-byte value in a sequential order of keys. It perform insertions in multiple batches of key/value pairs, each batch being in its own transaction.

Fillrandomsync: This workload is also taken from the Google levelDB benchmark suite [29]. It inserts 1 million pairs in random key order. Like the previous benchmark, it also performs inserts in multiple batches, each batch being its own transation.

mt-bench: Working in parallel, a writer thread inserts 3000 key/value pairs within a single transaction, another writer thread performs some deletions, each in its own transaction, and each of 5 reader threads scans the existing entries in forward and reverse orders.

4.9 Results

Two sets of results are presented. The first set of results compares the performance of Atlas-enabled MDB-NVM against the disk-based MDB and measures the gain in performance from using `NVRAM`-enabled applications. The second set of results measures the cost of `NVRAM` persistence and identifies quantify some of the sources of overhead.

Disk-based performance Comparison: In figure 4.5, column 2 shows the total time taken by the Atlas-enabled in-memory persistent version of each workload. Column 3 is a measure of the speedup obtained with Atlas (between 3x and 31x) over a disk-based version with disk-subsystem-write-caching enabled. The evaluation machine has a RAID system with a 512-MB battery-backed DRAM write cache, considered reliable by the RAID controller. Column 4 shows the speedup obtained when there is no disk-subsystem write-caching. The numbers in column 4 are obtained by re-running the workloads on an HP Z600 workstation containing 2 quad-core Intel E5620 Xeon processors (2.4GHz) running Red Hat Linux. These results demonstrate that a significant performance improvement can be obtained by using `NVRAM`based in-place persistence over the disk-based persistence.

Cost of Persistence: To measure the source and the cost of persistence, the benchmarks are run in different Atlas modes as described below:

- Transient (TR): This is the base mode with transient data structures alone.
- Atlas (AT): This is the default mode in Atlas and uses all the optimizations [19].
- No-helper (NH): This mode is the same as AT, except that the helper thread is turned off, which means no log pruning.

- Log-only (LO): This mode is the same as AT, except that all visibility barriers are removed.

Column 2 and 3 in Figure 4.6 shows the problem size and the single threaded runtime for the transient version of MDB-NVM respectively for the benchmarks listed in column 1. Columns 4, 5 and 6 present the ratios of sequential transient runtime (column 3) for each benchmark to different Atlas modes described above. The transient version of MDB-NVM was obtained by compiling MDB-NVM with ordinary GNU C compiler and using a mmap-ed file instead of persistent region. Ratios in column 4 show that the persistence comes at a cost, while ratios in column 5 show that the Atlas helper thread does not add much performance overhead. Finally, ratios in column 6 confirm that cache line flush instructions and memory fences are significant sources of persistence overhead.

The cost of persistence below is estimated using CLFLUSH instruction instead of new instructions for cache management such as CLWB (see section 2.3). This is a pessimistic estimate and we expect the performance of MDB-NVM to improve further with the instruction for the following reasons. Each combination of CLFLUSH and MFENCE used in MDB-NVM is replaceable with CLWB and SFENCE combination, which is expected to have better performance. A further optimization is possible as only a single SFENCE is required at the end of the copy-on-write phase for all the CLWB issued during the copy-on-write phase because those updates only need to be guaranteed to be visible by the time the new metadata page updates are visible. The programming techniques described in this chapter remain relevant.

4.10 Related Work

There are many key-value stores available such as TokyoDB [30], BerkleyDB [27] which predate `NVRAM`. These key-value stores are not designed with `NVRAM` as a focus. In general, there is a lack of real-world data structures designed to take advantage of `NVRAM`. A key-value store targeting `NVRAM` was published around the same time this work was carried out [31]. Venkataraman et. al. presented distributed key-value store targeting `NVRAM` [32].

Their work predates this work. The work described here differs from their work as MDB-NVM lends itself to be used with a wider range of NVRAMprogramming libraries supporting various failure consistency semantics.

Various programming libraries have been developed to aid the programmer in developing applications that take advantage of persistence provided by NVRAM. Besides Atlas, Mnemosyne [16], NV-Heap [17], Pmemio [21] are all examples of programming libraries. Mnemosyne adds persistence semantics to transaction-based programs and uses write-ahead persistent redo logs to enforce failure consistency. For MDB-NVM, one can expect the persistence overhead to be similar to Atlas as MDB-NVM requires logging in a limited area of code. However, implementing MDB-NVM is possibly easier with Atlas compared to Mnemosyne, as the persistent writes outside the FASE are automatically tracked and made visible by Atlas before making the subsequent FASE visible. In Mnemosyne, a programmer has to explicitly request for persistent writes outside the transaction to be published.

4.11 Summary

Through the designs of common data structures such as the key-value store, this work showed how to design a persistent data structure to take advantage of NVRAM while minimizing the cost of persistence. While this work showed developing a persistent algorithm using one such programming library Atlas, the general technique such as using the combination of copy-on-write and logging-enabled failure atomic section is generally applicable for other programming libraries such as Mnemosyne [16].

Workloads	Runtimes w/ Atlas (us)	Runtime Ratios	
		$\frac{Disk-c}{Atlas}$	$\frac{Disk-unc}{Atlas}$
`Mtest`	61345	25	287
`Fillseqsync`	39845	3.7	460
`Fillrandomsync`	40014	3.7	518
`mt-bench`	80072	31	168

Figure 4.5: In-memory vs disk-based persistence: Disk-c: disk caching enabled, Disk-unc: disk caching disabled

Benchmarks	Problem Size	TR sequential runtime (us)	AT	NH	LO
`mtest`	3000	122695.0	5.0	4.9	3.0
`Fillseqsync`	1000000	6130	6.5	6.5	3.6
`Fillrandomsync`	1000000	7020	5.7	5.8	3.3
`mt-bench`	3000	22145	3.6	3.5	2.2

Figure 4.6: Sources of persistence overhead and the cost of persistence.

Chapter 5

Recoverable Allocation of Non-volatile Memory

Prior work [16, 17, 19] in this area has focused on developing non-volatile memory programming libraries (`NVMPLs`) which enable programmers to specify a failure-atomic section (FASE) [19] of updates to persistent data using familiar programming languages such as C/C++.

This work focuses on efficient memory management of persistent memory, an issue that was largely sidestepped by previous work. In addition to ensuring that online allocations and deallocations are efficient, two primary challenges are addressed here:

1. Allocator metadata consistency: The internal invariants have to be maintained in `NVRAM` in a fail-safe manner across restarts and tolerated failures to continue to allocate memory correctly.
2. Failure-induced persistent memory leaks: If persistent memory is allocated but a failure occurs before an application handle can be assigned to that memory, that memory location is essentially leaked since it is unreachable on program restart.

This paper describes Makalu[1], a persistent memory allocator that uses offline garbage collection (GC) to provide leak-freedom. Persistent data has two distinct phases with respect to a mutator application: (1) ***online***, when the mutator is active and (2) ***offline***, when the mutator is absent, but the heap is still accessible. After a failure, once the `NVMPL` restores persistent data to a consistent state, Makalu performs the parallel mark phase of mark-and-sweep GC offline followed by incremental sweeps online to avoid both

[1] Makalu is the fifth highest mountain in the world. Its four-sided pyramid shape represents the challenges this work simultaneously addresses: memory management, garbage collection, persistent vs transient data, and failure-resilience.

failure-induced and programmer-induced leaks. This approach enables us to interoperate with other NVMPLs, while supporting unrestricted online usage of standard de-/allocation interfaces within a C/C++ program.

Makalu has built-in failure-resilience and recovery mechanisms to guarantee consistency of its metadata without depending on any NVMPL. Note that Makalu does not determine what data needs to be persisted or the structure of the persistent data. It only determines the structure of the heap in NVRAM.

Makalu's approach achieves interoperability with existing code while providing a safety-net against all sources of memory leaks. By having the offline GC restore certain metadata, it avoids the need to ensure full consistency of metadata at every allocation, normally an expensive online operation. Makalu's raw online allocation speed and multi-threaded scalability are comparable to well known transient allocators, a tremendous improvement compared to state-of-the-art persistent allocators. Makalu has been integrated with two NVMPLs, Atlas and Mnemosyne, and it saw up to 2x speed improvement in some scientific applications compared to NVMPLs' default allocators. This observation leads one to believe that offline garbage collection may become an integral part of future NVRAM memory management schemes.

5.1 Contributions

This work makes the following contribution:

1. A careful analysis of the requirements and opportunities for NVRAM memory allocation.
2. A set of techniques that result in an NVRAM memory allocator that is often competitive with widely used DRAM allocators, and provides failure consistency at orders of magnitude lower cost than prior approaches.
3. A demonstration that recovery time garbage collection not only simplifies the programming model, but also, by allowing reconstruction of metadata at recovery time,

greatly speeds up NVRAM memory allocation. The offline GC is performed at a small fraction of the cost of general GC.

5.2 Background

5.2.1 Architectural Assumptions

This work makes similar assumptions about the underlying architecture as found in prior NVMPL work [16, 17, 19]. NVRAM is assumed to retain data through tolerated failures, such as power failures. For convenience, NVRAM latency is assumed to be comparable to DRAM, which may or may not be present. At the lowest level, initial NVRAM devices may make trade-offs resulting in write latencies appreciably longer than DRAM. But this increased latency may not be user-visible. It appears likely that memory controllers will have enough capacitive power backup such that write requests, once accepted by the memory controller, can be viewed as persistent. Even if the power and the CPU die, the controller will ensure that accepted requests are written. Hence, the actual write latency to the NVRAM device may not matter for the purposes of this work.

NVRAM endurance is expected to be orders of magnitude better than that of SSDs [2, 7] but solutions have been proposed [33] if wear-out is a concern. This work does not address NVRAM endurance issues.

Several levels of volatile caches and potentially other volatile buffers exist between the CPU and NVRAM. This work assumes a tolerated failure (such as a process crash, an OS kernel panic, and a power failure) to be fail-stop. In such an event, the data that are already in NVRAM survive, but other data in volatile hardware structures do not.

Instructions are available to selectively evict or flush a cache line from the volatile caches into NVRAM (such as Intel x86 CLFLUSH [20] and CLWB [11]). Once evicted, these cache lines will eventually reach memory. These instructions are expensive [16, 19] and should be sparingly used.

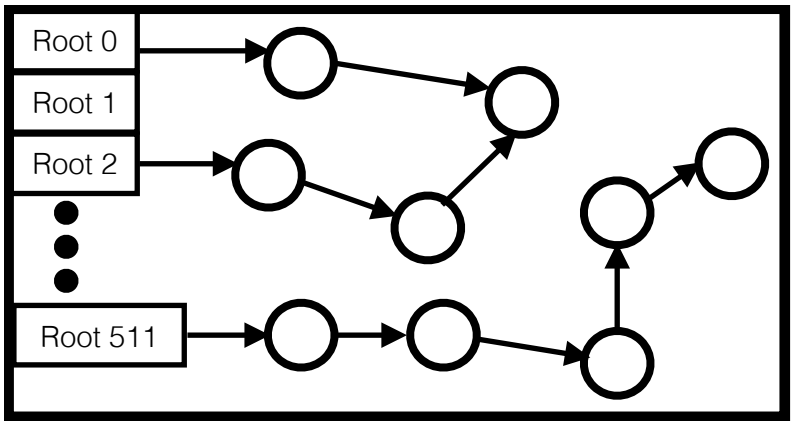


Figure 5.1: A snapshot of persistent heap with heap objects reachable from top level region roots within a NVRAM region.

5.2.2 Programming Assumptions

NVRAM is mapped directly into the process address space without any buffering and is accessible using CPU loads and stores. Persistent data is stored in named containers (a concept similar to mmapped files) called NVRAM regions. A persistent heap exists within the realm of an NVRAM region. When a heap is in a consistent state offline, all useful data that should be accessed upon restart must be reachable from a set of known persistent roots, otherwise, data can be considered garbage and reclaimed. In Makalu, there are 512 top-level region roots stored in known locations within the NVRAM region. These assumptions are similar to ones made by most current NVMPLs [16, 17, 19, 21]. Figure 5.1 shows a sample persistent heap along with top level roots. This work assumes that an NVRAM region is always mapped at the same base address so that existing persistent-to-persistent pointers embedded in it remain valid, while persistent-to-transient pointers are ignored by the offline GC. With more precise pointer identification in the future, the offline GC phase can automatically clear such pointers for the programmer.

This work assumes that Makalu will be used in conjunction with other NVMPLs. The usage model requires the programmer to identify persistent data and manage them within an NVRAM region using malloc/free-like calls. Like much recent research in this area [16,

19], this work assumes an explicit persistence model where persistent data are directly identified by the programmer at allocation time. In the context of this work, this persitence designation is done by using Makalu's APIs to allocate such data. This is in contrast to reachability-based persistent schemes [22, 34, 35] where programmers are not required to indicate persistence at object allocation time; instead, all data reachable from a persistent root must be made persistent. Failure-resilience of persistent data has a cost and the programmer may not want arbitrary data, such as passwords, to be transparently persisted. Thus this work chooses to favor explicit programmer control, also avoiding implicit reachability scans during execution.

Note that Makalu only guarantees the consistency of persistent heap structure, and not of the data stored within. The latter is expected to be provided by the `NVMPL` in use.

5.2.3 Terminology

Some commonly used memory allocation terms have the following meaning in this paper:
Memory object: A contiguous sequence of bytes in a persistent heap. The starting address is returned by the allocator while fulfilling a memory allocation request and the size corresponds to the number of bytes actually allocated.
Granule: The unit of actual memory allocated. In Makalu, all memory requests are rounded up to some multiple of the granule size, which is 16-bytes.
Block: A larger fixed-size contiguous sequence of bytes of virtual address space that can be divided into smaller memory objects to fulfill memory requests. In Makalu, the default size of a block is 4096 bytes (same as the page size in common operating systems). The starting address of a block is always page-aligned in Makalu.
Chunk: A contiguous sequence of one or more block(s).

5.3 NVRAM Allocator Challenges

Traditional factors, such as memory consumption and allocation speed, have historically influenced the design of transient memory allocators. Designing a persistent memory

allocator for NVRAM offers the following additional challenges.

5.3.1 Failure-Induced Inconsistencies

A failure in the middle of updating the allocator's metadata can lead to metadata inconsistencies. This work categorizes such inconsistencies as ***internal*** w.r.t the allocator. Such internal inconsistencies often have disastrous consequences such as a failure to restart properly, erroneous re-allocation of memory objects currently in use, or the leakage of large chunks of memory.

A failure may also cause discrepancies at the mutator/allocator interface. Such inconsistencies are classified as ***external*** w.r.t the allocator. For example, consider a scenario where a failure occurs after a call to malloc has returned (i.e. after an allocator has updated the internal metadata in a fail-safe manner) but before the returned address is stored in a persistent location by the mutator. This is essentially a failure-induced memory leak since upon restart, the allocator deems the returned memory object "allocated" though it is not reachable from any persistent root.

5.3.2 Transient vs. Persistent Metadata

This work classifies NVRAM allocators' metadata into two categories: ***core*** and ***auxiliary***. Core metadata is irrecoverable once corrupted. Auxiliary metadata on the other hand can be re-created using consistent core metadata. Although auxiliary metadata can be re-created at the beginning of each restart, and maintained in transient memory to avoid the cost of failure consistency, doing so may cause a long restart time, especially if recreating such auxiliary metadata is time-consuming. On the other hand, maintaining it in NVRAM may make restart instantaneous but at the expense of online overhead to maintain its failure consistency.

5.3.3 Online Failure Consistency Overhead vs. Recovery

Core persistent metadata often need to be updated in a way that always ensures their consistency. Such a mechanism frequently uses expensive instructions to flush cache lines after each update, and can adversely affect the allocation speed.

The consistency of auxiliary metadata stored in NVRAM need not be guaranteed as aggressively as the core metadata because it can be recreated from core metadata. For instance, an allocator may choose to only periodically flush cache lines containing updates to auxiliary metadata rather than after every update, so long as any resulting inconsistency is detectable. This approach reduces the online consistency overhead but increases the recovery time. If a failure occurs when the metadata is momentarily inconsistent, it has to be re-computed in the recovery phase. Hence, tradeoffs exist between recovery time vs. the online consistency overhead.

5.3.4 Safe Deallocation

Many of the existing NVMPLs such as Mnemosyne [16] and Atlas [19] use transactional logging mechanism to guarantee failure-atomic updates within a FASE. When a persistent allocator is used in conjunction with such an NVMPL in a multi-threaded application, it has to handle deallocation requests from within a FASE in the following special manner: the deallocation of the object itself and its reuse to fulfill future memory requests have to be delayed until the allocator can confirm that the deleted memory object will never become live in the future. Logs belonging to partially executed FASEs could potentially hold the memory object's address. When the log is replayed during recovery to restore the consistency of user data, the memory object may be accessed via the reference in the log despite the program's request to deallocate it earlier during the online execution phase. Memory allocators for programs written using software transactional memory have to tackle a similar problem (see [36]). A persistent allocator needs to offer a mechanism to delay deallocation requests until the NVMPL in use can confirm that the deallocations can be done safely.

5.4 Overview of the Approach

5.4.1 Integrated De-/allocation and Garbage Collection

Makalu is built upon the Boehm-Demers-Weiser garbage collector (**bdwgc**) [37]. This work chose to build upon the bdwgc framework instead of other transient allocators, such as Hoard [38] or jemalloc [39], because the approach described here relies heavily on garbage collection, and bdwgc provides inherent support for it. Grafting a garbage collector onto a separately designed allocator is nontrivial, and in fact, quite complex [40]. Having to deal with the persistence of both sets of data structures in Makalu's case makes such grafting even more complicated.

However, Makalu and bdwgc differ in several key aspects. In addition to the transformation needed for the allocator to become failure-resilient and function correctly across restarts, Makalu and bdwgc have some major differences in allocation and deallocation strategies. As bdwgc was designed for automatic memory management online, it has poor support for explicit deallocations. Explicit online deallocation is important in Makalu as it only supports GC offline. While bdwgc supports thread-local allocations, it does not support thread local deallocation. Each deallocated object is returned to the global freelist which requires holding a global lock. This approach lacks multi-threaded scalability and prevents immediate memory reuse. Makalu, on the other hand, supports thread-local deallocations, and also uses a simple strategy to tackle the issue of memory blowup in multi-threaded applications, when some threads favor allocation while others favor deallocation (see section 5.7). This was not a concern in bdwgc in the absence of thread-local deallocations. Makalu is compared with the intermediate modified version of bdwgc in the evaluation (see section 5.13). Our results show that Makalu has the ability (rare among allocators) to support both explicit deallocation and garbage collection efficiently.

Similar to bdwgc, Makalu's heap is structured as a Big-Bag-of-Pages, maintaining persistent metadata only at the block level and in separate headers (see section 5.5.1). This approach helps minimize the amount of persistent metadata. Core information stored in the

header about the layout of the heap (e.g. object size associated with a block) is considered to be irrecoverable once lost, and hence is only updated with ACID guarantees. Makalu relies on locks for isolation and a built-in log-based approach for ensuring all-or-nothing visibility for a set of updates to `NVRAM` (section 5.8).

In this work's setting, the garbage collector is run in a fully conservative mode, with no pointer location or type information communicated to the collector. This suffices to ensure metadata consistency. As discussed in section 5.9, other choices are also possible. The notion of offline garbage collection to mitigate failure-induced leaks, improve interoperability, programmability and online allocation performance explored in this paper can be applied in the context of strongly-typed languages such as Java and in the presence of precise garbage collection as well.

5.4.2 Choosing Persistent Metadata

To reduce the cost of persistent metadata updates, Makalu maintains a list of free objects in transient memory (see section 5.6) during the online phase. An allocation and a deallocation only require the corresponding memory object to be respectively removed from and added to the transient freelist. A failure may cause outstanding memory objects in Makalu's transient freelist to be lost momentarily. Makalu uses offline garbage collection to reclaim such objects and fix all other failure induced external inconsistencies based on the reachability of memory objects in the persistent heap. As a positive side effect, it also reclaims persistent memory leaked due to programming errors.

To avoid expensive computation at clean non-failure restarts, Makalu stores some selected auxiliary information, such as the header look-up table (see section 5.5.2), in persistent memory. It minimizes the expense of persistent updates to these structures during the online phase, by assuming them to be inconsistent after a failure, and rebuilding them offline from core persistent metadata. Thus, online updates to these structures only need to be visible in `NVRAM` by the time Makalu stops gracefully. This assumption enables Makalu to potentially accumulate some number of updates to persistent structures before

Interface	User	Description
MAK_start	N	One time call to set up and start Makalu in a new NVRAM region
MAK_restart	N	Restart Makalu from existing metadata
MAK_start_off	N	Restart Makalu offline
MAK_collect	N	Request GC offline
MAK_close	N	Signal Makalu to stop gracefully
MAK_set_free_cb	N	Set callback function for free (see section 5.11)
MAK_safe_free	N	Execute deferred free(s) (see section 5.11)
MAK_malloc MAK_calloc MAK_realloc MAK_free	P	Drop-in replacement for standard C/C++ de-/allocation methods
MAK_set_root MAK_get_root	P	Sets/gets top level NVRAM region roots

Table 5.1: List of Makalu's public interfaces. User: N = NVMPL, P = Programmer

having to guarantee visibility. (see section 5.8.1).

5.4.3 APIs Provided by Makalu

Table 5.1 presents a list of Makalu's major functions in its public interface. Programmers are only responsible for invoking a handful of these functions. For integration with a transaction-based NVMPL, Makalu provides interface to defer deallocation requests until the NVMPL in use can confirm that the deallocated object is truly unreachable (see section 5.3.4). While the programmer-facing interface is largely self-explanatory, the discussion of deferred deallocation and NVMPL-facing interface is deferred until section 5.11.

5.4.4 Comparison with Existing NVRAM Allocators

In both Mnemosyne [16] and Atlas [19], persistent memory allocations must be done within FASEs in order to guarantee the absence of failure-induced memory leaks. In addition to overheads incurred by failure-atomicity requirements of a FASE for every allocation, this

clearly is a programming constraint. NV-Heaps [17] provides automatic garbage collection using reference counting but requires weak pointers to correctly deal with cyclic data structures. Some existing NVRAM allocators, such as nvm_malloc [18], require two method calls just to allocate a persistent object. Other NVMPLs, such as pmem.io [21], combine allocation, initialization and publication steps into a single allocation method call. These interfaces are far removed from traditional allocation interfaces. Avoiding failure-induced memory leaks requires building a consensus between the allocator and the NVMPL after failure regarding what allocated memory objects are in use vs. free. There are several ways to obtain this consensus. One simple approach, but a burdensome one for programmers, is to require explicit code that traverses persistent data structures and reports them to the allocator after a failure, so that others can be deallocated. Another approach, one that seems to be taken by existing NVMPLs, is to tightly coordinate de-/allocation actions with the failure consistency semantics as described earlier. This work presents a superior approach that relies on offline garbage collection by tracing through application data structures to identify reachable NVRAM locations, essentially reaching consensus with the NVMPL in use. This way, familiar de-/allocation interfaces remain unchanged and can be called anywhere in the program, both within and outside FASEs.

```
BEGIN_FASE();
/* allocate */
Node* t_tmp = nvm_alloc(sizeof(Node));

/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;

/* publish */
p_queue->tail->next = t_tmp;
p_queue->tail = t_tmp;
END_FASE();
```

Figure 5.2: An allocation using NVMPL's default allocator [16, 19]

```
/* reserve */
Node* t_tmp = nvm_reserve(sizeof(Node));

/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;

/* allocate + publish */
BEGIN_FASE();

p_queue->tail->next = t_tmp;
nvm_activate(t_tmp, &(p_queue->tail), t_tmp,
           NULL, NULL);

END_FASE();
```

Figure 5.3: An allocation using `nvm_malloc` [18]

```
/* allocate */
Node* t_tmp = MAK_alloc(sizeof(Node));

/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;

/* publish */
BEGIN_FASE()

p_queue->tail->next = t_tmp;
p_queue->tail = t_tmp;

END_FASE()
```

Figure 5.4: An allocation using Makalu

Figures 5.2, 5.3 and 5.4 shows same code snippets using different flavors of persistent allocators and a generic `NVMPL` abstraction of a FASE. Prefixes 't_', and 'p_' denote transient and persistent program variables respectively. Each code snippet shows the common programming idiom of allocation, initialization, and publication of a persistent node of a queue. Using a generic abstraction of a FASE [19], each code snippet adds a node to the persistent queue in a fail-safe and thread-safe manner. With the `NVMPL`'s default allocator as shown in figure 5.2, allocation and publication must happen within the same FASE to avoid failure-induced memory leaks, resulting in a large FASE. Another stand-alone persistent allocator, `nvm_malloc` [18] as shown in figure 5.3 supports reserving and initializing a persistent memory object before entering the FASE. However, the actual allocation and publication steps must still occur within the FASE as shown in figure 5.3 because `nvm_malloc` combines allocation and publication steps into a single allocation method. The primary inconvenience of this approach is the non-traditional interface. The use of Makalu, as shown in figure 5.4, results in the smallest FASE, and a more familiar programming paradigm. In fact, while going from a transient to a persistent version of this code, the only change necessary in the Makalu-enabled version is the replacement of the allocation call with Makalu's corresponding one.

Note that the primary goal of this work is to understand the challenges of developing an interoperable and leak-free allocator, and investigate design decisions that can minimize failure consistency overhead. Improving the raw performance of an allocator is only a secondary goal.

5.5 Internal Structures and Layouts

This section describes the internal structures of Makalu and how they are laid out in persistent memory within the `NVRAM` region. Each such structure in Makalu is classified as either ***core*** or ***auxiliary*** metadata as described in section 5.3.2.

```
header {
   void* hb_block;
   long hb_sz;
   int hb_flag;
   long hb_mark_bits[BITS_SZ];
   long hb_n_mark_bits;
   ...
}
```

Figure 5.5: Structure of the block header

5.5.1 Persistent Block Header

Both an individual block as a well as a contiguous set of blocks (a chunk) in Makalu have a persistent header associated with them. Figure 5.5 shows the important header fields. The fields `hb_block`, and `hb_sz` store the starting address of a block or a chunk and its total size respectively. The field `hb_flag` indicates whether a block or a chunk is currently in use or free. If a block is currently used to fulfill memory requests, `hb_sz` stores the size of memory objects allocated from that block. Note that all objects allocated from a single block are of the same size in Makalu. Makalu regards the above three fields in each header as part of the core metadata and the rest are auxiliary. Makalu updates these fields using built-in support for ACID guarantees described in section 5.8.

Each object in a block has a corresponding bit in `hb_mark_bits`. If the bit is set, the object is either already allocated or some thread has already added to its freelist intending to allocate it. Alternatively, a thread cannot add to its freelist an object within a block whose mark bit is already set.

Field `hb_n_mark_bits` stores the count of mark bits currently set for convenience purpose. Although mark bits are stored in persistent memory, Makalu regards them as auxiliary metadata because it has the ability to recreate them offline based on the reachability of objects using GC.

A block header is allocated within ***header spaces***, which are one or more fixed-length sections of memory within an `NVRAM` region (and outside the heap) specifically designated

for this purpose and shown in figure 5.6. Allocating headers only in header spaces enables Makalu to precisely know where all the core information is located so that it can be found during recovery to recreate auxiliary metadata.

Occasionally, one or more free adjacent blocks having their own headers coalesce to form a chunk requiring only a single header for the chunk beyond that point (see section 5.7.2). This block coalescing action frees up one or more headers which are added to the ***header freelist (HFL)*** for future reuse. HFL is auxiliary metadata as it can be recreated by scanning header spaces for all free headers.

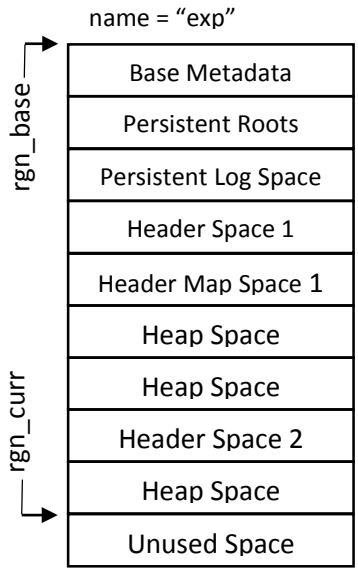


Figure 5.6: Layout of Makalu in an NVRAM region

5.5.2 Persistent Header Map

The header map in Makalu provides a method to conveniently look up corresponding header information for a given memory object, a block or a chunk. Additionally, it provides iterators to selectively iterate over blocks (free vs. allocated) via headers and without

touching the actual blocks. This map is adopted from bdwgc [41]. It is stored in ***header map space***, another specially designated section of memory within an `NVRAM` region for this purpose and shown in figure 5.6.

This map is auxiliary metadata that can be re-created in the header map space from scratch by adding each header from one or more header space(s). It is nonetheless stored in `NVRAM` to avoid the expense of rebuilding it at each normal re-start. A single scan through a header space during offline recovery is enough to create both the header map and the HFL. Therefore, Makalu maintains their failure consistency less aggressively (see section 5.8.1) and rebuild them from scratch after each failure, which presumably should be rare.

5.5.3 Persistent Log Space

Makalu uses a log-based approach to update the core metadata in a fail-safe manner (see section 5.8). Since the persistent logs must be consistent at all times, the log space is a core data structure. As part of setting up a new region, it designates a fixed amount of space, as shown in figure 5.6, within the `NVRAM` region for persistent logs to be written. This space is reused repeatedly across execution cycles.

5.5.4 Persistent Roots

Makalu designates a ***persistent root space*** (separate from the heap space) within each `NVRAM` region for storing top-level `NVRAM` region roots. Makalu supports up to 512 top-level `NVRAM` region roots so that useful data within the `NVRAM` region's heap space can be conveniently accessed. Any i^{th} region root can be accessed using setter and getter methods listed in table 5.1.

5.5.5 Persistent Base Metadata

Apart from the information in the block header, Makalu also maintains the following information as part of the core metadata.

- `NVRAM` region base (`rgn_base`) and max heap (`rgn_curr`) address

- Log space starting address (`md_log_space_start`)
- Current log version (`md_log_version`) (see section 5.8)
- List of header spaces
- Address of the header map space
- Persistent root space start address

All of the above information is stored in base metadata space shown in figure 5.6. Most fields in base metadata, such as log space starting address, do not change once set. Other metadata fields, such as the current log version, are updated with ACID guarantees when necessary. Recovery is not possible without the consistent information provided by each field in the base metadata.

5.5.6 Transient Chunk Freelists

During the online phase, Makalu maintains a global transient list of free chunks segregated by the number of free blocks in them. When Makalu restarts, it recreates the list by adding all free chunks to it using the iterator provided by the header map.

5.5.7 Transient Object Freelists

Makalu classifies memory objects as *small* ($\leq$ 400 bytes), *medium* ($>$ 400 bytes, $\leq$ half a block), and *large* objects ($>$ half a block). During the online phase, Makalu maintains small object freelists on a per thread basis and global freelists for medium objects. Both small and medium object freelists are segregated by object size and each freelist is a transient LIFO list. Large object de-/allocations are handled directly by the chunk freelist (see section 5.6).

5.5.8 Transient Reclaim Lists

Makalu creates a list of partially allocated blocks (containing some free objects) at the beginning of each online phase by iterating over allocated blocks using the iterator provided by the header map. The reclaim list is a global transient structure organized as a set of

object-size-segregated lists. Such a list enables us to sweep one block at a time looking for free objects of a particular size to refill a freelist.

5.6 Allocation

Allocation requests for small and medium objects are rounded up to the nearest granule multiple. For small objects, the allocating thread simply removes and returns the first item in the thread local freelist for that size. For medium sized objects, it removes and returns the first item from the appropriate global freelist after acquiring a lock for that size. Allocating objects from a transient freelist requires no persistent updates – note that the mark bit for the object is already set by this time (see below).

5.6.1 Refilling an Empty Freelist

If a freelist for a particular size is empty, the allocating thread performs an *incremental sweep*, i.e. it scans a partially allocated block to refill the freelist. It acquires a lock for a reclaim list corresponding to that size, removes the first block in the list and looks up the header for the block using the header map. Each free object, as indicated by its mark bit in the header, is added to the transient freelist while simultaneously setting the mark bit. A set mark bit indicates to another thread (going through the block for free objects at later times) that the object is either already in someone else's freelist or allocated.

A gracefully terminating thread gives each remaining object in its transient freelists back to the block by clearing the corresponding mark bit in the header for that object (so that other threads can use it to refill their freelist). During this process, if Makalu notices that a handful of objects have become available in a specific block, it adds that block back to the reclaim list making it available to other threads for sweeping. Before a graceful shutdown, Makalu also reinstates all objects in medium object freelists to the corresponding block in a similar manner.

All persistent updates to mark bits are guaranteed to be visible in `NVRAM` by the time Makalu gracefully stops, using techniques described in section 5.8.1. This guarantee is

sufficient for mark bits to be reliably used in the next online execution cycle (to create the reclaim list) following a graceful shutdown.

A failure will momentarily leave objects in transient freelists unaccounted for. Therefore, Makalu starts with a clean slate in recovery mode and reconstruct a set of mark bits for each block, purely based on object reachability, by using GC. The objects residing in freelist prior to a failure would appear allocated but unreachable in the heap, and are guaranteed (modulo GC conservatism) to be discovered by the offline GC. Hence, the GC's presence enables us to cheaply maintain frequently updated freelists in transient memory.

If the reclaim list for a particular size is empty, the allocating thread allocates a new block (see below) from the chunk freelist. All objects from the newly allocated block are then added to the empty freelist after setting the corresponding mark bits in the header.

5.6.2 New Block Allocation

To allocate a new block, Makalu searches for the smallest chunk (c_s) (ideally a chunk with a single block) available among the transient chunk freelists. Recall that the chunk freelist is segregated by the chunk size (number of blocks in them). It removes c_s while holding a lock for the specific freelist where it finds c_s. Next, Makalu obtains the header for c_s. A block is allocated from c_s using the following steps:

1. Remove the first block (b_1) from c_s by adjusting the chunk's size (`hb_sz`) and the beginning address (`hb_block`) in its header.
2. Allocate a header for b_1 and add it to the header map.
3. Set b_1 header fields `hb_block`, `hb_sz` with the starting address, and the size of the object to be allocated from b_1 respectively. Set the `hb_flag` indicating currently in use.

Once the block is allocated, the remaining portion of c_s is returned to the chunk freelist appropriate for its new size.

Failure-induced partial `NVRAM` updates from steps 1-3 above can cause inconsistencies in the core persistent metadata, leading to undesirable effects, such as a permanently leaked

block. Therefore, the allocating thread performs all persistent updates to core metadata in steps 1-3 with ACID guarantees (see section 5.8 for failure atomicity).

5.6.3 Large Object Allocation

Allocation requests for large objects are rounded up to the nearest multiple of a block and serviced directly from the chunk freelist. The process is similar to allocating a new block described in section 5.6.2.

5.6.4 Expansion of Heap Space

The allocating thread expands the heap when the chunks are insufficient to fulfill an outstanding memory request. Heap expansion requires performing the following steps with ACID guarantees because updates to core metadata are involved.

1. Increment the `NVRAM` region bump pointer, rgn_curr (stored as base metadata, see section 5.5.5)
2. Allocate a header for the acquired chunk and add it to the header map
3. Store chunk's size, and the starting address in the header, and set the flag in the header

Once the heap is expanded, allocations can occur as described earlier.

5.7 Deallocation

Using the header map (section 5.5.2), the deallocating thread computes the block and the size of the object from the address being deallocated. If it's a small object, it's added to the start of the deallocating thread's local freelist corresponding to its size. It does not require any update to persistent metadata. Makalu does not attempt to return the object to the allocating thread. This is in contrast with the approach taken by some transient allocators such as Hoard [38] to prevent an allocator from inducing false cache line sharing in multi-threaded applications. Similar to [42], Makalu expects a programmer to allocate

cache-aligned objects where false sharing is a concern. Medium-sized objects are added to the corresponding global freelist after acquiring the per size lock (see section 5.6).

5.7.1 Object Freelists Truncation

The total bytes of memory held in each medium and small object freelist is capped at twice the size of the block. If a transient freelist (for small and medium objects) grows to its maximum capacity when a thread deallocates an object, the thread is responsible for truncating the freelist by half, leaving the top half of the objects for future allocation requests. This design has the following advantages:

- It prevents unbounded memory blowup in applications with producer-consumer de-/allocation patterns among threads [38].
- It bounds the amount of work that a gracefully terminating thread has to perform to purge its small object freelist. It also bounds the number of medium objects in the global freelist that Makalu has to process before a graceful shutdown.

To truncate a free list, a thread removes one object at a time from the freelist, looks up the block header for the object removed from the freelist, and marks the object as available in the future for other threads (to add to their freelist) by clearing the corresponding mark bit in the block header. Moreover, if a thread notices that a sufficient number of objects have become free in a block when clearing the mark bit, it adds the block back to the reclaim list. It acquires a lock for the appropriate reclaim list to do so. Note that objects in a single freelist may come from more than one block due to a number of factors such as objects being deallocated from a previous execution cycle, remotely allocated object being added to the local freelist following a deallocation and so on. Consequently, one or more blocks may be put back into the reclaim list.

Other threads can re-use the block returned to the reclaim list to refill their freelists at later times (section 5.6). All updates to mark bits are guaranteed to be eventually visible by the time Makalu shuts down gracefully (see section 5.8.1). If a failure occurs before all

mark bits become visible, GC will start with a clean slate and create a consistent set of mark bits in recovery mode as described in section 5.6.1.

5.7.2 Empty Block Deallocation

It is quite possible for the entire set of mark bits to be cleared for some block during the freelist truncation. A set of all clear mark bits indicates that objects belonging to the block are neither allocated nor do they reside in any of the transient freelists. At this point, it is safe for Makalu to deallocate the entire block and add it back to the chunk freelist. This enables the block to be re-used to fulfill memory requests for another size class. Figure 5.7 shows the pseudocode for block deallocation. The deallocating thread attempts to coalesce with the immediately preceding (lines 9–20) or the following block/chunk (lines 21–30) (if they exist and are free) to create a larger chunk of memory in the heap. All updates to the core metadata in header fields are performed using an internal interface (`store_nvm_*`) within an all-or-nothing code section demarcated by `start_nvm_atomic` (line 8) and `end_nvm_atomic` (line 36) (see section 5.8). If a failure occurs before the complete set of updates to the header metadata in method `deallocate` becomes visible in `NVRAM`, partial updates are guaranteed to be undone restoring the consistency of all headers involved. The GC will subsequently rediscover the completely empty block offline and will deallocate it using the same fail-safe approach.

5.7.3 Large Object Deallocation

Large object deallocation returns such objects directly to the appropriate chunk freelist using steps similar to those described above for empty allocated blocks.

5.8 ACID Guarantees for Metadata

Makalu uses internal interfaces presented in figure 5.8 to specify a set of persistent stores (to core metadata) that must be atomic w.r.t failure, i.e. which need to be visible in `NVRAM` on an all-or-nothing basis. Makalu uses undo-logs to enforce failure atomicity guarantees.

```
1: deallocate(Block* b)
2: {
3:  lock(gml);
4:  hdr* cHdr = map.find(b);
5:  hdr* pHdr = map.find(b-1);
6:  hdr* nHdr = map.find(b+1);

7:  /* ACID section */
8:  start_nvm_atomic();
9:  /* coalesce with previous block */
10: if (pHdr && isFree(pHdr)) {
11:  removeFromChunkFL(pHdr -> hb_block);
12:  /* ACID update, core metadata */
13:  store_nvm_word(&pHdr -> hb_sz,
14:    pHdr -> hb_sz + BLOCK_SZ);
15:  store_nvm_word(&cHdr -> hb_sz, 0);
16:  store_nvm_addr(&cHdr -> hb_block,NULL);
17:  uninstall(cHdr);
18:  cHdr = pHdr;
19:  coalesced = 1;
20: }
21: /* coalesce with next block */
22: if (nHdr && isFree(nHdr)){
23:  removeFromChunkFL(nHdr->hb_block);
24:  store_nvm_word(&cHdr->hb_sz,
25:   cHdr->hb_sz + nHdr->hb_sz);
26:  store_nvm_word(&nHdr->hb_sz, 0);
27:  store_nvm_addr(&nHdr->hb_block, NULL)
28:  uninstall(nHdr);
29:  coalesced = 1;
30: }
31: /* update the current hdr */
32: if (!coalesced){
33:  store_nvm_word(&cHdr->hb_sz, BLOCK_SZ);
34:  store_nvm_int(&cHdr->hb_flag, FREE);
35: }
36: end_nvm_atomic();
37: addToChunkFL(cHdr);
38: unlock(gml);
39:}
```

Figure 5.7: Pseudocode for empty block deallocation

`start_nvm_atomic` (lines 1–3) marks the beginning of the set of failure-atomic writes. A call to this method is always preceded by an acquisition of the global mutex lock which provides the isolation guarantees amidst multiple mutator threads. Within the atomic section, core metadata is modified using one of the `store_nvm_*` methods (lines 8–26) based on the metadata type. Details for the integer method are shown in the figure (lines 8–23). Each of these methods creates a log entry containing the memory address, the current value of the metadata , and its data type (lines 9–12) before storing the new value. The new log entry is published (lines 14–16) by stamping the entry with the current value of `md_log_version`. Recall that the current value of `md_log_version` is stored as Makalu's core base metadata (see section 5.5.5). Makalu ensures that the log entry is visible in `NVRAM` using memory fences (line 13) and cache line flushes (line 16) before storing the new value and making the value visible in `NVRAM` (lines 19–22). A call to `end_nvm_atomic` marks the end of the failure-atomic updates by incrementing the `md_log_version` and flushing it to `NVRAM` (lines 4–7).

If a failure occurs before the incremented value of `md_log_version` becomes visible in NVRAM, Makalu starts in an offline phase. It then infers that there are partial updates from the previous run if there is a log entry in the designated log space with the same version as `md_log_version` (the current log version visible in `NVRAM`). The last log entry with that version is obtained and the undo entries are applied in reverse order of their creation, thus nullifying the effects of original updates. Once it has fully replayed the relevant log entries and flushed all the effects of replay to `NVRAM`, it increments the log version (while still offline) and makes it visible in `NVRAM`. The number of log entries never grows beyond a certain number (20 entries) because the provided interface is only used internally in specific scenarios. Each scenario has a statically known number of related updates.

```
1:  start_nvm_atomic(){
2:   next = md_log_space_start;
3:  }
4:  end_nvm_atomic(){
5:   md_log_version++;
6:   FLUSH(md_log_version);
7:  }
8:  store_nvm_int(int* addr, int val){
9:   /* create a log entry */
10:  next->addr = addr;
11:  next->val.int_val = *addr;
12:  next->type = INT;
13:  MEMORY_FENCE();
14:  /* publish */
15:  next->version = md_log_version;
16:  FLUSH(next);
17:  /* next log entry pos. */
18:  next++;
19:  /* store the value */
20:  *(addr) = val;
21:  /* make the update visible */
22:  FLUSH(addr);
23: }
24: store_nvm_word(int* addr,int val);
25: store_nvm_char(int* addr,char val);
26: store_nvm_addr(void** addr,void* val);
```

Figure 5.8: Internal facility for failure-atomic updates

5.8.1 Eventual Visibility for Metadata

The consistency of auxiliary metadata such as header map and mark bits is guaranteed using a technique less aggressive than one described above. Multiple updates to these metadata can be allowed over the course of online execution before issuing a cache line flush to ensure their visibility. All updates only need to be visible by the time of the graceful shutdown to avoid expensive recovery and restart. We explicitly guarantee their visibility before the graceful shutdown to avoid the chance of dirty cache lines getting lost because of a hardware failure between graceful process termination and next restart. We use a scheme, roughly analogous to one used in [19], that tracks multiple updates to the same cache line using a fixed-size hash table. Modified cache lines that have not yet been flushed appear in the table. Hash table collisions are resolved by flushing the cache line corresponding to the previous entry and removing it. After the last update to auxiliary metadata before the graceful shutdown, the hash table is emptied by flushing each dirty cache line being tracked in the table.

This table enables Makalu to reduce the number of CLFLUSH and MFENCE (CLWB and SFENCE in future ISA) issued.

5.9 Offline Recovery and GC

The offline phase has distinct steps that must be initiated in the following sequence:

5.9.1 Recovery

Following a failure, Makalu starts in an offline mode. The log replay ensures that the core metadata is restored to a consistent state. Next, Makalu purges the existing header map and builds a new one by scanning each header space and adding headers currently in use to the header map. Mark bits in the header are also cleared in the process. Free headers found (not currently assigned to any block) are added to HFL (section 5.5.1).

5.9.2 Garbage Collection

The NVMPL with which the allocator has been integrated typically makes a request to collect garbage once it has restored the user data in the heap to a harmonious state. Such garbage collection is significantly simpler than a conventional garbage collector since there is no mutator present, and the set of garbage collection roots is limited to a small set of explicitly specified region roots. The effort that garbage collectors normally invest in, for example, stopping threads and parsing thread stacks, is unnecessary. So are the usual constraints on the compiler to avoid concealed pointers in registers, etc.

It would be possible to restrict NVRAM data structures so that pointer locations in NVRAM-allocated objects are apparent. Makalu could require that for NVRAM data structures, roots and pointer fields be declared with their correct type (and not, for example, as `void *` or `intptr_t`), and that unions be suitably restricted. Makalu could adopt at least some such restrictions or annotations in the future so that the garbage collector can reliably clear persistent-to-transient pointers. Such pointers are obviously invalid after a process restart.

For now, we instead adopt the parallel mark and sweep algorithm from bdwgc as described in [37,43] in a fully conservative mode for C/C++ setting in this work. It imposes the fewest restrictions on reuse of existing code in an NVRAM setting. Although this clearly affects the details we describe below, we do not believe it affects the fundamental benefit of greatly reducing the cost of metadata updates during allocation.[2]

The mark phase starts by analyzing persistent roots explicitly registered as part of the NVM region (see section 5.5.4).

The entirely empty blocks found at the end of the mark phase are deallocated using the process described in section 5.7.2. All mark bits for each partially allocated blocks are made visible in NVRAM before the offline recovery completes. This enables the reliable

[2] Note that the "black-listing" mechanism in bdwgc is not currently functional in this setting, since we have no data from prior GCs about misidentified pointers to unallocated memory. One can expect that, due to the small root set size, it is also much less necessary than usual. It could be restored with a fully concurrent, approximate, trace phase while the program is running. Since this would not be used to reclaim memory, it could be allowed to err in both directions. There should not be any need to synchronize with the mutator. Or we could just broaden the scope of garbage collection.

recreation of reclaim list at the beginning of the online restart and threads to subsequently refill object freelists using blocks in the reclaim list (i.e. perform online incremental sweeping; see section 5.6.1).

5.10 Execution Stages and Failure Mitigation

Makalu is designed to handle tolerated failures at all stages of its execution, offline and online. This section summarizes Makalu's expected behavior in the case of a failure at various stages of execution.

5.10.1 One-Time Online Initialization

The initialization is considered complete once the call to the method `MAK_start` returns. At the end of the method call, Makalu flushes all persistent metadata updates to `NVRAM`. Next, it stamps the `NVRAM` region with a "magic number" and flushes this update synchronously to `NVRAM` before the method returns. Each time Makalu restarts, it checks for this magic number to ensure that the given `NVRAM` region has been initialized properly. If a failure occurs before this number is visible in `NVRAM`, Makalu requires the client `NVMPL` to re-run the initialization routine. If a failure occurs after a successful initialization and before the first call to de-/allocation routines, Makalu's persistent metadata remains unaffected and hence, a client `NVMPL` may optionally skip Makalu's offline recovery routine altogether.

5.10.2 Online Re-initialization

Recall that a client `NVMPL` uses method call `MAK_restart` to re-initialize Makalu from its persistent metadata each time `NVRAM` region is reopened for access. Makalu only reads the persistent metadata to recreate transient structures during this stage. Hence, a failure does not affect Makalu at all at this stage. A client `NVMPL` can optionally skip Makalu's offline recovery routine in this case as well.

5.10.3 Online Execution

Any failure after the first call to de-/allocation routines and before the call to `MAK_close` can corrupt Makalu's metadata and lead to persistent memory leaks. In this case, a client `NVMPL` is expected to restart Makalu offline and invoke recovery/GC routine.

5.10.4 Offline Recovery

Recall that Makalu's offline recovery has the following three distinct steps. A failure before the completion of all three recovery steps requires Makalu to be restarted in offline recovery mode.

Step 1: Persistent log replay. First, Makalu replays outstanding persistent undo logs (if present) to restore the consistency of core metadata (see **??**). After Makalu flushes all the updates from log replay to NVRAM, it increments the log version and flushes it synchronously. If a failure occurs before the new incremented log version is visible in `NVRAM`, it detects the same outstanding logs and replays them again.

Step 2: Reconstruction of auxiliary metadata. Recall that Makalu recreates auxiliary persistent metadata from consistent core metadata. If a failure occurs after the successful completion of step 1 and before the completion of step 2, the recovery resumes by discarding the partially constructed auxiliary metadata, reclaiming the metadata space and reconstructing the auxiliary metadata in that space.

Step 3: Garbage collection. If a failure occurs during a GC or anywhere else after step 2 and before the call to `MAK_close` returns, Makalu discards partially constructed mark bits and restarts the mark phase with a clean slate. When the call to `MAK_close` returns, a complete set of mark bits are guaranteed to be visible in `NVRAM` and the recovery is complete.

5.11 Integration with NVMPL

5.11.1 NVMPL Facing Interfaces

Makalu's public interface (Table 5.1) provides an easy out-of-the-box integration with NVMPLs. An NVMPL is expected to enable Makalu to set itself up in a new NVRAM region using the one-time call to MAK_start. Each time an NVRAM region is reused online, Makalu expects an NVMPL to reinitialize it via MAK_restart. Makalu restarts from the metadata stored in NVRAM region and gets ready for de-/allocation requests by rebuilding various transient internal lists such as reclaim lists and chunk freelists from persistent metadata. As a part of closing the region, Makalu expects the NVMPL to call MAK_close to signal it to shutdown gracefully. This involves taking down transient freelists and guaranteeing visibility of all persistent metadata updates.

For reasons discussed in section 5.3.4, Makalu may need to defer deallocation requests made using MAK_free. Makalu allows the NVMPL to set up a deallocation callback method (which is invoked by Makalu each time MAK_free is called) via MAK_set_free_cb. For each object reported by Makalu through the callback, the NVMPL can use a secondary MAK_safe_free method to actually deallocate when it is safe to do so. This approach neatly hides the complexity of deferred deallocation from the programmer.

After a failure, the NVMPL is expected to start the recovery phase by using MAK_start_off. At the end of this method call, Makalu would have recovered its metadata to a consistent state. Once the user data is in a harmonious state, the NVMPL can invoke GC using MAK_collect.

5.11.2 Integration with Atlas and Mnemosyne

The default allocators found in two published NVMPLs, Atlas [19] and Mnemosyne [16], were substituted with Makalu. These two NVMPLs differ in their failure consistency semantics. Atlas infers durable critical sections from lock-based code, whereas Mnemosyne builds support for persistence around software transactional memory for persistence.

The integration was straightforward for the most part. Deferring deallocation was the most challenging aspect of integration in both cases. Both set up a deferred callback method at the beginning of each execution cycle. In Mnemosyne, deallocations within a transaction have to be deferred. Mnemosyne creates a list of objects which are deallocated within the transaction (as reported by Makalu using callback) and deallocates them as a post-commit action using `MAK_safe_free` method.

Similar to Mnemosyne, Atlas creates a list of objects deallocated (reported by Makalu) within a failure-atomic section (FASE) and associates the list with the current FASE. The logs associated with a FASE are pruned by a distinct helper thread in Atlas [19]. When the helper determines that a certain FASE can be pruned, it deallocates all objects in the list associated with the FASE using `MAK_safe_free` method. In both Mnemosyne and Atlas, a programmer only interacts with the standard malloc/free interface, while the complexity of deferred deallocation is completely hidden from them.

Although this work uses log-based `NVMPLs` to evaluate Makalu's approach, the allocator works with other forms of `NVMPLs`, say an `NVMPL` based on a copy-on-write approach for failure consistency. Note that Makalu's internal metadata or its internal log is never exposed to the programmer or `NVMPL` and has absolutely no relation with the log generated by a log-based `NVMPL`. Programmers and `NVMPLs` must concern themselves with only Makalu's API listed in table 5.1.

5.12 Related Work

Our work is most closely related to `nvm_malloc` and pmem [18, 21][3]. `nvm_malloc` differs from our work in two distinct ways. In `nvm_malloc`, memory allocation requires two method calls. The first only reserves persistent memory of the requested size. The second method takes the returned persistent address, together with at least one persistent result location, so that it can failure atomically perform the actual allocation and store (publish)

[3] Both of these work essentially use a similar two-step allocation algorithm that we henceforth refer to as `nvm_malloc`.

the allocated memory address in the provided location. This is thus not a drop-in replacement for conventional malloc and pays a higher cost than Makalu for every allocation. Finally, `nvm_malloc` does not address interfacing with other `NVMPL`s. It does not provide a mechanism to defer deallocations. Hence, it is not clear to us how it operates correctly with `NVMPL`s such as Atlas [19] and Mnemosyne [16], which use a log-based approach to support failure-atomic updates of persistent data.

None of the existing `NVMPL` allocators provide a safety net against programmer-induced memory leaks. Makalu's offline garbage collection works as this safety net. NV-Heaps [17] is the only published `NVMPL` which offers anything comparable, by providing a reference counting GC. However, it requires a programmer to distinguish between a strong and a weak reference to break cycles in a persistent heap. We argue that this is error-prone, especially when dealing with large and complex data structures. NV-Heaps is not publicly available and the paper does not indicate that GC was used as a means of reducing the `NVRAM` allocation cost, nor is it clear this could be done.

A plethora of transient memory allocators have been developed to satisfy the scalability needs of multicore computing [**?**,**?**,38,39]. Our work differs from these as our core objective is low cost *crash-resilient persistent* memory management in addition to multi-core scaling. The next section does compare our allocator with a commercially used transient allocator, Hoard [38].

Our offline garbage collection is an adaptation of the conservative garbage collection algorithm implemented in the bdwgc collector [37, 43]. However, garbage collection occurs only offline, when top-level roots are precisely known. Bdwgc does not support `NVRAM` allocation, and its free() implementation does not scale on multiprocessors.

5.13 Evaluation

Although `NVRAM` is getting closer to becoming widely available [44, 45], it is not at the moment. Hence, Linux *tmpfs* [25] was utilized to simulate NVRAM during the collection of results. As files in *tmpfs* are only backed by DRAM, a process crash and restart mechanism

was used to test Makalu's crash resilience and restart logic. In order to measure failure consistency overhead, full failure consistency mechanisms such as cache line flushes and memory fences were enabled, as it would have been in actual NVRAM systems.

Unless otherwise stated, all the code used in this evaluation were compiled using the GNU gcc/g++ compiler, version 4.8.4 at optimization level "`-O2`". All experiments were run on a 64-bit Ubuntu machine (kernel version 3.13.0-66-generic) that has 12 GiB of RAM, two Intel Xeon E5-2695 processors, 6 cores per socket (12 cores total), and hyper-threading switched off. All results collected were averaged over 6 runs.

We measured the persistence overhead in Makalu using CLFLUSH and MFENCE. Using these instructions enabled us to easily compare Makalu with other allocator as other NVRAM allocator nvm_malloc and `NVMPLs`such as Atlas and Mnemosyne are also implemented using such instructions. This is however a pessimistic estimate of Makalu's performance. With better instructions such as the combination of CLWB and SFENCE in future ISA to publish dirty cache line in `NVRAM`, we expect the performance of Makalu to improve further for the following reasons. First, each combination of CLFLUSH and MFENCE is replaceable by a combination of CLWB and SFENCE. This is true for Makalu's logs where each log has to be written syhnchronously before updating the metadata. However, all updates to the metadata (tracked in a table by Makalu) within a transaction only need to be visible before the transaction commits. Hence, a single SFENCE before the transaction commits is sufficient for all the CLWB issued within the transaction. Likewise, for lazy consistency guarantees (see section 5.8.1), a single SFENCE is necessary at the end of emptying the table that tracks the updated cache lines.

5.13.1 Comparison with Existing Allocators

The allocation speed, throughput, and multi-threaded performance of Makalu was compared with another persistent allocator, ***nvm_malloc***, which is built upon jemalloc [18]. Although the goal of this work is not to produce the fastest or the most scalable transient allocator, Makalu was compared with ***Hoard***(ver3.10) [38], one such popular allocator, to

show that Makalu's raw performance is comparable despite the extra failure consistency overhead it has to incur. Also include in the comparison are the following three versions of bdwgc:

1. An unmodified version (ver7.2) with multi-threaded GC, thread-local allocation enabled, and explicit deallocation disabled (***bdwgc***).
2. Same as (1), but with GC disabled and the default sub-optimal explicit deallocation enabled (***bdwgc-free***).
3. A modified version of (1), with a better support for explicit deallocation and GC disabled (***bdwgc-mod***).

Benchmarks

The following often used allocation benchmarks were used for comparison.

Larson: This benchmark has often been used to simulate a multi-threaded long running server [37, 38, 42, 46, 47]. The benchmark was configured to run for 10 seconds, with t threads where each thread runs for 10^4 rounds, allocating and deallocating 10^3 64-byte[4] objects in each round. It reports the allocation throughput in a given time window in terms of operations/sec.

Threadtest: This benchmark has been used by [38, 42, 46] to measure multi-threaded scalability performance of an allocator. Each thread allocates and deallocates memory in a tight loop with a configurable amount of work in-between. For t threads, the benchmark was run such that each thread performed 10^4 rounds of de-/allocation, and $\frac{10^5}{t}$ de-/allocations in each round.

Prod-con: The benchmark simulates applications which have two mutually exclusive sets of allocating and deallocating threads. Such de-/allocation pattern typically causes memory blowups [38] or performance degradation in poorly designed allocators. The benchmark starts an even number of threads t, and creates $\frac{t}{2}$ blocking queues [48]. It

[4] This is the smallest common allocation size; `nvm_malloc` only supports allocations in the multiples of 64 bytes

then assigns a producer-consumer thread pair to each queue. An object is allocated by a producer thread, passed to a consumer thread via a queue, and deallocated there. Each pair of producer-consumer threads de-/allocates $\frac{2\times10^7}{t}$ objects that are 64 bytes in size.

Results

The multi-threaded performance results presented in figure 5.9 shows that Makalu performs orders of magnitude better than `nvm_malloc` in common memory allocation patterns presented by above benchmarks. The difference in performance between Makalu and bdwgc-mod is remarkably thin, making it safe to say that Makalu is only slightly burdened by failure safety of its metadata. The failure consistency overhead of Makalu was compared with `nvm_malloc` in terms of the average number of cache line flushes issued per thread and present results in figure 5.10 for all three benchmarks.

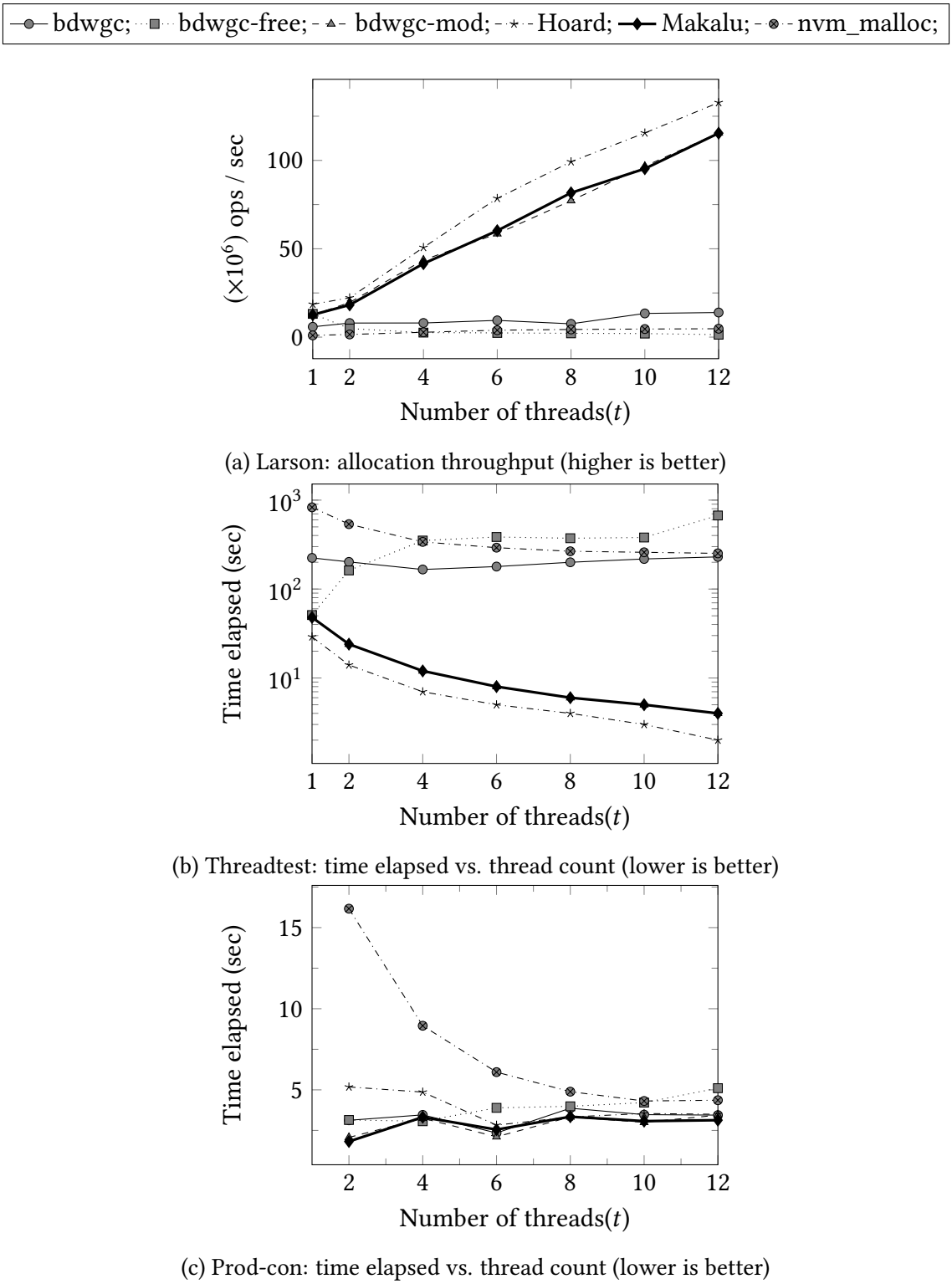
Figure 5.11 compares memory consumption of Makalu (transient + persistent) using the running process's peak Resident Set Size (RSS). Linux OS mainatains this information for each running process. Makalu's memory consumption is somewhere between the best and worst case observed and is comparable to that of `nvm_malloc` across all three benchmarks. This is especially true as the thread count increases. Notice also that Makalu outperforms both bdwgc and bdwgc-free especially in the case of prod-con benchmark mostly because Makalu implements measures to prevent memory blow-ups. Prod-con benchmark is suseptible to such memory blowups.

5.13.2 Recovery and Garbage Collection

Makalu achieves low online failure consistency overhead at the cost of offline recovery and GC in the rare event of failure. This section quantifies this offline cost.

Benchmark

The offline recovery and GC cost is evaluated using the following benchmark:

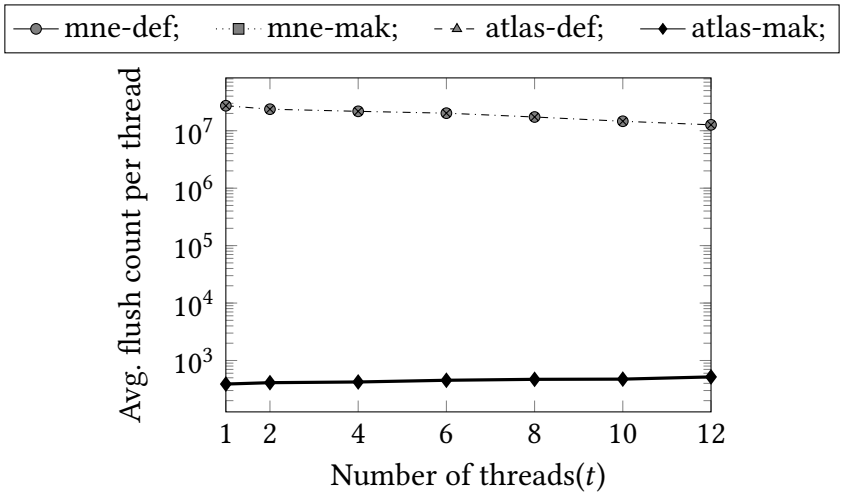


(a) Larson: allocation throughput (higher is better)

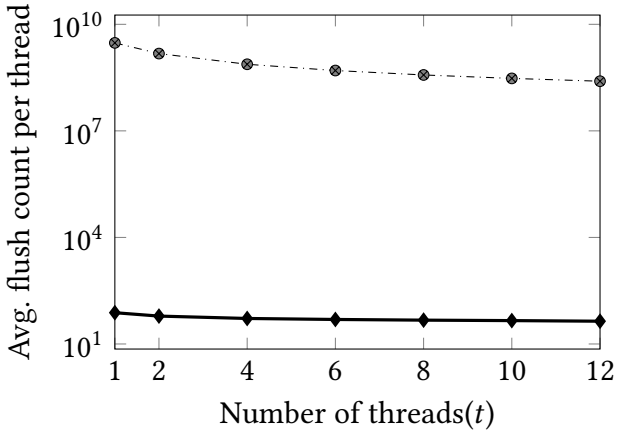
(b) Threadtest: time elapsed vs. thread count (lower is better)

(c) Prod-con: time elapsed vs. thread count (lower is better)

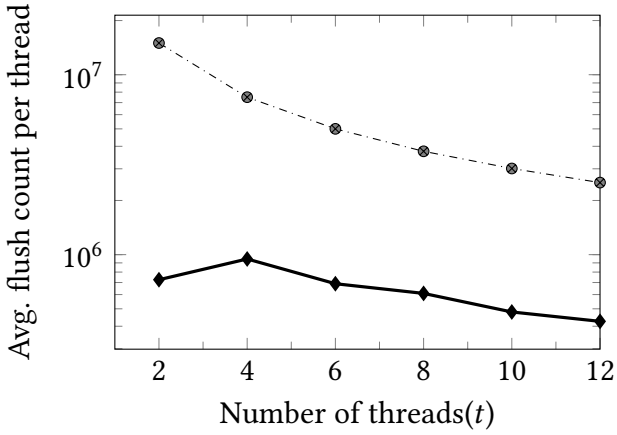
Figure 5.9: Allocation benchmarks: throughput and multi-threaded performance



(a) Larson (lower is better)



(b) Threadtest: (lower is better)



(c) Prod-con (lower is better)

Figure 5.10: Allocation benchmarks: failure consistency overhead (Makalu vs. `nvm_malloc`)

SZ MiB	MX_L $\times 10^3$	T_{ON} sec	T_{REC} ms	T_{GC} ms	$OVER_{off}$ %
200	3.39	1.01	4.18	60.98	6.45
400	6.66	2.00	6.77	133.97	7.05
600	9.84	2.97	7.36	154.94	5.46
800	13.06	3.94	11.73	276.90	7.33
1,024	16.67	5.04	13.66	349.88	7.22
2,048	32.82	10.20	28.23	699.23	7.13
3,072	49.48	15.08	37.96	1,030.57	7.08

Table 5.2: Makalu offline recovery and GC performance. SZ = size of allocated heap before crash; MX_L = longest pointer chain explored during GC; T_{ON} = online execution time before the crash; T_{REC} = time taken to recover allocation metadata and restart offline; T_{GC} = time consumed by offline GC; and offline overhead, $OVER_{off}$ = the ratio of time taken offline to time taken online to fill-up heap, $\frac{T_{REC}+T_{GC}}{T_{ON}} \times 100$.

Resur: The benchmark allocates SZ MiB of persistent memory in total by allocating persistent objects of random size up to half a page before the benchmark crashes abruptly using process abort. For most environments, this models an absurdly short interval between failures. With each allocation, a coin is tossed to either retain the allocated object, in which case it is made reachable from one of 512 `NVRAM` region roots (randomly selected) or deallocated immediately. The retained objects essentially form a collection of linked lists of variable length rooted at region roots. At the time of the crash roughly $SZ/2$ of the memory is reachable in the persistent heap. Following a crash, Makalu is started in offline mode. It first recovers its allocation metadata to a consistent state and then performs a parallel GC.

Results

Table 5.2 shows that the time to restart/recover metadata as well as the time to collect garbage (reported in 4^{th} and 5^{th} columns respectively) are quite small and grow modestly with the total size of the heap. Offline overhead data in the 6^{th} column shows that the total time spent in offline recovery and garbage collection remains a small and somewhat

constant fraction of the time spent in online allocation (3^{rd} column) even as the total size of the allocated memory (1^{st} column) and the maximum length of the pointer chain in the heap (2^{nd} column) grows.

This result greatly enhances the possibility of Makalu being profitably used in all but most peculiar cases where applications have very high rates of failure. In such cases, persistent applications may have more pressing problems than efficient memory allocation.

5.13.3 Comparison with NVMPL Default Allocators

In this section, the performance of Makalu is compared to that of the existing NVMPLs' default allocators. The default allocator in Mnemosyne is built upon Hoard [16, 38].

The possibility of in-place persistence of data became real only recently with the promise of NVRAM in the near future. As such, to the best of authors' knowledge, the earliest NVMPL work [16, 17] is barely half a decade old and still immature. Our work fills in a prior omission by providing a leak-free memory allocator with programmer friendly interface. It aims to promote the development of new NVRAM applications and a more programmable NVMPL, but it still somewhat suffers from the status quo of not having standard real NVRAM application benchmarks. Given the circumstances, the following three applications, which could benefit from in-place persistence in the future, were used. Furthermore, other transient memory management papers [38, 49] have often made use of some of these benchmarks.

Benchmarks

Barnes-Hutt: This benchmark is a multi-threaded N-body problem solver [50]. The initial set of 100,000 particle positions and forces are stored in NVRAM, as are the resultant particle positions and forces after each time-step.

N-queens: It is a lock-based, multi-threaded implementation of a recursive search algorithm for finding a solution to the n-queens problem [51]. It uses a pool of workers and work queues. Both work queues, as well as units of work, are allocated in NVRAM. A

unit of work is essentially a search frontier to be further explored. Each worker removes a unit of work from the queue and pushes new work generated back to the queue. The benchmark explored the solution for the 16-queens problem using a variable number of worker threads. Each worker has to acquire a lock to add or remove work from the queue.
Cholesky: It is a multi-threaded, tile-based algorithm for decomposing a dense matrix into a lower triangular matrix and its conjugate transpose [52]. Using a variable number of threads, the benchmark allocated a 1000×1000 input matrix in `NVRAM`, decomposed it using tile size 4×4 and stored results in `NVRAM` as well.

The same version of the Makalu allocator was integrated with the Mnemosyne and the Atlas library. The Makalu-integrated Mnemosyne and Atlas shall be referred to as ***mne-mak*** and ***atlas-mak***, whereas versions with default allocators shall be referred to as ***mne-def*** and ***atlas-def*** respectively. Mnemosyne requires special Linux kernel support and compiler support. Therefore, Mnemosyne was compiled using Intel compiler prototype edition 3 with -O2 optimization and results were obtained on a machine (same architectural specification as above) running Centos 2.6.32 Linux distro.

Results

Figure 5.12 compares the default allocators in each `NVMPL` to Makalu using speedups obtained on the above benchmarks. Since Atlas and Mnemosyne data are collected on two different systems, they should not be compared to each other. Furthermore, Atlas and Makalu speedups are calculated based on the single-threaded performance of their respective default allocators on above benchmarks on their respective machines. Table 5.3 summarizes these base values. In all three benchmarks, the Makalu-integrated version of `NVMPL` outperforms the default version by orders of magnitude. The result also demonstrates the easy interoperability of Makalu with more than a single `NVMPL`. In order to isolate and compare the allocation cost, the evaluation process was only concerned with avoiding inconsistencies in heap metadata for these sets of results. Maintaining the consistency of the user data stored in persistent heap itself was ignored.

	Barnes-Hutt *sec*	N-queens *sec*	Cholesky *sec*
atlas-def	16.44	23.72	19.67
mne-def	18.10	19.69	13.98

Table 5.3: Single-threaded base performance for Atlas and Mnemosyne default allocators.

For one such NVMPL, namely Atlas, more extended results for above benchmarks with full failure consistency mechanisms enabled are presented in figure 5.13. For this set of results, the full instrumentation of code was enabled using Atlas' LLVM compiler [19] for ensuring failure consistency of user data, in addition to guaranteeing the absence of memory leaks and consistency of heap metadata in case of a failure.

Results in figure 5.13 demonstrate that turning on full failure consistency support in Atlas hides some of the gains in allocation speed (that could be observed in figure 5.12) from using Makalu. Nevertheless, Makalu yields superior performance to the default Atlas allocator. Barnes-Hutt and Cholesky use barriers for synchronization among threads whereas N-queens uses pthread mutexes. Results in figure 5.13 show that Makalu-integrated Atlas yields superior performance for both lock- and non-lock-based code.

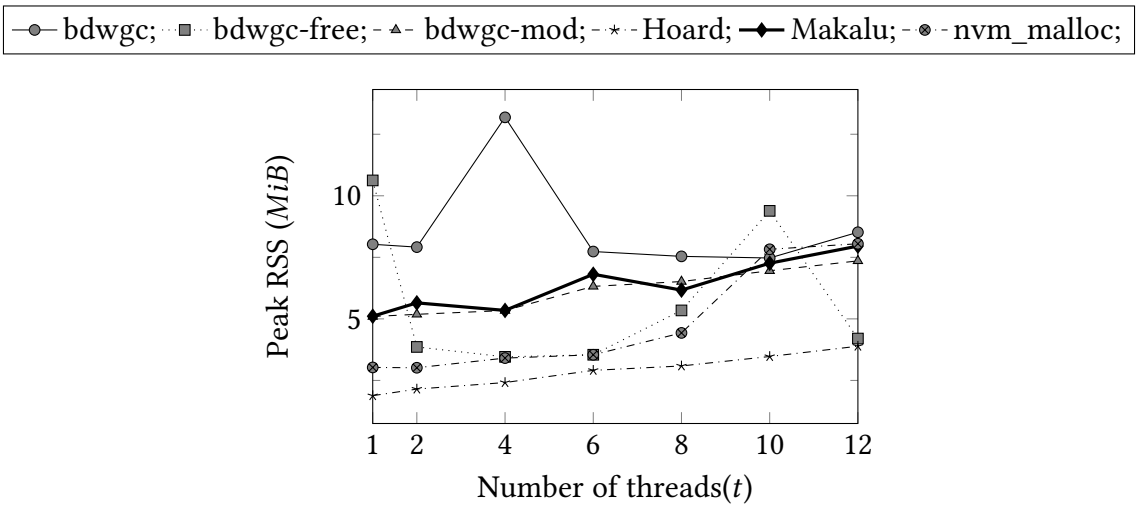
	Barnes-Hutt *sec*	N-queens *sec*	Cholesky *sec*
atlas-def	63.59	59.63	66.16

Table 5.4: Single-threaded base performance for the Atlas' default allocator (full failure consistency enabled).

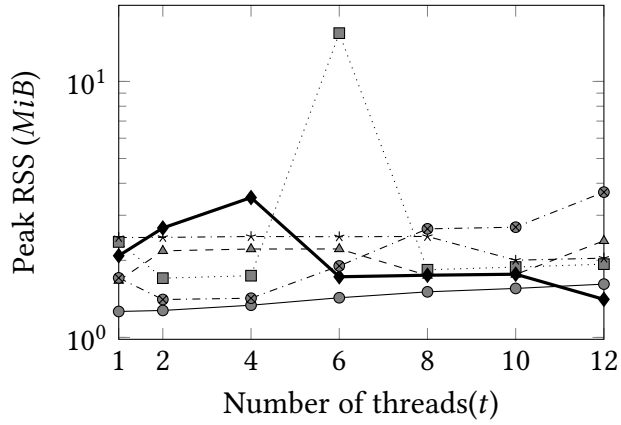
This work have shown that it is possible to build a memory allocator for NVRAM that maintains the standard malloc()/free() programming model, correctly ensures persistence of metadata, and interoperates with multiple NVRAM persistence libraries. Surprisingly this is possible at a cost comparable to transient memory allocators.

The crucial observation is that by relying on offline garbage collection during failure recovery, the per allocation persistence overhead is greatly reduced. A typical small object

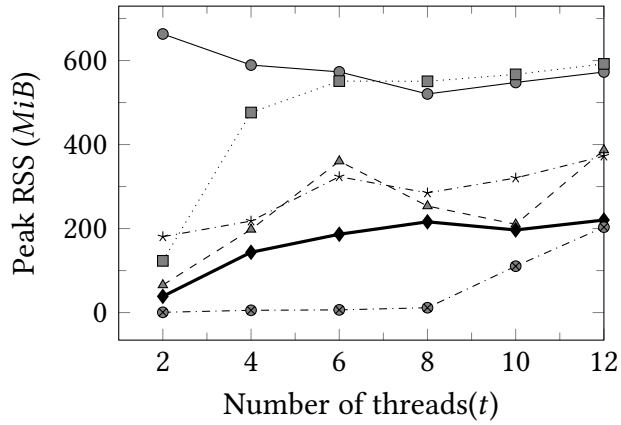
persistent allocation does not need to flush any data to persistent memory since all relevant metadata can effectively be reconstructed from the object graph during recovery.



(a) Larson: Peak RSS vs. thread count (lower is better)

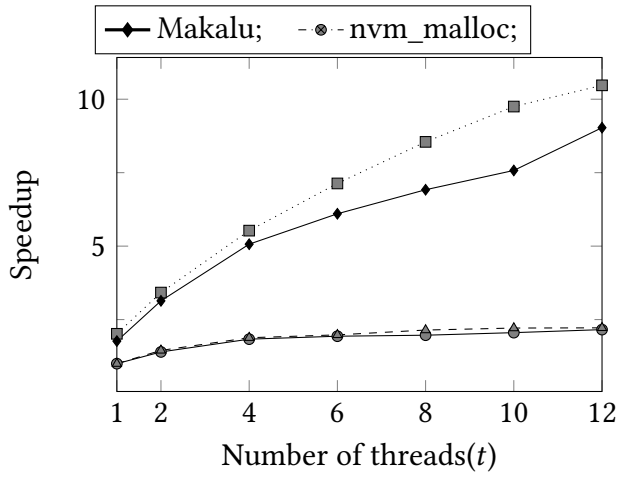


(b) Threadtest: Peak RSS vs. thread count (lower is better)

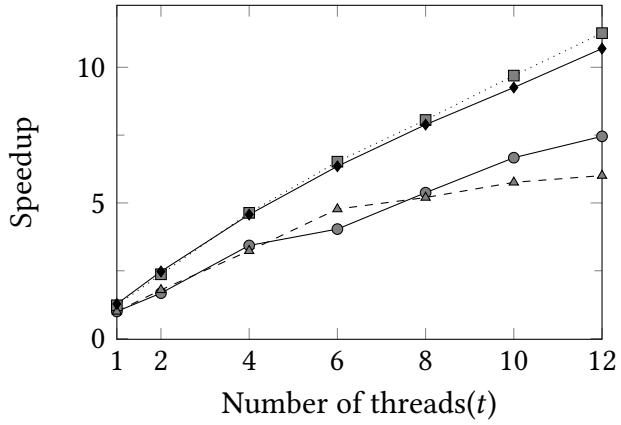


(c) Prod-con: Peak RSS vs. thread count (lower is better)

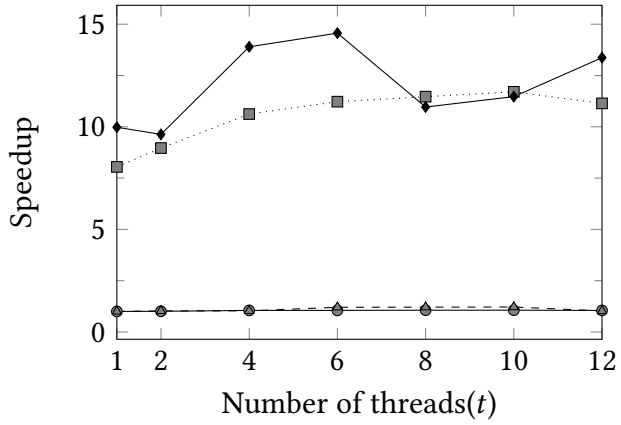
Figure 5.11: Allocation benchmarks: peak memory consumption



(a) Barnes-Hutt (higher is better)

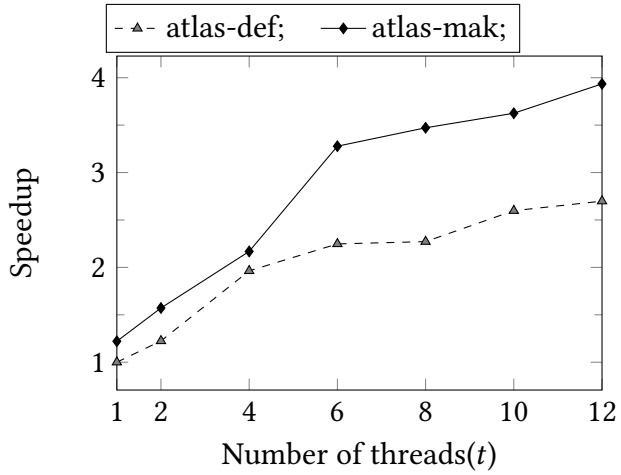


(b) N-queens (higher is better)

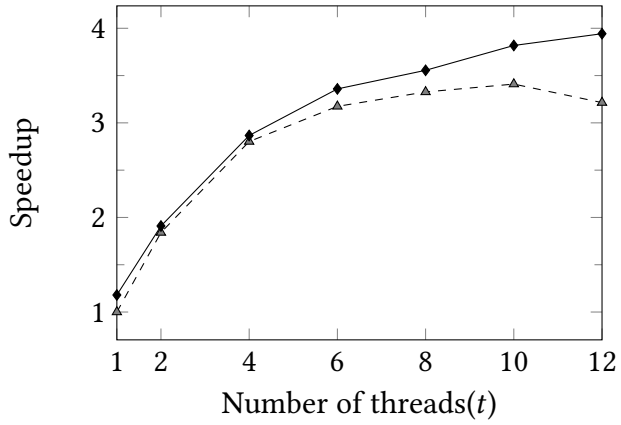


(c) Cholesky (higher is better)

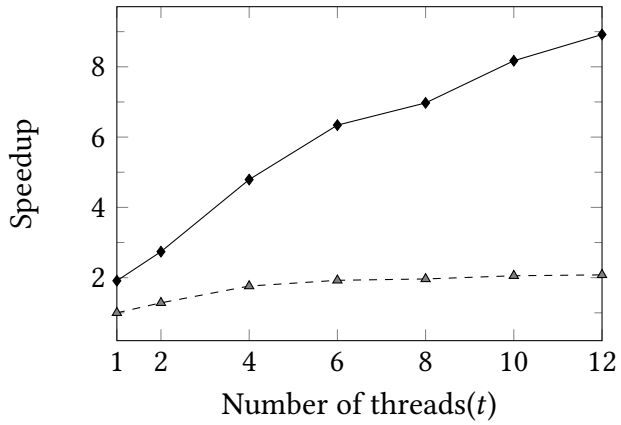
Figure 5.12: Comparison of Makalu with `NVMPLs`' default allocators. Speedup is computed w.r.t. single threaded performance of respective `NVMPLs`' default allocators. Refer to table 5.3 for single-threaded base timing values.



(a) Barnes-Hutt (higher is better)



(b) N-queens (higher is better)



(c) Cholesky (higher is better)

Figure 5.13: Comparison of Makalu with the Atlas' default allocator: full failure consistency enabled. Speedup computed w.r.t. single threaded performance of Atlas default allocator. Refer to table 5.4 for the base timing values.

Chapter 6

Relative Addressing and Precise Garbage Collection of Persistent Heaps Using Persistent Types

Persistent data stored in NVRAM need to be accessible across process and system restarts. For general-purpose programming, NVRAM programming libraries (NVMPLs) enable persistent data to be stored in an NVRAM region. An NVRAM region is a named container. Similar to a mmap-ed file, the NVRAM region can be mapped to a process' virtual address space such that the persistent data stored in the region can be accessed using CPU load and store instructions. It is possible to store absolute virtual addresses within the NVRAM region when storing data structures such as a linked-list but doing so poses several problems. An NVRAM region that stores absolute addresses from a previous execution has to be mapped by each subsequent process in each subsequent execution to the same starting address in order for the absolute addresses stored in NVRAM, which is simply not practical. This approach may also cause address range collision in different scenarios and subsequent failure in mapping an NVRAM region into a process' own address space. Many NVMPLs, such as Mnemosyne and Atlas, continue to use this approach [16, 19] in their initial prototypes.

A superior approach to storing absolute addresses within an NVRAM region is to store all persistent addresses within an NVRAM region as offsets from the start of such a region. In other words, persistent addresses stored in an NVRAM region are relative addresses. Emerging programming systems/libraries such as NVL-C [53] use this approach. For such programming systems, we need an allocator that stores its own persistent metadata as relative addresses so that its own metadata is meaningful across restarts. We need an allocator that can also handle user allocated addresses in a persistent heap as relative addresses. Makalu described in chapter 5 is not suited for this purpose since it uses

absolute addresses for its metadata and can only handle absolute addresses in the heap. Furthermore, conservative pointer analysis for finding reachable objects, as done in Makalu, produces suboptimal results with relative addresses. A conservative pointer analysis may produce a larger number of false positives when each integer that appears to be a relative address is converted into an absolute pointer. This issue arises more frequently with relative addressing because the values of relative addresses are small and comparable to frequently stored integer values. Consider two integers stored in persistent memory, one a relative address and other an integer value. Both may appear as a valid pointer in heap after converting integer values to absolute addresses by adding the appropriate starting address of the NVRAM region. Therefore, we also need a precise pointer analysis for the offline garbage collection described in chapter 5 to enable it to work properly with relative addresses. It is profitable to consider precise garbage collection for `NVRAM` programming because programming libraries such as NVL-C already enforce type safety at both compile time and runtime (e.g. restrictions against non-volatile to volatile pointer conversions). A precise garbage collection can leverage such type information maintained by a programming library like NVL-C to precisely locate pointers in a heap object. However, using relative addresses is not free. Each time a persistent address stored in `NVRAM` needs to be used, the relative address has to be converted into an absolute address, requiring pointer arithmetic.

In this chapter, we discuss an extension to Makalu, called ***Makalu-rel***, that allows it to use relative addresses for its persistent metadata. Furthermore, we describe the design of a precise offline garbage collection for a persistent heap. We also present some runtime techniques to reduce the cost translating persistent relative addresses within the allocator. Similar to Makalu, we expect Makalu-rel to be used along with programming libraries such as NVL-C and we provide a clean interface for programming libraries and programmers to interact with it. Our results show that using relative addressing has advantages, and it comes at some nominal cost. Our results also confirm that the performance of precise garbage collection is comparable to that of the conservative approach taken by Makalu.

6.1 Contributions

This work makes the following contributions:

- a leak-free, persistent allocator that allows de-/allocation of typed persistent objects and is interoperable with any programming library that supports relative addressing and typed allocation
- a precise parallel mark-and-sweep offline garbage collection that co-ordinates with the programming library through a clean interface to resolve types and perform precise pointer identification
- runtime techniques to reduce the cost of translating the relative addresses for persistence
- evaluation of the cost of relative addressing and precise garbage collection in a relatively addressed heap

6.2 Background

6.2.1 Architectural Assumptions

Makalu-rel makes architectural assumptions similar to Makalu (see section 5.2.1).

6.2.2 Programming Assumptions

Makalu-rel makes similar assumptions as Makalu (see section 5.2.2) unless otherwise noted in this section. Similar to Makalu, we expect Makalu-rel to be used along with an `NVMPL`. We expect a user program to provide Makaku-rel with a type identifier along with the size of allocation for each allocation request. By providing the type identifier associated with an allocated object, we expect Makalu-rel to be able to obtain the following type information from the programming library offline.

- the number of pointers in the object

- an array of all offsets within the object (computed from the beginning of object) where pointers are stored

Similar to Makalu, Makalu-rel also assumes that all of the useful data stored in a heap are reachable from the heap roots (stored as relative addresses) when the user data is in consistent state offline.

6.2.3 Terminology

See section 5.2.3.

6.3 Challenges

Apart from the challenges discussed in section 5.3, the design of a persistent memory allocator with relative addressing and precise offline garbage collection has to address the following additional challenges.

Resolving Persistent Type Information A persistent heap may store objects that have been allocated over several execution cycles across several restarts. If a precise offline garbage collection is to be performed over such a persistent heap, the type description corresponding to type identifier for each such allocated object in the heap also need to be tracked in a fail-safe manner across all execution cycles and restarts. However, for wider interoperability, the mechanism by which a collector resolves type with the programming library needs to be non-specific to a particular programming library.

Address Translation Overhead Translating relative addresses to absolute addresses require performing pointer arithmetics and if this occurs within a frequently executed code, the effect will be observable in terms of increased overhead. An allocator, as well as the collector, has to address this challenges when it comes to accessing its own metadata.

6.4 Overview of Our Approach

Makalu-rel is based on Makalu allocator and garbage collector described in chapter 5. We preserve Makalu's approach to explicit online de-/allocation, followed by offline garbage collection. However, the offline garbage collection algorithm is different in Makalu-rel compared to Makalu's conservative garbage collection. Both use the parallel mark-and-sweep approach. However, Makalu-rel uses type information to obtain precise pointer locations in heap objects. Makalu requires no coordination with the NVRAM programming library (`NVMPL`) during the offline phase for garbage collection. In contrast, Makalu-rel has to coordinate with the `NVMPL` to obtain detailed type information (e.g. pointer offsets, number of pointers) for each object being marked during offline.

In order to allocate memory, Makalu-rel also requires a 32-bit type identifier along with the size of the allocated object, whereas Makalu only required the size. This type identifier for the object is persisted and used later by the offline collector to get the detailed type information about that object from the `NVMPL`.

Internally, unlike Makalu, all persistent metadata in Makalu-rel is stored using relative addresses and thus require translation from relative to absolute address during any lookup of metadata. We implemented some runtime optimizations to reduce the translation overhead, e.g. caching the most recently used translated metadata addresses.

6.4.1 Makalu-rel APIs

Makalu-rel's API remains largely unchanged comparable to Makalu's (see section 5.4.3). Most of the changes are to the `NVMPL` facing interface which we will discuss later in section 6.11.

The only change to the programmer facing interface compared to Makalu is the allocation interface. Makalu only required the size to be supplied as an argument to `Mak_malloc`, whereas Makalu-rel requires both the size as well as a 32-bit type identifier to be supplied as arguments, as shown below:

```
MAK_malloc(size_t sz, unsigned int type);
```

The deallocation interface remains the same.

6.5 Internal Structures and Layouts

The transient structures that are described in section 5.5 remains unchanged in Makalu-rel. However, the persistent structures in Makalu-rel use relative addressing as described below.

6.5.1 Persistent Block Header

Recall from section 5.5.1 that one of the fields in the block header, `hb_block`, stores the starting address of the block for which the header is assigned. This information is used often throughout Makalu-rel and is crucial in rebuilding internal metadata such as the header map (see section 5.5.2) and other freelists after a failure. Hence the heap block starting address is stored as a relative address in Makalu-rel. Figure 6.1 shows the code snippet for assigning a header to the block, where the block address is being stored as a relative address at line 4.

```
1: struct hdr* MAK_install_header(struct hblk *h) {
2:   struct hblk *h_rel = (struct hblk *) REL_ADDR(h);
3:   hdr * result = alloc_hdr();
4:   MAK_STORE_NVM(result->hb_block, h_rel);
5:   ...
6:   //add hdr to the header map
7:   SET_HDR(h_rel, (hdr*) REL_ADDR(result));
8:   return result;
9: }
```

Figure 6.1: Installing a new block header in Makalu-rel

6.5.2 Persistent Header Map

Recall from section 5.5.2 that the header map enables Makalu to map a given object to the block header for the block to which the object belongs. The address of the block, the address of the corresponding header in this map, as well as all its internal structure are

stored as relative addresses in Makalu-rel. The new mapping for the recently allocated header is set in the header map at line 7 in figure 6.1. Note that header address is converted to a relative address before being stored in the map.

6.5.3 Persistent Base Metadata

Recall from section 5.5.5 that Makalu stores the following core metadata in `NVRAM`:

- `NVRAM` region base (`rgn_base`) and max heap (`rgn_curr`) address
- Log space starting address (`md_log_space_start`)
- Current log version (`md_log_version`) (see section 5.8)
- List of header spaces
- Address of the header map space
- Persistent root space start address

In Makalu-rel, all the addresses in these fields such as log space starting address, root space start address, the header space addresses etc, are stored as relative addresses. The frequently used relative addresses (e.g. log space starting address) are translated into absolute addresses when Makalu-rel re-/starts and stored in transient variables. Others are translated as needed.

6.6 Reducing Address Translation Overhead

Recall that a block header stores information such as allocation size within the block, mark bits to indicate which objects are allocated or free, etc. Hence, a block header contains frequently used persistent metadata and is a major source of persistent address translation overhead. Furthermore, looking up the header also requires translating addresses for the components of header map structure. Therefore, we introduced a header cache to store the absolute addresses of a fixed number of most recently looked up headers. This cache maps a given object to the correct header for the block to which the object belongs, if the header exists in the cache. Each thread local heap maintains its own header cache. A

similar header cache is also maintained at the global level. Looking up the header in the header cache incurs low overhead as opposed to translating addresses in the header map, traversing the map and finally translating the header address when found.

Figure 6.2 shows the header lookup in Makalu-rel where it first consults the header cache. Makalu-rel computes the corresponding block for the given pointer in line 3 and computes the corresponding entry in the header cache in line 4. It then checks whether the header cache entry corresponds to the given block 'h' (corresponding to 'p') in line 55. If it does, then it is a cache hit, but before returning the header, it further checks whether the header in the cache entry is still assigned to the same block as h in line 6. We need this second check to ensure that the header to block assignment has not changed (e.g. by another thread) since we last cached this result. This check maintains the header entry coherence among participating local header caches. If there is a cache miss, we look up the header from the header map, obtain the translated address, and update the header cache (lines 11-13).

```
1:  MAK_INNER hdr* MAK_get_hdr(void* p){
2:    hdr* hhdr;
3:    hblk* h = HBLKPTR(p); //compute p's block
4:    hc_e* hce = HCE(h, hdr_cache, hc_sz); //compute hdr cache entry
5:    if (HCE_VALID_FOR(hce, h)) { //check cache entry is valid
6:      /* cache hit */
7:      hhdr = hce->hce_hdr
8:      if (hhdr->hb_block == REL_ADDR(h)) //check header is valid
9:        return hhdr;
10:   }
11:   /* cache miss, update cache */
12:   hhdr = HDR(p); //look up header from hdr map
13:   UPDATE_HC(hhdr, p, hce);
14:   return hhdr;
15: }
```

Figure 6.2: Looking up a header from header cache

6.7 Allocation

Allocation requests of all sizes are padded with 4 extra bytes to store the type identifier supplied by the user program as an additional argument. Makalu-rel then follows the usual algorithm described in section 5.6 to allocate an object padded with these extra bytes. Before returning the allocated object, it stores the user-supplied type identifier at the start of the allocated object, and flushes the type identifier to `NVRAM` synchronously. It then returns the starting address of the allocated address just past the type identifier. Calls to Makalu-rel's allocation routine return absolute addresses for convenience. When free objects in a block are added to Makalu-rel's transient freelist (see section 5.6), it computes their absolute addresses. Allocation from the freelist thus neither requires address translation nor writes to persistent metadata (see section 5.6. We expect the user program to only store relative addresses in the heap. Absolute addresses can be converted into relative addresses before storing them in the heap using Makalu-rel's helper methods.

6.8 Deallocation

Makalu expects user programs to provide an absolute address for deallocation. The provided pointer is decremented to the beginning of the actual allocation (i.e. at the beginning of the stored type identifier). The rest of the deallocation follows the steps described in section 5.7.

6.9 Failure Consistency Guarantees

Makalu-rel uses technique similar to the one described in section 5.8 to provide ACID guarantees while updating core metadata. Makalu-rel however uses relative addresses to create persisent undo log entries in persistent log space, such that, after a failure, such logs can be applied to persistent metadata mapped to possibly different absolute addresses. Figure 6.3 shows code for creating an undo log entry using a relative address (line 2) for an integer data type.

```
1: void create_int_log_entry(int* addr, int val){
2:   curr_log_e->addr = (int*) REL_ADDR(addr);
3:   curr_log_e->val.int_val = val;
4:   curr_log_e->type = INTEGER;
5:   curr_log_e->version = MAK_persistent_log_version;
6:   FLUSH_SYNC(curr_log_e);
7:   curr_log_e++;
8: }
```

Figure 6.3: Creating undo log using relative address

Likewise, during recovery after a crash, these log entries are replayed by translating the relative addresses to absolute addresses as shown in figure 6.4.

```
1:  void MAK_recover_metadata(){
2:    ...
3:    while ((char*)e >= MAK_persistent_log_start) {
4:      addr = (void*) ABS_ADDR(e->addr);
5:      switch (e->type){
6:        ...
7:        case INTEGER:
8:          *((int*) addr) = e->val.int_val;
9:          FLUSH_FAS(addr, sizeof(int));
10:         break;
11:       ...
12:     }
13:     e -= 1
14:   }
15: }
```

Figure 6.4: Replay of Undo logs with relative addresses

The eventual visibility guarantees for auxiliary metadata works the same way as described in section 5.8.1.

6.10 Offline Recovery and Garbage Collection

Offline recovery after a failure proceeds similar to Makalu by first recovering the core metadata to a consistent state using undo logs, then purging and building a new header map by scanning the header spaces. The recovery logic in Makalu-res has to translate addresses before applying undo logs and likewise before scanning header spaces.

The garbage collection (GC) in Makalu-rel case uses parallel offline mark phase with precise pointer identification, followed by on-demand onlines. The sweep over marked blocks happens lazily (incrementally) online on as needed basis when freelists need to be refilled.

6.10.1 Precise Parallel Mark Algorithm

Makalu-rel uses one or more threads to discover all persistent objects in the heap reachable from persistent roots. Each mark thread allocates a fixed-size transient mark stack. Makalu-rel also allocates a larger global mark stack. The structure of a mark stack entry is shown in figure 6.5. The `current` field in the entry stores the starting address of the heap object (as returned by a call to `MAK_malloc`). The array of `offsets` contains the location of the pointers in the object to be marked relative to current, and the field `work` signifies the number of pointers that this marking thread needs to mark.

```
struct mse {
   char* current;
   unsigned int* offsets;
   unsigned int work;
};
```

Figure 6.5: Mark Stack Entry for Offline GC

Each marking thread can offload work from the global stack into their own local stack if their mark stack becomes empty. Likewise, they can also offload objects to be marked from their own stack into the global stack if their stack overflows. The global stack is

resized on overflow.

The offline mark phase starts by pushing all the persistent roots stored as relative addresses into a mark stack. For each object referenced by a pointer (including persistent roots), Makalu-rel performs the following steps:

1. Mark the object. Makalu-rels first obtains the header for the block to which the object belongs. In the header, it then sets the correspoding bit for the object to indicate that it is reachable.
2. Get type identifier for the object. Recall that the type indentifier for an object is always stored at the beginning of the allocated object (see 6.7).
3. Obtain type information from `NVMPL`. Using the type identifier, Makalu-rel obtains information such as the number of pointers in the object and the pointer offsets.
4. Create a mark stack entry. Using information from 3, Makalu-rel populates `offsets` and `work` fields in the entry and adds it to the mark stack.

For each mark stack entry in the mark stack, each marking thread performs one of the following two actions:

1. Objects larger than threshold If the number of pointers to be analyzed in the object is larger than a threshold (default value 512), it carves off the threshold number of pointers to mark, and pushes rest of the work back on the stack (for distributing work possibly among other marker threads). The work field and the starting address of offset array are updated accordingly before pushing the entry back to the stack.

2. Objects smaller or equal to threshold If the number of pointers to be analyzed in the object is within a threshold, the marking thread uses steps 1-4 listed above to analyze each pointer in the object that it is supposed to analyze (as indicated by the work field in mark stack entry).

The empty mark stack entry marks the end of the mark phase. Similar to Makalu, all mark bits and other auxiliary allocation metadata are made visible in `NVRAM` before Makalu

shuts down gracefully in an offline mode such that the reclaim list and other freelists can be created reliably when later restarting online.

6.11 Interaction with NVRAM programming libraries

The API for `NVMPL` to interact with Makalu-rel remains largely unchanged. We expect Makalu-rel to simply work out-of-the-box with programming libraries such as NVL-C as long as the `NVMPL` has a mechanism to reliably keep track of type information and provide Makalu-rel with type descriptions on request. We describe below appropriate API calls for different phases of Makalu-rel.

6.11.1 Online Start

To set up a heap in a new `NVRAM` region for the first time, we expect `NVMPL` to call `MAK_start` and pass in as argument the absolute starting address of the heap, and the maximum size to which the heap can expand. Note the the starting address needs to be page-aligned. The maximum size is stored as persistent metadata.

6.11.2 Offline Restart and Recovery

Following a crash, we expect `NVMPL` to start the Makalu recovery process by calling `MAK_restart` and passing in a new absolute starting address of the heap depending on the address to which the `NVRAM` region is mapped. Once the recovery finishes, we expect `NVMPL` to call `MAK_collect`. We expect the `NVMPL` to pass in as an argument a callback function with the following signature:

```
int MAK_persistent_type_descr( unsigned int type,
  unsigned int** offset_vector, unsigned int* count);
```

This callback function is expected to take the type identifier for an object as the first argument and return the corresponding array of offsets to pointer locations within an

object of such a type and a pointer to the number of pointers within the object of such a type (i.e. the length of offset_vector) in second and third argument fields respectively. Makalu-rel's precise garbage collection utilizes this callback to obtain type details such as the the number and the locations of pointers in the object of that type as described in section 6.10.1.

6.11.3 Online restart

Once the heap has been setup using `MAK_restart`, we expect the `NVMPL` to call `MAK_restart` to resume online allocation after a recovery or a previous graceful shutdown. The method call takes absolute starting address of the heap as an argument.

In all states, we handle failure in Makalu-rel in a manner similar to Makalu.

6.12 Evaluation

Although `NVRAM` is getting closer to becoming widely available [44, 45], it is not currently a simple matter to create an `NVRAM` testbed. Hence, Linux *tmpfs* [25] was utilized to simulate NVRAM during the collection of results. As files in *tmpfs* are only backed by DRAM, a process crash and restart mechanism was used to test Makalu's crash resilience and restart logic. In order to measure failure consistency overhead, full failure consistency mechanisms such as cache line flushes and memory fences were enabled, as it would have been done for actual NVRAM systems.

All the code used in this evaluation was compiled using the GNU gcc/c++ compiler version 5.4.0 at optimization level `"-O2"`. All experiments were run on a 64-bit Ubuntu machine (Linux kernel version 4.10.0-42) that has 8 GiB of RAM, 4 cores Intel Core i5-750 CPU @ 2.67 GHz. All results collected were averaged over 6 runs.

We used CLFLUSH/MFENCE instead of the new CLWB/SFENCE instructions to obtain an estimate of persistence overhead. We explained the reason for doing so and how we expect the performance of our allocator to improve with new ISA in section 5.13. A similar reasoning holds for this evaluation as well. We compared the two versions of Makalu in

this section:

mak-abs: The version of Makalu described in chapter 5 that uses absolute addresses within a peristent heap and metadata and utilizes conservative garbage collection.

mak-rel: The version of Makalu described in this chapter.

6.12.1 Allocation Performance Comparisons

We compared the allocation speed, throughput, and multi-threaded performance of Makalu-rel with Makalu to understand the impact of relative addressing and tracking and storing type identifiers next to allocated objects.

Allocation Benchmarks

We used the same commonly used allocation benchmarks that we described in section 5.13.1:

Larson The benchmark was configured to run for 10 seconds, with t threads where each thread runs for 10^4 rounds, allocating and deallocating 10^3 64-byte

Threadtest For t threads, the benchmark was run such that each thread performed 10^4 rounds of de-/allocation, and $\frac{10^5}{t}$ de-/allocations in each round.

Prod-con Each pair of producer-consumer threads de-/allocated $\frac{2\times10^7}{t}$ objects that are 64 bytes in size.

Results

Figure 6.6 shows multithreaded performance for the above allocation benchmarks. We can see that mak-abs yields a better performance results compared to makau-rel. Makalu-rel has a number of sources of extra overhead. First, the extra overhead comes from address

translation. Second, the extra overhead also comes from having to track type information, store it in allocated object and flush it each time the object is allocated. This flushing overhead is evident from results presented in figure 6.7. In each case, Makalu-rel issued higher number of flushes per thread compared to Makalu. The peak memory consumption for both versions of allocator seem to be comparable for these set of benchmarks as seen in figure 6.8.

6.12.2 Comparison Using Scientific Applications

We used the same set of scientific applications that we described in section 5.13.3. We chose these applications due to the large number of de-/allocations they perform. The goal is to assess the performance difference between Makalu and Makalu-rel using scientific applications that are closer to real-world applications.

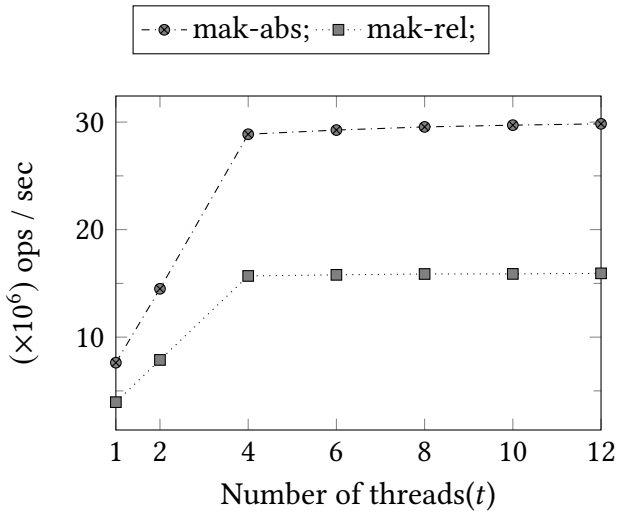
Barnes-Hutt: It uses the initial set of 100,000 particle positions and forces are stored in `NVRAM`, and stores all results in `NVRAM` as well after each time step.

N-queens: The benchmark explored the solution for the 16-queens problem using a variable number of worker threads.

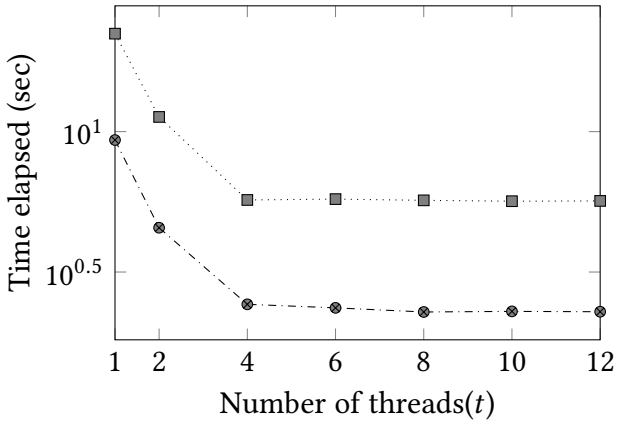
Cholesky: Using a variable number of threads, the benchmark allocated a 1000×1000 input matrix in `NVRAM`, decomposed it using tile size 4×4 and stored results in `NVRAM` as well.

Results

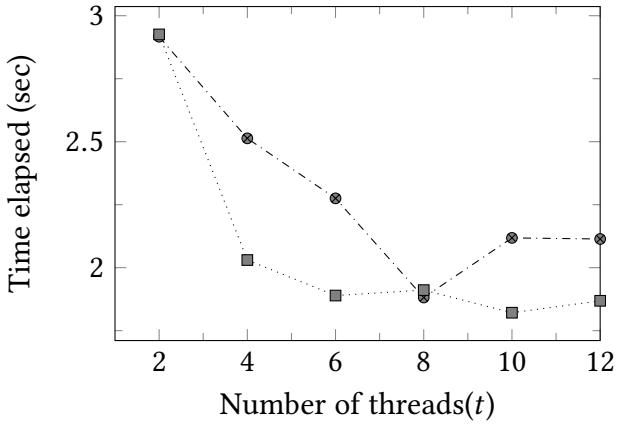
Performance numbers in figure 6.9 shows that the performance difference between Makalu-rel and Makalu is more visible in some applications than in others. The overhead of address translation appears if large number of de-/allocations are along the critical path (e.g Cholesky). In most cases, the performance is comparable between the two versions of



(a) Larson: allocation throughput (higher is better)

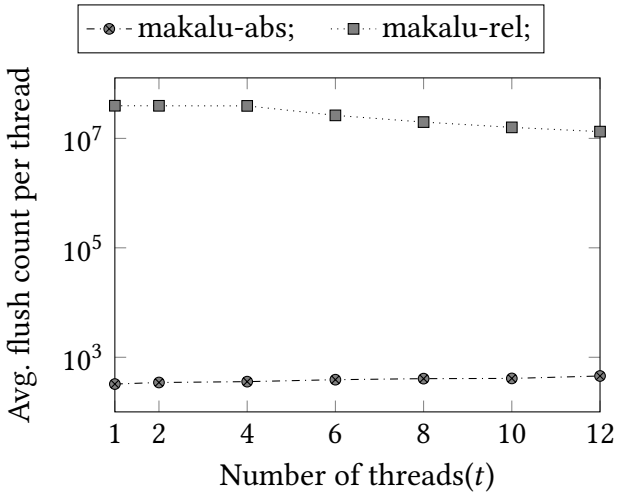


(b) Threadtest: time elapsed vs. thread count (lower is better)

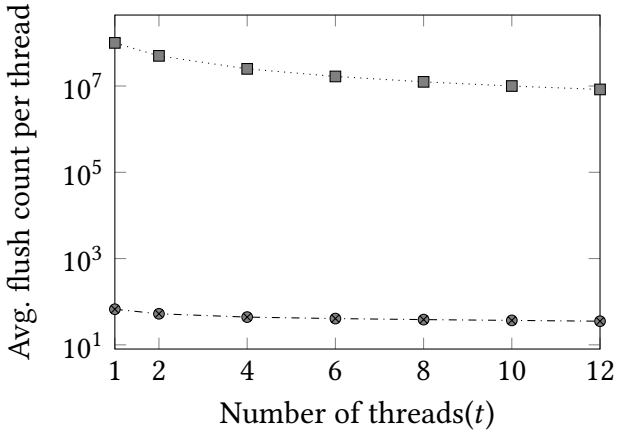


(c) Prod-con: time elapsed vs. thread count (lower is better)

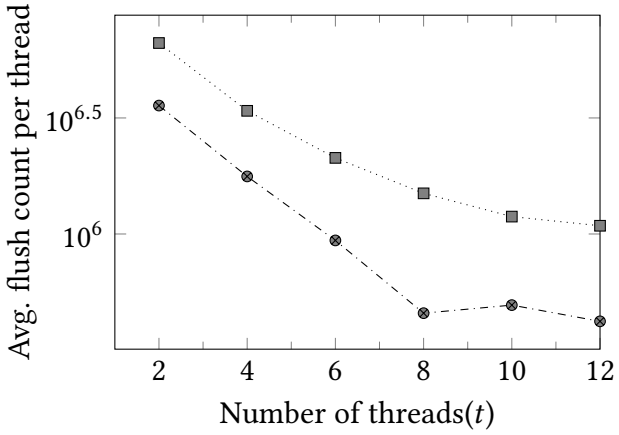
Figure 6.6: Allocation benchmarks: throughput and multi-threaded performance. Relative scalability of the two cases as a function of the number of threads (up to the number of hardware cores)



(a) Larson (lower is better)



(b) Threadtest: (lower is better)



(c) Prod-con (lower is better)

Figure 6.7: Allocation benchmarks: failure consistency overhead (Makalu vs. Makalu-rel)

allocators, with Makalu performing slightly better for the same reasons as explained in section 6.12.1.

6.12.3 Comparison: Conservative vs. Precise Garbage Collection

We compare Makalu-rel's recovery and garbage collection with Makalu's using the benchmark *Resur* described in section 5.13.2. We seek to understand how precise pointer identification compares in overhead to the conservative approach. Likewise, we also seek to understand how address translation overhead affects both recovery and garbage collection.

Results

Figure 6.10(a) compares the time required for the two versions of the allocator to recover its persistent metadata to a consistent state offline. Similar to what we observed in section 5.13.2, the recovery time grows almost linearly with heapsize. However, there is a constant difference in the overhead between Makalu-rel and Makalu. This difference is primarily due to address translation needed by Makalu-rel during the recovery.

Figure 6.10(b) compares the time needed by the Makalu-rel's precise collection with Makalu's conservative collection. In Makalu, a conservative pointer analysis is required per pointer, whereas Makalu-rel requires a callback and type description lookup. In this regard, the two versions simply exchange the source of overhead. The performance of Makalu-rel GC is in general lower than that of Makalu mostly because of the address translation needed for each heap object and metadata that need to be looked up. A mark routine constitutes a "hot code" section in the mark phase. Any perturbation such as translating addresses within the mark routine has a large impact on performance as the method is executed thousand of times during the mark phase. Therefore Makalu-rel GC is slower than Makalu by a constant factor for each heap size. Also, precise collection traverses the heap in a less cache friendly manner than conservative collection. In conservative collection, a block of object may be contiguously scanned for possible pointers, which is also not the case in precise collection.

6.13 Related Work

Many other programming frameworks such as Open Community Runtime have utilized addressing relative to the starting of the block to preserve portability and ease of use of data stored in such blocks across processes [54].

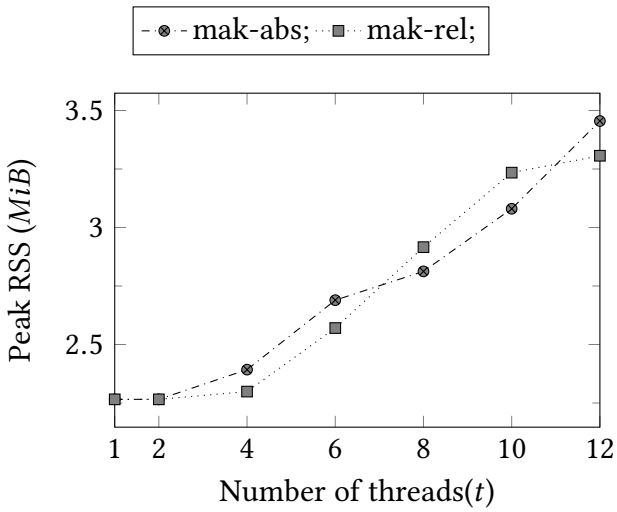
Previous work exists on enabling precise garbage collection in C/C++. For instance, Magpie [55] looks at a source-to-source compilation of C programs to achieve precise garbage collection in C. It is not clear whether Magpie handles relative addresses efficiently. It certainly is not designed to persist allocation and garbage collection states in a fail-safe manner.

Likewise, a large body of work exists on enabling parallel garbage collection [56, 57]. Our approach for parallel mark phase is adapted from [37] and the approach taken to balance the mark load among marking threads is similar to [58], although much simpler in our case.

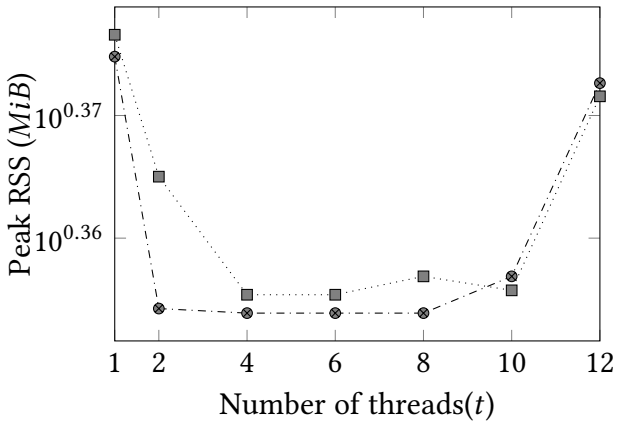
For `NVRAM` programming, NVL-C enforces certain type safety for persistent objects during compile-time and runtime [53]. It is possible to know precise pointer locations for a program compiled using NVL-C compiler. However, it only uses simple reference counting technique to collect unreachable objects, thereby being unable to collect cyclic data structures. We expect such a programming library to benefit from using Makalu-rel.

6.14 Summary

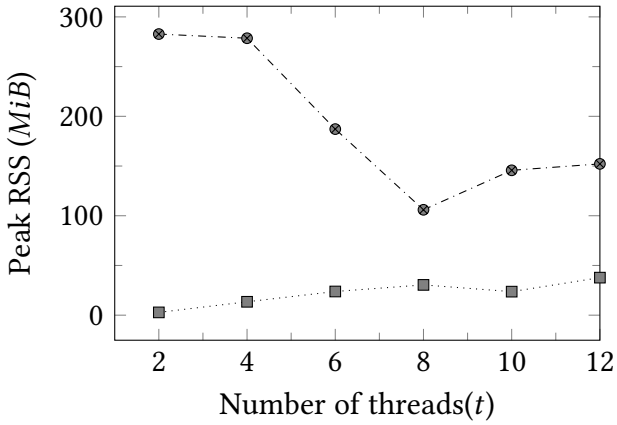
We described a new version of persistent memory allocator that uses relative addresses for its internal metadata and facilitates the use of such addresses in heap. We also described a design and interface for a precise garbage collection that uses persistent types to identify pointers in the heap precisely. Our results show that using relative addresses has many benefits and is a superior approach to storing persistent data, but doing so has performance implications for allocation speed, time to recover and collect garbage. We introduced a runtime technique to reduce the burden of address translation within heap metadata.



(a) Larson: Peak RSS vs. thread count (lower is better)

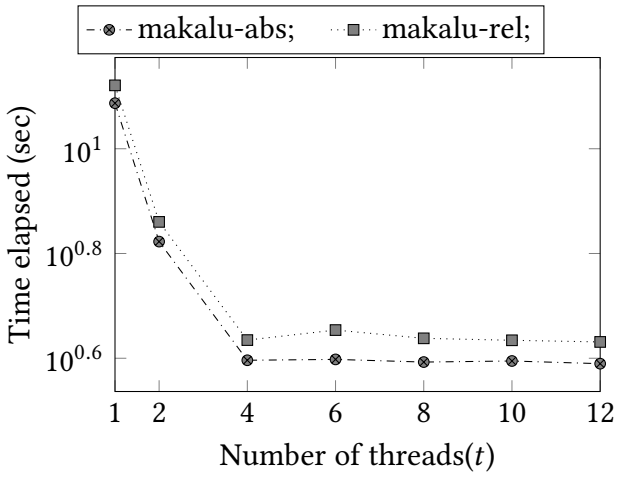


(b) Threadtest: Peak RSS vs. thread count (lower is better)

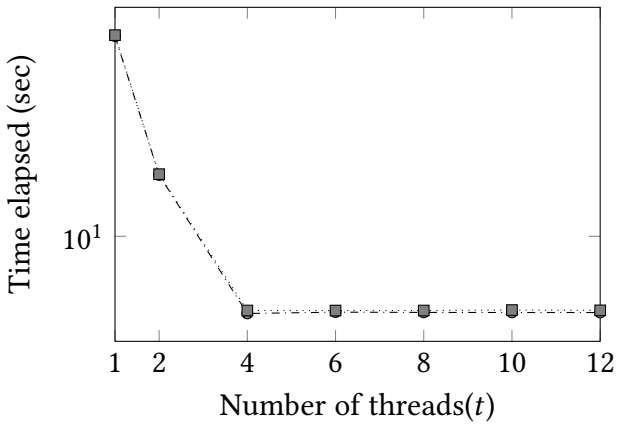


(c) Prod-con: Peak RSS vs. thread count (lower is better)

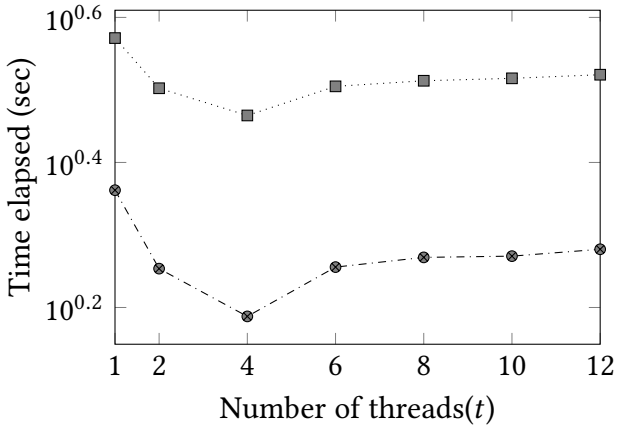
Figure 6.8: Allocation benchmarks: peak memory consumption



(a) Barnes-Hutt (lower is better)

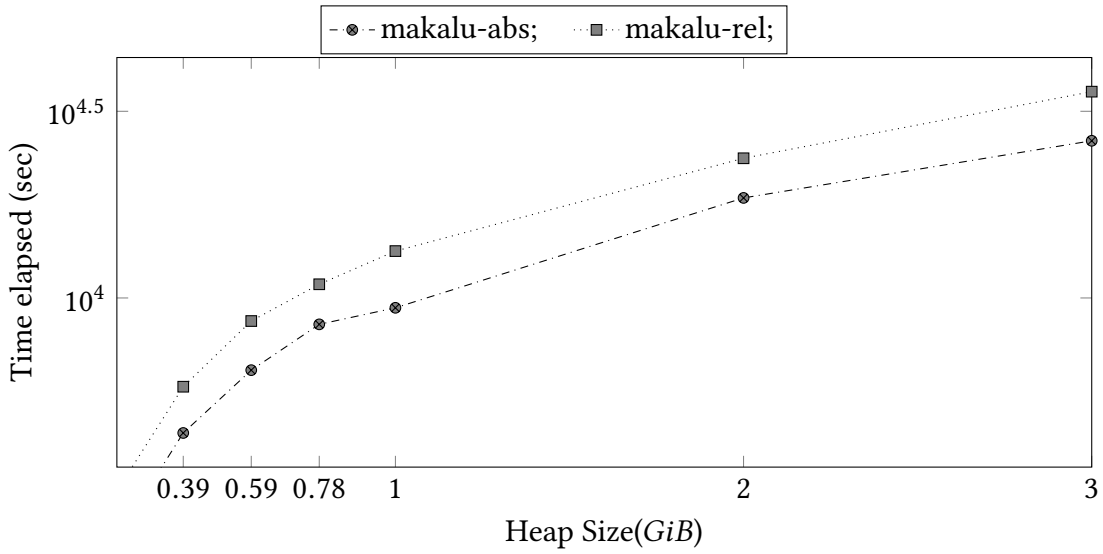


(b) N-queens(N=16): (lower is better)

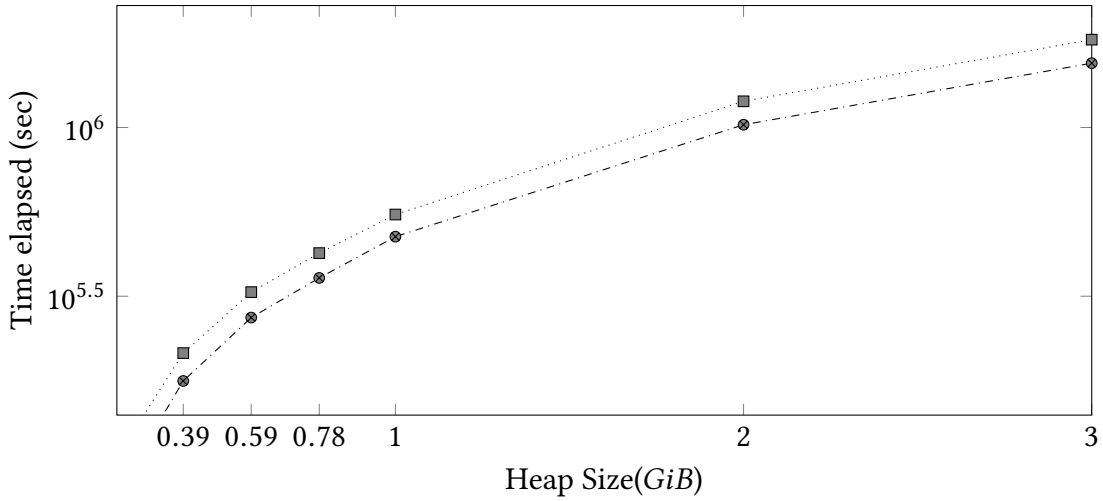


(c) Cholesky (lower is better)

Figure 6.9: Scientific applications: multi-threaded performance (Makalu vs. Makalu-rel)



(a) Time to recover metadata offline after a failure (lower is better)



(b) Time to collect garbage offline after a failure(lower is better)

Figure 6.10: Comparison of Recovery and GC time in Makalu and Makalu-rel

Chapter 7

Future Work and Conclusion

7.1 Future Work

This thesis work presented software techniques such as combining the transactional log-based approach with copy-on-write for failure consistency. An interesting line of work to pursue is to look various lock-free and non-blocking data structures for non-volatile memories, how their failure consistency can be guaranteed (either manually or automatically), and to quantify the overheads involved.

Likewise, we presented a persistent version of a commercially used key-value store. The current work on `NVRAM` programming suffers from a lack of standard benchmarks and a lack of variety in real life applications from various domains. Another avenue of work is to explore how persistence provided by `NVRAM` can benefit applications in various domains. A collection of such applications as benchmarks for future `NVRAM` programming library developers can be invaluable. Likewise, an interesting study can be made of how `NVRAM` persistence affects the design and performance of applications in trending fields such machine and deep learning.

This thesis also presented one of the first general-purpose interoperable persistent memory management frameworks with offline garbage collection. With a large persistent heap, recovery may take a significant amount of time. Hence, an interesting avenue of work is to explore incremental recovery and garbage collection. The garbage collection presented here works offline. Another interesting direction for future work could be to use online garbage collection on a persistent heap, and to perform such garbage collection incrementally. Exploring alternatives to garbage collection to avoid persistent leaks is another interesting direction for future work. Performing garbage collection requires

discovering all reachable objects which can involve a large amount of work for large persistent heaps. An alternative to garbage collection can include co-operative mechanism between the user application and the persistent memory allocator to compare and discover what is allocated yet unreachable after a failure. Such mechanisms are worthy of further exploration.

7.2 Conclusions

Emerging byte-addressable non-volatile memory technologies allow persistent data to be stored in the same format as it is manipulated. Programmers can directly allocate, store and manipulate persistent data in NVRAM using CPU load and store instructions. In architectures containing NVRAM, volatile caches and DRAM may continue to exist and may temporarily store updates to persistent data. In case of a crash, such persistent updates may become only partially visible in NVRAM leaving the persistent data in NVRAM in an inconsistent state. Hence, updating persistent data correctly with respect to failure is central to being able to take advantage of the persistence provided by NVRAM.

In this thesis work, we presented a survey of NVRAM technologies and their characteristics in Chapter 2. This chapter also presented a survey of hardware primitives such as ISA extensions that are being developed to enable NVRAM programming. Programming using low-level hardware primitives is cumbersome. Hence, we introduced a higher level persist-reuse programming model in chapter 3 to achieve failure resilient update of persistent data. In Chapter 4, we realized this persist-reuse model by combining a log-based transactional approach with a copy-on-write. Programming libraries such as Atlas, which use persistent transactional logs to achieve failure atomicity incur high overhead in terms of writing and flushing persistent logs. We showed that these overheads can be mitigated by combining log-based approaches with a copy-on-write mechanism to update persistent data. Much of the previous and current work in NVRAM programming suffers from a lack of real world applications that take advantage of persistent memory provided by NVRAM. To address this problem we also developed a real world NVRAM application and described its design and

performance characteristics in Chapter 4. The failure consistency in this application is guaranteed by combining copy-on-write with Atlas' log-based transactional semantics.

In chapter 5, we addressed persistent memory management, another important component of NVRAM programming. Prior to our work, no satisfactory NVRAM memory allocator was available for NVRAM programming. Each NVRAM programming library contained a persistent memory allocator that could not be used with another programming library. These existing allocators also put severe programming restrictions in terms of where allocations could be performed and required complex interfaces for de-/allocation. In Chapter 5 presented a careful assessment of challenges in designing a leak-free persistent memory allocator that is interoperable with several programming libraries. This chapter then presented a memory allocator design that combined offline garbage collection to address the challenges. The presented allocator, Makalu, is one of the first published leak-free and interoperable memory allocators. This chapter also introduced several techniques to reduce persistence overhead within a memory allocator and showed how a persistent memory allocator impacts programmability.

The allocator described in Chapter 5 uses absolute addresses to store its metadata. Furthermore, it expects absolute addresses to be stored in the heap for the conservative garbage collection to work properly. This is a limitation as persistent regions and heaps in NVRAM may be mapped to different addresses across restart and by different processes. For Makalu described in Chapter 5 to work, a persistent region always has to be mapped to the same address. Chapter 6 describes a modified design for Makalu that uses relative addressing for its own metatada. Furthermore, the modified Makalu handles relative addresses stored in its heap correctly for garbage collection. Conservative garbage collection does not usually work well with relative addresses and hence, this chapter also presents a parallel precise offline garbage collector design.

The work described in this thesis represents an advancement in NVRAM programming and paves the way for future advancement in this field. For instance, before the persistent allocator described in this thesis was developed, developing a new NVRAM programming

library also required developing an allocator from scratch. Now, such library developers may simply choose to use Makalu. We expect several software techniques presented in this thesis, such as combining copy-on-write with log-based approach for failure consistency, offline garbage collection to avoid persistent leaks to become the norm for `NVRAM` programming in the future.

Appendices

Appendix A

CPU Caching Policy Implications in Pre-NVRAM Architecture

As a prelude to this thesis work, we studied widely available Intel x86-64 bit architecture that pre-dated the extended ISA described in chapter 2. The study was done from the perspective of NVRAM programming and led to numerous insight such as the cost and behavior of the current cache management instruction and the need for a better ones, CPU caching options, behavior and its implication to NVRAM programming. This chapter presents this study and insights gained.

A.1 Caching Policy Choices: Write Back vs. Write-Through

It turns out that constraining the order in which writes become visible on `NVRAM` is at the core of maintaining consistency. Consider, for example, a common programming idiom where a persistent memory location N is allocated, initialized, and published by assigning the allocated address to a global persistent pointer p. If the assignment to the global pointer becomes visible in NVRAM before the initialization (presumably because the latter is cached and has not made its way to NVRAM)[1] and the program crashes at that very point, a post-restart dereference of the persistent pointer will read uninitialized data. Assuming writeback (WB) caching mode, this can be avoided by inserting cacheline flushes for the freshly allocated persistent locations N before the assignment to the global persistent pointer p. This ensures that the initialization and publication are visible in this order on NVRAM. Higher-level guarantees such as transactional semantics can be built on top of this low-level visibility constraint.

[1] Note that hardware or compiler reordering may have a similar effect.

However, reasoning in terms of low-level interfaces, such as cacheline flushes, is error-prone. Such approach would be akin to memory fence based multithreaded programming, which are not a very successful programming paradigm. Additionally, insertion of such cache flush instructions at appropriate program points requires recompilation and precludes the use of existing code. This work explores the viability of write-through (WT) caching mode for persistent data. Using WT mode on `Intel x86-64` [20], all CPU stores result in writes to all levels of caches and through to system memory. Reads continue getting the benefit of caching. Though an individual WT CPU store may be slower than a corresponding WB one, expensive cache flushes are no longer required, and unmodified legacy code may be reused in some contexts. In the context of traditional programming where volatile caches are functionally invisible, WB is traditionally accepted as a superior choice to WT. However, expensive instructions required to manipulate the state of caches when manipulating persistent data in `NVRAM` warrants consideration of caching policies other than conventional WB.

A.2 Contributions

The experiments in this work were performed on Intel x86 architecture before Intel released the ISA extensions which included optimized instructions (e.g. CLWB) to make stores visible in persistence domain. In fact, the experiments conducted as a part of this work highlighted the need for such optimized instructions. The experiments here report numbers based on Intel's existing CLFLUSH instruction at the time. In the light of the existing architectures at the time, this work made the following contribution through the study of such architecture in the light of developing programs that utilize `NVRAM`-based durability:

1. asserts that using WT "selectively" for persistent data structures appears to be a viable alternative, since programmability benefits with WT are a given in any application, and
2. an experimental version of a Linux kernel that allows programmer to selectively switch the caching policy of virtual address range associated with the persistent

data on-the-fly via a set of system calls.

The experimental results reported by this work may need revision in the light of emerging changes in ISA and memory architecture. However, given the benefits of WT caching policies from the perspective of programmability of NVRAM (see A.3), the case that this work makes for selectively using WT caching policy for persistent data for certain types of workloads still holds true.

A.3 Description of Our Approach

This work assumes a programming environment where failure-atomic sections of code are used to transition data structures from one consistent state to another. In a multithreaded program, such a section of code provides transactional guarantees of failure-atomicity, consistency, isolation, and durability. Failure-atomicity implies an all-or-nothing behavior for visibility of updates to NVRAM. Publication safety [59] is honored so that if a failure-atomic section completes, the effects of all operations on persistent data executed within or before that code section must have reached `NVRAM`.

While a programmer reasons in terms of transactional semantics, the underlying implementation may use a write-ahead log to track accesses to persistent locations to support failure-atomicity [16, 17]. Once the transactional region commits, the logs have to be flushed to NVRAM before the persistent user locations are written and flushed out. Once all of this is successfully performed, the log entries can be discarded.

Consider figure A.1 where a node of a list is allocated, initialized, and inserted at the head. As shown, most of the code sequence can be non-transactional. This allows reuse of existing library code, such as for creating a complex data structure, and inserting that data structure into client objects atomically. This is similar to the previous publication example. Since failure-atomicity is not implied for non-transactional code, log entries are not required outside transactions. But to support publication safety, the implementation must make sure that all updates made before a transaction reach NVRAM at or before the commit point of that transaction. This ensures that restart code that sees a new head value

will also see the corresponding node fields. If WB caching mode is used, and conventional single- line cache flush instructions are used, the non-transactional persistent updates must still be tracked (or directly flushed) so that the relevant addresses can be flushed out at or before the commit point of the succeeding transaction. This incurs costs for non-transactional code, and prevents reuse of existing library code to ac- cess persistent data, even outside transactions.

Now consider using WT caching mode for figure 2. Since any write will be written all the way down to system memory, the non-transactional code does not have to be tracked at all. The transactional code will still require logging in order to support failure-atomicity. Visibility constraints can be enforced without added code, and hence legacy code can be reused without modification outside of transactions.

A.4 Methodology

This work explored performance tradeoffs of different caching modes available on `Intel x86-64` [20]. In addition to `WB` and `WT`, we experimented with two additional caching modes using `Linux 2.6.32`: uncacheable (`UC`, bypasses cache for read and writes), and write combining (`WC`, same as `UC` except writes are combined using a write-combining buffer for efficiency). In the write combining (WC) caching mode, writes are retained in the WC buffer temporarily (see figure 3.1), improving performance. The WC buffer does not participate in the cache coherence protocol, and thus the WC mode provides very weak memory ordering guarantees.

A.4.1 Linux Kernel Modification

Experimenting with wide range of caching policy requires having fine control of caching policies over virtual address range at the memory page level granularity. This section outlines modifications to `Linux` kernel such that programmer have control over caching policies. We implemented system calls such that user level applications could request changes to the caching mode of a memory region mapped to persistent storage (defined

PAT Entry	Cache Policy
PAT0	Write Back
PAT1	Write Through
PAT2	Uncacheable
PAT3	Strictly Uncacheable
PAT4	Write Back
PAT5	Write Through
PAT6	Uncacheable
PAT7	Strictly Uncacheable

Table A.1: Page attribute table after Linux boots up in `Intel x86-64` [20]

by a starting and an ending virtual address) at page-level granularity. Although `WT` is available as an option in `Intel x86-64`, recent versions of `Linux` do not support this mode by default. Hence, we modified the `Linux` kernel initialization code to enable `WT` support.

In `Intel x86-64` with kernel-level privilege, one can specify caching policy at a page-level granularity. Using bits `PAT` (7th bit), `PCD` (4th bit), and `PWT` (3rd bit) in the page table entry, one can point to the suitable entry in the Page Attribute Table (`PAT`) [60]. `PAT` is a 64-bit register where each of eight entries is represented by 8 bits. Table A.1 shows the default `PAT` values. Notice that entries 4-7 mirror entries 0-3. For instance, setting `PAT` to 1, `PCD` to 0 and `PWT` to 1 would result in pointing to the 5th entry in `PAT` which is write through (WT) by default in Intel x86-64 following a restart or power- up. Current Linux (2.6.x or higher) does not support `WT` policy. Hence, as a part of initialization, Linux changes the 1st and the 5th entry in `PAT` register to Write Combining (`WC`). We change the 5th entry to Write Through (`WT`) and leave other entries unchanged. Next, we add system calls to Linux kernel to change cache policy of page(s) in a given address range to write back, write through, write-combining, and uncacheable.

To modify bits in page table entry, we utilize Linux kernel API. When a system call is made by a user application process, the system call code obtains the pointer to `task_struct` for the current process. Next, it obtains the `mm_struct` inside the `task_struct`, which contains information related to memory management. `Linux` employs four levels of

hierarchy for page table management [60]. The starting address of the top level page table Page Global Directory (PGD) is stored in mm_struct. Using the mm_struct and the given address, the provided macro computes the corresponding PGD entry for the given address. Next, using the PGD entry and the current address, it computes the corresponding entry in second level table called the Page Upper Directory (PUD). Likewise, using the PUD entry and the current address, it computes the corresponding entry in third level table called the Page Middle Directory (PMD). Finally, it obtains the Page Table Entry (PTE) using the PMD entry and the current address. We modify the PTE, write the new value using the macro set_pte, and flush the Translation Lookaside Buffer (TLB) entry for the modified page.

A.5 Benchmarks

In order to study the performance characteristics of different caching modes in Intel x86-64, we experimented with different flavors of a synthetic benchmark and 3 persistent data structures. The synthetic benchmark linearly traverses an array within a loop, spanning several memory pages, performing: R (read), or W (update to each element), or RW (read and update to each element). The array takes about 800KB. Transactions are not used in the synthetic benchmark. The persistent data structures used are a multithreaded queue, a failure-atomic version of Christopher Clark's hashtable [61] using transactions, and a multithreaded copy-on-write array-based list[2] (henceforth called cow_al).

In queue, two threads insert and remove 100,000 elements making it a write-intensive benchmark. In hashtable, the program inserts and removes 4000 elements with traversal in between, making it read-write balanced. cow_al maintains an internal array that, once created, is never modified. For every mutating operation, a fresh copy of the array is created, modified, and a pointer atomically switched. To allow multiple writers, a version counter is maintained if it has changed since the mutation started, a fresh copy of the internal array is made with the updated version and the process repeats until the atomic update is successful (with the original version). A read operation, such as a query, obtains a handle to

[2] This benchmark is inspired by Java's CopyOnWriteArrayList.

the internal array and proceeds, without any cross-thread interference or synchronization. Our `cow_al` driver maintains 2 threads, each of which repeatedly performs a mutation followed by a traversal of the list such that each thread makes a total of around 1,000,000 writes to persistent locations. A distinguishing characteristic of this benchmark is that most of the code in this benchmark is non-transactional, the only transactional one being for the atomic pointer switch. All reported results are averages over 4 runs and are obtained on a quad-core Intel(R) Xeon(R) E5620 with 12M cache and running at 2.4GHz.

A.6 Results

The first three sets of columns in figure 3 show the runtimes for the synthetic loop benchmark in each of the described flavors – note that no visibility constraints were added here. It is evident that `WT` has read characteristics (`R`) of `WB` and write characteristics (`W`) of `UC`. The latter indicates that the write-combining buffer is not used and hence an mfence is not required after a `WT` store to enforce ordering. Volume 3A, Section 11.3 of Intel64 Architecture Software Developer's Manual [20] mentions that write combining is allowed in `WT`. However, our experimental results indicate that the `Intel x86-64` machine used in this experiment does not use it. Likewise, `WC` has read characteristics (`R`) of `UC` and write characteristics (`W`) of `WB`. As expected, stores in `WT` mode are expensive. But given that stores to persistent data in `WB` mode require expensive cache line flushes, the question is whether `WT` mode is attractive when ensuring correct visibility semantics on `NVRAM`. In our experiments, we observed that a cache line flush (guarded by mfence) takes around 300 cycles to complete. Interestingly, our results appear to indicate that a cache line flush is equally expensive for clean, dirty, and invalid lines in `Intel x86-64`.

Next, assuming that the array resides in persistent memory, we added sufficient instructions to ensure that stores become visible in order. The R benchmark stays unchanged as it does not contain stores. The modified `W` and `RW` benchmarks are denoted by adding `_b` to their corresponding names, i.e. `W_b` and `RW_b`.

In `WB` mode, the visibility constraint is imposed by a `FLUSH` [32]. On `Intel x86-64`, a

```
//allocate
1: node = nvm_alloc(sizeof(node_t));
//initialize
2: node->value = val;
3: node->next = head;
//publish
4: atomically {head = node;}
```

Figure A.1: Typical allocation, initialization, publication sequence

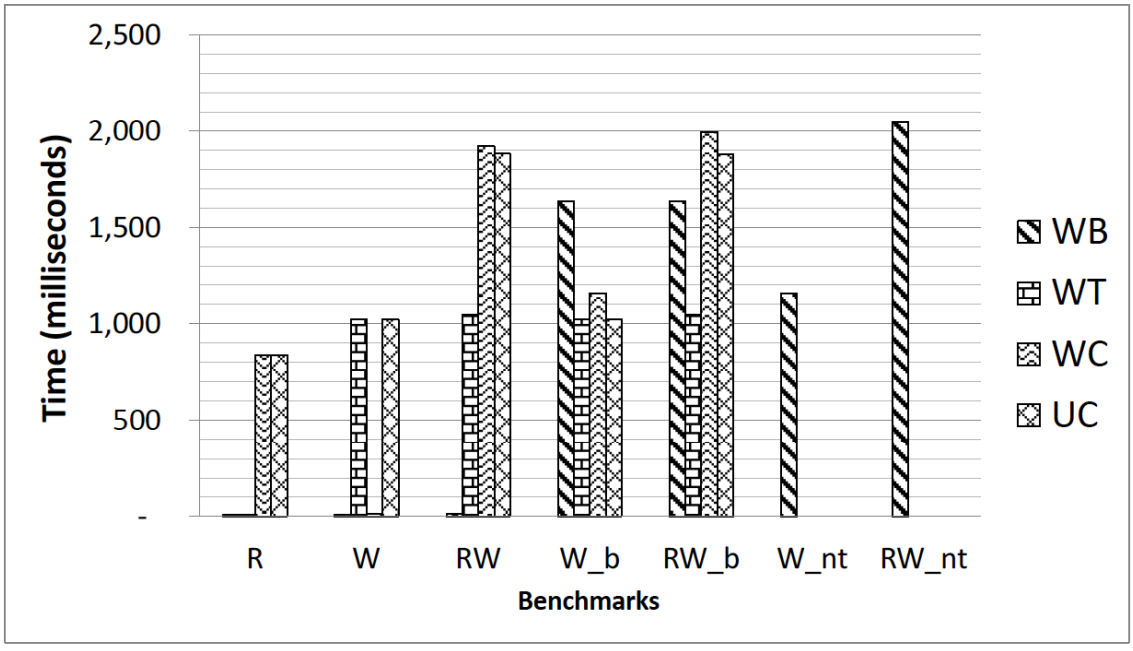


Figure A.2: Runtimes for synthetic benchmark flavors

FLUSH comprises a memory fence (mfence), followed by a cache line flush of the relevant address (clflush), followed by another mfence. The first mfence ensures that the store buffer is emptied before issuing the clflush. The memory fences also ensure the correct instruction execution order. As noted earlier, more optimized instructions (e.g. CLWB) for flushing a cache line have been added to Intel and ARM ISA since the publication of this work [11, 13]. The results reported below may vary with these new instructions. However, the actual machine with these instructions are still not widely available and new ISA itself have gone through several revisions where certain instruction (e.g. pcommit) has been deprecated (see chapter 2). Under UC and WT modes, we insert a directive asm volatile ("" ::: "memory") following a store to prevent compiler instruction reordering. Since the Intel x86-64 architecture follows a TSO memory model, a separate memory fence is not required. We believe that model is semantics-preserving but the Intel x86-64 memory model may need further clarification for non-WB caching modes. In case of WC mode, W_b and RW_b require a hardware memory fence (mfence) after stores so that the write-combining buffer is flushed. Lastly, in WB mode, we experimented with movntq (move with a non-temporal hint) followed by mfence to achieve visibility and ordering guarantees of memory stores. We denote these benchmarks by adding _nt to their original names (i.e. W_nt and RW_nt). The aim is to understand any differences in the performance characteristics between using movntq and WC mode.

In figure 3, the last four sets of columns show runtimes for W_b, RW_b, W_nt, and RW_nt. Results for both W_b and RW_b show that employing a WT mode is around 50% faster than WB mode. Furthermore, W_b under WT mode runs around 12% faster than W_nt. The memory pages in RW_nt are mapped in WB mode and hence this benchmark suffers from cache line eviction each time a write to the line is performed (as outlined in Volume 1, Section 10.4.6.2 of [20]), which slows down reads. This eviction cost may also explain the slight difference between the runtimes of RW_b in WC mode and RW_nt. As a result, RW_b under WT mode runs $2x$ faster than both RW_b under WC mode and RW_nt. We note here that results not shown here indicate that the performance characteristics of the synthetic

benchmark remain unchanged regardless of whether the array fits in L1 cache.

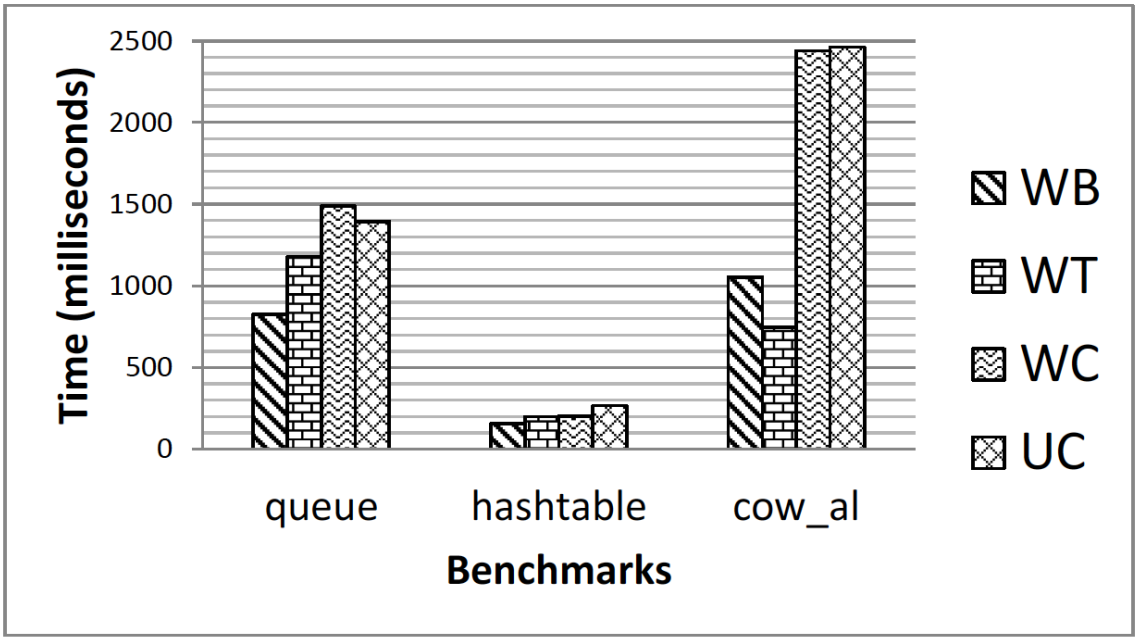


Figure A.3: Runtimes for persistent data structures

Figure A.3 shows the performance comparison for the persistent data structures. queue performs the best in `WB` mode — it is 43%, 80%, and 69% slower in `WT`, `WC`, and `UC` modes respectively. queue is write-intensive with some interspersed reads and most of the updates in this benchmark occur within transactions. Note that transactional implementations require frequent cache line flushes in `WB` and frequent memory fences in `WB` and `WC` modes. `hashtable` also performs the best in `WB` mode – it is 26%, 30%, and 70% slower in `WT`, `WC`, and `UC` modes respectively. Note that hashtable has a sizable number of both reads and writes, so a mode that offers good performance for both access patterns will provide good performance. Additionally, all updates to persistent data structures occur within transactions in hashtable.

On the other hand, `cow_al` performs the best in `WT` mode – it is 41%, 226%, and 229% slower in `WB`, `WC`, and `UC` modes respectively. `cow_al` has a balanced number of reads and writes but the distinguishing characteristic is that almost all of them are nontransactional.

It appears that in a workload with such a pattern (as exemplified by a copy-on-write style implementation), WT mode may perform the best because most of the memory accesses are unconstrained and the long latency of persistent writes in this mode can be hidden by the microarchitecture. WB mode suffers because the non-transactional writes still need to be flushed out of the caches. Reads are still important which explains the poor performance of WC and UC. We believe that the performance characteristics of cow_al (shown in figure A.3) mostly track that of RW_b (shown in figure A.2). In general, the average performance of the 3 benchmarks in WT mode is competitive, and it provides the additional programmability benefits discussed in Section A.3.

A.7 Related Work

Volos et al. [16] use movntq along with mfence to make writes to persistent data instantly visible in persistent memory. They use FLUSH along with regular WB stores if reads are involved. To the best of our knowledge, we are the first to present the implications of using WT caching mode for persistent data. Additionally, we present a comprehensive comparative study and analysis of all the caching modes typically found in modern architectures. Coburn et al. [17] and Condit et al. [14] both advocate changes at the hardware level to enforce consistency and visibility of persistent data. In contrast, we focus on techniques that use existing architectural support.

A.8 Summary

While persisting data in NVRAM, certain consistency semantics have to be maintained in order for such data to be reusable across machine crashes and restarts. We presented performance tradeoffs of various caching modes on modern architectures that can be used in such a context. There is no universally dominant strategy, but overall WT appears competitive. Its ability to accommodate unmodified legacy code outside transactions may give it an edge among caching strategies supported by current processors. Clearly these results may change in the presence of new cache flushing primitives, such as a mechanism

for flushing a cache line to memory without invalidating the cached copy.

References

[1] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 267–278, ACM, 2009.

[2] "Process integration, devices and structures," *International Technology Roadmap for Semiconductors (ITRS)*, 2013.

[3] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and future directions for the scaling of dynamic random-access memory (dram)," *IBM J. Res. Dev.*, vol. 46, pp. 187–212, Mar. 2002.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[5] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 2–13, ACM, 2009.

[6] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

[7] Micron, "3D XPoint Technology." https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[8] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 707–722, ACM, 2015.

[9] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

[10] M. H. Kryder and C. S. Kim, "After hard drives – what comes next?," *IEEE Transactions on Magnetics*, vol. 45, pp. 3406–3413, Oct 2009.

[11] Intel Corp., *Intel Architecture Instruction Set Extensions Programming Reference*.

[12] A. M. Rudoff, "Intel developer zone: Deprecating the pcommit instruction." https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[13] D. Brash, "Arm community: Armv8-a architecture evolution." https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution.

[14] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[15] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 265–276, IEEE Press, 2014.

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[17] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[18] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm_malloc: Memory allocation for nvram," in *Accelerating Data Management Systems Using Modern Processor and Storage Architectures Workshop, In conjunction with VLDB*, 2015.

[19] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, (New York, NY, USA), pp. 433–452, ACM, 2014.

[20] Intel Corp., *Intel64 and IA-32 Architectures Software Developer's Manuals Combined.*

[21] "Pmem.io: Persistent memory programming." http://pmem.io/.

[22] A. L. Hosking and J. Chen, "Mostly-copying reachability-based orthogonal persistence," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, (New York, NY, USA), pp. 382–398, ACM, 1999.

[23] L. M. Silva and J. G. Silva, "System-level versus user-defined checkpointing," in *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No.98CB36281)*, pp. 68–74, Oct 1998.

[24] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental evaluation of application-level checkpointing for openmp programs," in *Proceedings of the 20th*

Annual International Conference on Supercomputing, ICS '06, (New York, NY, USA), pp. 2–13, ACM, 2006.

[25] P. Snyder, "tmpfs: A virtual memory file system," in *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pp. 241–248, 1990.

[26] H. Chu, "Mdb: A memory-mapped database and backend for openldap," *3rd International Conference on LDAP(LDAPCon)*, Oct. 2011.

[27] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999.

[28] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[29] "Google leveldb." https://github.com/google/leveldb/.

[30] "Tokyo cabinet: a modern implementation of dbm." http://fallabs.com/tokyocabinet/.

[31] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, (New York, NY, USA), pp. 4:1–4:8, ACM, 2013.

[32] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2011.

[33] T. Gao, K. Strauss, S. M. Blackburn, K. S. McKinley, D. Burger, and J. Larus, "Using managed runtime systems to tolerate holes in wearable memories," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 297–308, ACM, 2013.

[34] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, "Readings in object-oriented database systems," ch. An Approach to Persistent Programming, pp. 141–146, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[35] A. Dearle, G. N. C. Kirby, and R. Morrison, "Orthogonal persistence revisited," in *Proceedings of the Second International Conference on Object Databases*, ICOODB'09, (Berlin, Heidelberg), pp. 1–22, Springer-Verlag, 2010.

[36] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg, "Mcrt-malloc: A scalable transactional memory allocator," in *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, (New York, NY, USA), pp. 74–83, ACM, 2006.

[37] H.-J. Boehm, "Fast multiprocessor memory allocation and garbage collection," Tech. Rep. HPL-2000-165, Internet and Mobile Systems Laboratory, HP Laboratories, Palo Alto, CA, December 2000.

[38] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (New York, NY, USA), pp. 117–128, ACM, 2000.

[39] J. Evans, "Jemalloc: A scalable concurrent malloc(3) implementation." https://github.com/jemalloc/jemalloc, year = 2006.

[40] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman, "Conservative garbage collectors for general memory allocators," ISMM '00, (New York, NY, USA), pp. 71–79, ACM, 2000.

[41] H.-J. Boehm, "A garbage collector for c and c++." http://www.hboehm.info/gc/gcdescr.html.

[42] B. C. Kuszmaul, "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2015, (New York, NY, USA), pp. 41–55, ACM, 2015.

[43] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Softw. Pract. Exper.*, vol. 18, pp. 807–820, September 1988.

[44] "NVDIMM special interest group." http://www.snia.org/forums/sssi/NVDIMM.

[45] C. Mellor, "SanDisk, HP take on Micron and Intel's faster-than-flash XPoint," *The Register*, October 2015.

[46] M. Aigner, C. Kirsch, M. Lippautz, and A. Sokolova, "Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures," in *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '15, (New York, NY, USA), ACM, 2015.

[47] P.-A. Larson and M. Krishnan, "Memory allocation for long-running server applications," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), pp. 176–185, ACM, 1998.

[48] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

[49] T. Endo and K. Taura, "Reducing pause time of conservative collectors," in *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, (New York, NY, USA), pp. 119–131, ACM, 2002.

[50] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[51] B. Bernhardsson, "Explicit solutions to the n-queens problem for all n," *SIGART Bull.*, vol. 2, pp. 7–, Feb. 1991.

[52] J. E. Gentle, *Matrix Algebra: Theory, Computations, and Applications in Statistics*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[53] J. E. Denny, S. Lee, and J. S. Vetter, "Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, (New York, NY, USA), pp. 125–136, ACM, 2016.

[54] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sept 2016.

[55] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for c," in *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, (New York, NY, USA), pp. 39–48, ACM, 2009.

[56] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), pp. 157–164, ACM, 1991.

[57] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, "A parallel, incremental and concurrent gc for servers," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), pp. 129–140, ACM, 2002.

[58] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Proceedings of the 2001 Symposium on JavaTM*

Virtual Machine Research and Technology Symposium - Volume 1, JVM'01, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2001.

[59] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc, "Single global lock semantics in a weakly atomic stm," *SIGPLAN Not.*, vol. 43, pp. 15–26, May 2008.

[60] D. Bovet and M. Cesati, *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[61] C. Clark, "A hash table data structure in C." https://github.com/davidar/c-hashtable, 2012.