NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems

Joel E. Denny* Oak Ridge National Laboratory dennyje@ornl.gov

Seyong Lee Oak Ridge National Laboratory lees2@ornl.gov

Jeffrey S. Vetter Oak Ridge National Laboratory vetter@ornl.gov

ABSTRACT

Computer architecture experts expect that non-volatile memory (NVM) hierarchies will play a more significant role in future systems including mobile, enterprise, and HPC architectures. With this expectation in mind, we present NVL-C[1]: a novel programming system that facilitates the efficient and correct programming of NVM main memory systems. The NVL-C programming abstraction extends C with a small set of intuitive language features that target NVM main memory, and can be combined directly with traditional C memory model features for DRAM. We have designed these new features to enable compiler analyses and run-time checks that can improve performance and guard against a number of subtle programming errors, which, when left uncorrected, can corrupt NVM-stored data. Moreover, to enable recovery of data across application or system failures, these NVL-C features include a flexible directive for specifying NVM transactions. So that our implementation might be extended to other compiler front ends and languages, the majority of our compiler analyses are implemented in an extended version of LLVM's intermediate representation (LLVM IR). We evaluate NVL-C on a number of applications to show its flexibility, performance, and correctness.

1. INTRODUCTION

NVM main memory hierarchies are playing an increasingly significant role in most computing systems, such as mobile, enterprise, and HPC architectures. Much of this trend is because NVM devices offers advantages over DRAM memory or magnetic hard disk drives (HDDs), in terms of power, density, performance, or cost [18, 10, 2, 20, 13, 9].

This trend is impacting system architecture and pertinent software (e.g., memory management in operating systems). Initially, these NVM devices have been integrated into existing systems in ways that hide the complexity from higher levels of software and applications: replacing a HDD with a solid-state disk (SSD) that uses NAND-Flash devices. This substitution greatly simplified the transition because the underlying complexity was hidden by the operating system and I/O subsystem, with relatively minor changes to the software stack. However, it becomes increasingly important to design software and applications that exploit the characteristics of these NVM devices. In the case of SSDs, the device drivers, operating systems, and I/O subsystems have been optimized for the underlying devices [5, 16, 15].

As the NVM technologies continue to improve, they become more credible for integration at other levels of the storage and memory hierarchy, such as either a peer or replacement for DRAM (see §2.1). In this case, scientists will be forced to redesign the architecture of the memory hierarchy, the software stack, and, possibly, their applications to gain the full advantages of these new capabilities [13].

Simply put, we posit that these new memory systems will need to be exposed to applications as first-class language constructs with full support from the software development tools (e.g., compilers, libraries) to employ them efficiently, correctly, and portably.

1.1 Key Contributions

In this paper, we present NVL-C: a novel programming system that provides language-level support for NVM main memory. More specifically, in NVL-C, we design, implement, and evaluate several novel static analyses and transformation techniques on these new constructs in order to provide support for efficient, portable, and correct execution of applications that contain NVM data structures. The key contributions of this paper are:

1. We design a novel and intuitive programming model (NVL-C) for NVM that facilitates efficient, portable, and correct execution.
2. We implement a prototype of this programming model using extensions to C and LLVM IR.
3. We describe novel static analyses and transformations in LLVM to support correct and efficient code generation for an underlying memory system including NVM.
4. We evaluate NVL-C on a set of applications to demonstrate its flexibility, performance, and correctness.
5. We identify two optimizations that significantly improve performance in our test applications: NVM pointer hoisting and aggregation of transaction data.

*Work performed while employed as Research Scientist I with the University of Tennessee Joint Institute for Computational Sciences.

[1]Pronounced "novel C".

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 ACM. ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907303

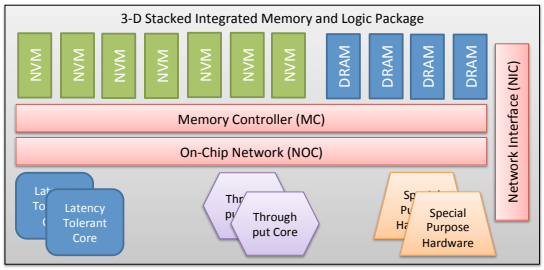


Figure 1: Expected architecture of future memory systems.

2. MOTIVATION AND NVL-C OVERVIEW

2.1 Emerging Memory Hierarchies

Contemporary NVM technology offers a useful compromise between the capabilities of DRAM and HDD [13, 18]. For example, computing systems use flash as SSDs through a POSIX file system interface, or through emerging interfaces, such as NVMe or NAND Flash Memory Interface (FMI). Often, these flash-based SSDs fully replace HDDs in mobile devices and laptops. While more expensive per byte than HDDs, flash is faster, so it reduces the performance bottleneck. However, like HDDs, flash is block-addressable and slower than DRAM, so it is still advantageous for systems to include DRAM banks as a form of volatile memory.

In contrast, future architectures, as illustrated in Figure 1, will include direct load/store interfaces to memory, bypassing the PCIe interface. Moreover, newer NVM technologies, such as PCM, STT-RAM, and ReRAM, promise to be byte-addressable and nearly as fast as DRAM. Thus, they are often described as storage-class memories (SCM). Like flash, SCM is also more power-efficient than DRAM. Unlike flash, SCM technologies are still immature and suffer from write endurance limitations and higher costs.

Eventually, NVM might be able to fully replace both HDD and DRAM. However, at least in the near future, we anticipate that NVM will continue to gradually replace HDD but complement DRAM, as in Figure 1.

2.2 Design Goals

Our design goals for providing programming support for NVM are as follows:

1. *Minimal, familiar, programming interface.* The interface to allocating, accessing, and managing NVM should be intuitively familiar to programmers and have minimal changes to existing language primitives or APIs. The interface must balance this simplicity against the requirements for efficiency, correctness, composability, and modularity. Moreover, the interface should provide concepts to share different types of memories.
2. *Pointer safety.* Programming systems for NVM involve new pointer types with new categories of pointer bugs [4]. Pointer safety constraints should be enforced to minimize or eliminate the occurrence of such bugs. Enforcement at compile time rather than at run time should be preferred when possible to maximize the pace of correct application development.
3. *Transactions.* The programming system must support transactions in order to avoid corruption of NVM in the case of application or system failure.
4. *High performance.* Code automatically inserted by the compiler to manage memory, to enforce safety constraints, and to support transactions can entail a performance penalty. Programmer-specified performance hints and additional compiler passes should be implemented to reduce the overhead of such features.
5. *Modular Implementation.* To maximize flexibility and experimentation, the compiler and runtime should be designed modularly. At the core should be a common compiler middle-end implementation that is general enough to (1) be targeted by multiple compiler front ends for multiple high-level languages and (2) target multiple runtime implementations.

```
1  #include <nvl.h>
2  struct list {
3    int value;
4    nvl struct list *next;
5  };
6  void add(int k) {
7    nvl_heap_t *heap = nvl_open("foo.nvl");
8    nvl struct list *a
9      = nvl_get_root(heap, struct list);
10   nvl struct list *b
11     = nvl_alloc_nv(heap, 1, struct list);
12   b->value = k;
13   b->next = a->next;
14   a->next = b;
15   nvl_close(heap);
16 }
17 void remove(int k) {
18   nvl_heap_t *heap = nvl_open("foo.nvl");
19   nvl struct list *a
20     = nvl_get_root(heap, struct list);
21   #pragma nvl atomic heap(heap)
22   while (a->next != NULL) {
23     if (a->next->value == k)
24       a->next = a->next->next;
25     else
26       a = a->next;
27   }
28   nvl_close(heap);
29 }
```

Figure 2: NVL-C Linked List Example

2.3 Programming Model Overview

Figure 2 presents an example of an NVM-stored linked list implemented in NVL-C. The type `struct list` is the list's node type. The `add` function inserts a node with value `k` at the beginning of the list. The `remove` function iterates the list and removes all nodes with value `k`. This example is based on a similar example presented by Coburn et al. for their NV-heaps system [4].

This example demonstrates some of the ways in which NVL-C addresses the design goals we enumerated in §2.2. First, notice that this NVL-C code is almost the same as the C code you might write for a linked list stored in volatile memory. The only required differences are (1) the `nvl` type qualifier to mark pointers into NVM and (2) the API function calls for memory management. Second, if NVL-C didn't require and enforce the `nvl` type qualifier, we could have accidentally coded the `add` function to call `malloc` instead of `nvl_alloc_nv` without receiving any warning from the compiler. As a result, we would accidentally link a volatile-memory-stored node into our NVM-stored linked list, producing a persistent dangling pointer at program termination. Finally, the `remove` function uses the NVL-C `atomic` pragma to declare the `while` loop as a transaction. The system then guarantees that, despite any power outages or other failures, either all or none of the nodes with value `k` are removed.

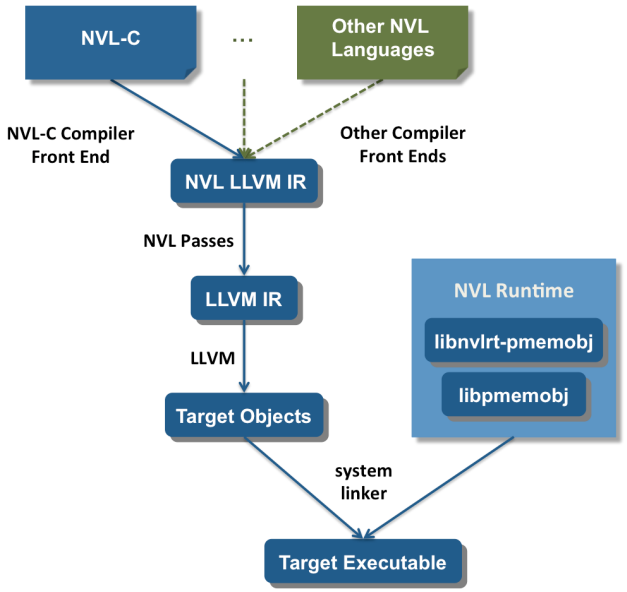


Figure 3: NVL-C System Implementation

2.4 Implementation Overview

As depicted in Figure 3, our NVL-C system implementation consists of a modular compiler and runtime that address our final design goal from §2.2. First, the NVL-C compiler front end's job is minimal: it validates the input NVL-C code and translates it to an extended version of LLVM IR. We use the term *NVL* instead of NVL-C to identify all remaining components of our NVL-C system implementation. The reason is that we designed those components to be shared with future implementations for other high-level languages, such as C++ or Fortran, if extended to support our NVM programming model. These components include the *NVL passes*, which are a set of LLVM passes that perform the bulk of the NVM-related compiler analysis and transformation work, and the *NVL runtime*.

Second, we have currently built our NVL runtime on top of Intel's `libpmemobj` library in order to use its NVM allocation functionality and transaction functionality [1]. However, we have encapsulated the interface of the NVL runtime in our `libnvlrt-pmemobj` library so that `libpmemobj` can easily be replaced by alternate runtime implementations in the future.

2.5 NVL-C Advantages

NVL-C is inspired by the NV-heaps system of Coburn et al. [4] and shares many of the same goals and programming model features. However, NVL-C also has a number of advantages, most of which stem from the fact that NVL-C is a language extension, not just a library:

1. *Improved interface*: whereas the NV-heap's programming interface is a C++ API, NVL-C's interface consists of a small set of C language extensions. As a result, NVL-C is easier to learn and program: instead of learning specialized macros, types, getters, and setters from a C++ templated class hierarchy, the user can program in familiar C syntax, allocate C types in NVM, and reference those allocations and their members with pointers. The difference in code readability can be seen by comparing Figure 2 with the NV-heaps linked list example [4].
2. *Compiler-supported*: whereas NV-heaps is implemented purely as a C++ library, NVL-C is supported by a compiler. Thus, NVL-C supports (a) additional safety constraints and diagnostics for common NVM programming errors, (b) compile-time enforcement of safety constraints that NV-heaps enforces at run time, (c) specialized compiler analyses and optimizations, and (d) implementation reuse for other high-level languages.
3. *Portable*: NV-heaps targets the fast, byte-addressable NVM technologies of the future, and NV-heaps assumes the theoretical hardware extensions of BPFS [5] for efficient durability support. In contrast, NVL-C is currently implemented on top of `libpmemobj`, which supports a wide range of persistent storage devices, from HDD to flash to future NVM, without special hardware extensions. Still, the focus is on future NVM.

Moreover, we have so far discovered two optimizations that can significantly improve performance: aggregation of transaction data (§3.9.2) and NVM pointer hoisting (§4.4). In this paper, we describe a novel interface for manually specifying the former optimization, but we are also exploring automation of both optimizations as part of our future work.

3. PROGRAMMING MODEL

3.1 Terminology

We use the terms *lvalue* and *rvalue* frequently in this section. By lvalue, we mean an expression that designates an object in memory, and we say that an lvalue is converted to an rvalue by a load of that object from memory. For example, given the declarations `int i;` and `int *p;`, then a subsequent expression `i` or `*p` is an lvalue that designates an integer stored in memory, and that integer is loaded to become an rvalue when `i` or `*p` is used as an operand in an expression, such as `i+1` or `*p+1`.

3.2 The `nvl` and `nvl_wp` Type Qualifiers

NVL-C's most important C extension is the `nvl` type qualifier. Both syntactically and semantically, `nvl` is similar to any C type qualifier, such as `const` or `volatile`: it modifies the behavior of load and store operations on lvalues of so-qualified types. Specifically, for lvalues of `nvl`-qualified types, the objects designated by those lvalues are loaded from and stored to NVM. For brevity, we often use the term *NVM-stored* to describe such a type, lvalue, or object. NVL-C adds one other type qualifier, `nvl_wp`, that has the same syntax and semantics as `nvl` except that it designates weak pointers, which we describe in §3.7. For lvalues of types that are not qualified with `nvl` or `nvl_wp`, the objects designated by those lvalues are loaded from and stored to volatile memory instead. For example, Figure 2 line 19 declares a variable `a`, whose type is not NVM-stored, so `a` is stored in volatile memory. However, `a` is a pointer whose target type is NVM-stored, so the lvalue `*a` is NVM-stored.

3.3 Array and Struct Types

Generally in C and NVL-C, a type qualifier on an array type has the same meaning as that type qualifier on its element type. Thus, an array is NVM-stored if and only if either the array type or its element type is NVM-stored.

When accessing a member of an NVM-stored object of struct type, the compiler adds the `nvl` type qualifier to the type of the resulting member lvalue unless the member's

Pointer Class	Permitted
NV-to-V	no
V-to-NV	yes
intra-heap NV-to-NV	yes
inter-heap NV-to-NV	no

Table 1: Pointer Classes

type was declared NVM-stored. This behavior is similar to the way the resulting member lvalue would be const if the struct object were const. For example, in Figure 2, the `value` member of `struct list` is declared on line 3 with type `int`, but the type of the variable `a` declared on line 19 is `nvl struct list *`, so the type of the lvalue `a->value` is `nvl int`. Likewise, on line 26, the type of the lvalue `a->next` is `nvl struct list * nvl`.

3.4 Pointer Types

Generally in C and NVL-C, type qualifiers on pointer target types affect the behavior of load and store operations on the lvalues that result from dereferencing those pointers or, as in the example in §3.3, that result from accessing a member via those pointers. In the cases of the `nvl` and `nvl_wp` type qualifiers, the sizes of those pointers are also affected: as discussed in §4.2, pointers to NVM are wide.

Generally in C and NVL-C, it is a compile-time error to remove a type qualifier from a pointer's target type by means of a type conversion as that would break the behavior guaranteed by the type qualifier. For example, it is a compile-time error to assign a pointer of type `const struct list *` or of type `nvl struct list *` to a pointer of type `struct list *`. However, while some C compilers by default merely warn about violations of this constraint, the NVL-C compiler strictly enforces it for the cases of `nvl` and `nvl_wp`. Moreover, unlike C type qualifiers, it is also a compile-time error to *add* `nvl` or `nvl_wp` to a pointer's target type by means of a type conversion. These stricter constraints for `nvl` and `nvl_wp` are necessary because arbitrarily removing or adding them for a pointer target type could lead to load and store operations that are invalid for the memory addresses on which they're performed, producing persistent corruption of NVM data and segmentation faults.

As listed in Table 1, Coburn et al. identify several new pointer classes and two novel pointer safety constraints [4]. The first constraint does not permit what Coburn et al. call *NV-to-V pointers*, which are pointers from NVM to volatile memory. The justification is that, when a program terminates, all its volatile memory is cleared while NVM remains, so an NV-to-V pointer becomes a dangling pointer. While Coburn et al. state that their NV-heaps system enforces this constraint at run time, NVL-C enforces this constraint at compile time by constraining the use of the `nvl` and `nvl_wp` type qualifiers. Specifically, any NVM-stored pointer type must have an NVM-stored target type. For example, in Figure 2 line 4, if the pointer target type of the `next` member of `struct list` were declared without the `nvl` type qualifier, then the declaration of `a` on line 19 would be a compile-time error because the lvalue `a->next` would have type `struct list * nvl`, which is an NV-to-V pointer.

V-to-NV pointers are pointers from volatile memory to NVM. Like NV-heaps, NVL-C permits them. For example, Figure 2 line 19 declares the variable `a` as a V-to-NV pointer.

NV-to-NV pointers are pointers from NVM to NVM. For example, the pointer designated by the lvalue `a->next` is an NV-to-NV pointer. The second pointer safety constraint enforced by the NV-heaps system involves NV-to-NV pointers, and we discuss it in §3.6.

3.5 Loads and Stores

Generally in C and NVL-C, a load operation on an lvalue produces an rvalue of the same type except that top-level type qualifiers are removed because top-level type qualifiers have no meaning for rvalues. In the cases of the `nvl` and `nvl_wp` type qualifiers, this means that the object designated by an NVM-stored lvalue is loaded only into volatile memory, and there is no such thing as an NVM-stored rvalue. For example, as we explained in §3.3, in Figure 2 line 26, the type of the lvalue `a->next` that appears as the right-hand operand of the assignment operator is `nvl struct list * nvl`, so this lvalue is stored in NVM. Before the assignment is performed, a load is performed on that lvalue, and the resulting rvalue's type is `nvl struct list *`, so the rvalue is stored in volatile memory.

Storing to an NVM-stored lvalue is the reverse of loading: data is copied from volatile memory to NVM. For example, in Figure 2 line 24, the load for the right-hand operand of the assignment operator produces an rvalue of type `nvl struct list *`, and the left-hand operand is an lvalue of type `nvl struct list * nvl`.

3.6 NVM Allocations

NVL-C does not currently permit adding `nvl` or `nvl_wp` to variable declarations in order to convert their storage durations from automatic or static to persistent. For example, in Figure 2, it would be a compile-time error to `nvl`-qualify the type of the variable `a` on line 19 or of the function parameter `k` on line 17. Instead, NVL-C applications allocate NVM in the same manner as the NV-heaps system: via a hybrid of the traditional HDD and DRAM programming interfaces. First, NVM storage is always organized into separate *NVM heaps*, each identified by a unique file name. For this purpose, the constructs `nvl_heap_t`, `nvl_open`, and `nvl_close` are the NVM heap analogues of C's `FILE`, `fopen`, and `fclose` for normal files. An NVM heap can also be renamed, duplicated, or removed using file system commands. Second, NVM storage is always allocated dynamically within NVM heaps. For this purpose, `nvl_alloc_nv` is the NVM heap analogue of C's `malloc` function for the volatile heap.

For example, Figure 2 line 7 calls `nvl_open` to open the file `foo.nvl`, which must be an NVM heap. `nvl_open` maps the NVM heap's memory into the program's virtual address space and returns a pointer to an `nvl_heap_t`, a data structure that is stored in volatile memory to describe the NVM heap. On that `nvl_heap_t`, line 11 calls `nvl_alloc_nv` to dynamically create a new allocation within that NVM heap. The argument list specifies the number of elements and the type of each element, and the NVL-C compiler uses that type to compute the return type of this call as `nvl struct list *`. Line 15 calls `nvl_close` on that `nvl_heap_t` to close the NVM heap, unmap its memory, and free any volatile memory associated with it.

When an application opens an existing NVM heap using `nvl_open`, there are no V-to-NV pointers into that heap, so there is no way to access any NVM allocation or load any NV-to-NV pointer from the heap except via a distinguished

allocation called the heap's *root*. The root can be set and get using the functions `nvl_set_root` and `nvl_get_root`. For example, line 9 contains a `nvl_get_root` call, whose return type is computed from the second argument as `nvl struct list *`. This example assumes that `nvl_set_root` has previously been called, and `nvl_get_root` would return a null pointer otherwise. Before closing a heap or before the application terminates, it is up to the programmer to ensure that all allocations in an NVM heap are reachable from the root if he wants those allocations to be retrievable later.

For each `nvl_get_root` or `nvl_set_root` call, the compiler computes a *type checksum* that includes all C- and ABI-level details of the root type, including type names, field widths, and signedness. Each `nvl_set_root` call records its type checksum alongside the root, and each `nvl_get_root` call fails if its type checksum does not match the previously recorded type checksum, thus preventing access to the heap using the wrong root type. Because all other allocations in a recently opened NVM heap are accessed via the root, checking the root type is sufficient to check their types. In our example, the heap consists entirely of a single linked list, and the root contains a pointer to the head node.

In §3.4, we described how NVL-C enforces the first of two pointer safety constraints that are also enforced by the NV-heaps system. We now describe how NVL-C enforces the second. As listed in Table 1, Coburn et al. divide NV-to-NV pointers into two categories: *intra-heap* and *inter-heap*, which identify whether the source and target NVM heaps are different [4]. Their second pointer safety constraint does not permit inter-heap NV-to-NV pointers. The justification is that there is no guarantee that the source and target NVM heaps will always be open or even exist at the same time. Moreover, when an NVM heap is reopened, it might be mapped to a different virtual address than when the pointer was created. Thus, the pointer would become a dangling pointer, and garbage collection schemes (see §3.7) would have difficulty tracking references to allocations. To address these problems, inter-heap NV-to-NV pointers could be expanded into arbitrarily large data structures that store NVM heap file names. However, programmers can develop such data structures themselves just as they would for references across any two files in a file system.

Like NV-heaps, NVL-C enforces the constraint against inter-heap NV-to-NV pointers at run time. Any V-to-NV pointer contains an identifier for the target NVM heap. When a V-to-NV pointer is stored to NVM, an NV-to-NV pointer is created, so this identifier is first checked against the identifier of the NVM heap into which the pointer is being stored. If the identifier fails to match, an inter-heap NV-to-NV pointer would be created, so a run-time error occurs. As long as this constraint is enforced, it is redundant for an NV-to-NV pointer to contain an identifier for the target NVM heap. That is, that identifier won't be needed again until the NV-to-NV pointer is loaded back into volatile memory to create a V-to-NV pointer. In that case, the required target NVM heap identifier is just the identifier of the NVM heap from which the NV-to-NV pointer is loaded.

3.7 Automatic Reference Counting

As part of their effort to protect against persistent corruption of NVM data, NV-heaps and NVL-C provide automatic reference counting. In the NV-heaps system, reference counts are stored per object whose type is a C++ class for NVM. However, NVL-C stores reference counts per NVM allocation. Because NVL-C permits NVM allocations of C types, such as arrays and structs, a reference count can then apply to multiple objects.

There are two reference counts for each NVM allocation, both of which must reach zero before the allocation is automatically freed. The first is for NV-to-NV pointers into the allocation, and the second is for V-to-NV pointers into the allocation. Each reference count is automatically incremented and decremented as associated pointers into the allocation are created and either overwritten or destroyed. The NV-to-NV reference count persists in NVM as long as the allocation does, but the V-to-NV reference count is effectively set to zero when the NVM heap containing the allocation is closed or when the application terminates.

If an NVM allocation is part of an NV-to-NV pointer cycle and is unreachable from the root of the NVM heap and from all V-to-NV pointers, then its NV-to-NV reference count can never reach zero, so it becomes a persistent memory leak. Thus, like the NV-heaps system, NVL-C provides both strong and weak pointers so that programmers can avoid pointer cycles. Because NV-to-V pointers are not permitted, it is impossible for V-to-NV pointers to appear in pointer cycles, so all V-to-NV pointers are strong pointers. NV-to-NV pointers whose types are `nvl`-qualified are strong pointers. NV-to-NV pointers whose types are `nvl_wp`-qualified are weak pointers. For example, in Figure 2, the type of the lvalue `a->next` is `nvl struct list * nvl`, which is a strong pointer. If we were to change the declaration of the `next` member of `struct list` on line 4 to have type `nvl struct list * nvl_wp`, then the lvalue `a->next` would be a weak pointer instead. Declaring a weak pointer is the only reason we have found to declare a struct member's type to be NVM-stored, and then all objects of that struct type must be NVM-stored.

For pointer types, strong vs. weak is the only difference between the `nvl` and `nvl_wp` type qualifiers. For other types, there is a major difference: `nvl_wp` is only permitted on pointer types or, as implied by §3.3, on array types whose element types are pointer types. Any other `nvl_wp`-qualified type is a compile-time error. In this respect, `nvl_wp` is similar to C's `restrict` type qualifier. It is fine to assign a strong pointer to a weak pointer and vice versa. In fact, every load of a weak pointer produces a strong V-to-NV pointer. Once the target allocation has been freed, loading the weak pointer produces a null V-to-NV pointer.

Every pointer to NVM must be initialized when allocated or else, when overwriting or destroying the pointer, it is not possible in general to determine whether the previous contents of the pointer are valid and thus whether a reference count must be decremented. This is true even for weak pointers because they are actually implemented as strong pointers to proxies for the target allocations. If an automatically or statically allocated V-to-NV pointer is not explicitly initialized at the time of allocation, the compiler can provide a null initialization. For example, Figure 2 line 19 provides an explicit initialization in the declaration of the V-to-NV pointer `a`. It is a compile-time error if a jump, such as a goto statement, bypasses the initialization of an automatically allocated V-to-NV pointer. This constraint is straight-forward to implement, and ISO C99 states a similar constraint for variably modified types.

```
1  #include <nvl.h>
2  void matmul(nvl_heap_t *heap, nvl float a[I][J],
3              nvl float b[I][K], nvl float c[K][J],
4              nvl int *i) {
5    while (*i<I) {
6      #pragma nvl atomic heap(heap) clobber(a[*i:N])
7      for (int ii=0; ii<N; ++ii, ++*i) {
8        for (int j=0; j<J; ++j) {
9          float sum = 0.0;
10         for (int k=0; k<K; ++k)
11           sum += b[*i][k] * c[k][j];
12         a[*i][j] = sum;
13 }}}}
```

Figure 4: NVL-C Matrix Multiply Example

3.8 Type Safety

Strong type safety in NVL-C can generally help to prevent persistent corruption of NVM data, and it is especially important for correct automatic reference counting. Void pointers, pointers casts, unions, variadic functions, and incomplete struct types are avenues in C to circumvent type safety, so NVL-C constrains their use for NVM-related types. These constraints do not affect normal C types, so NVL-C is still a superset of C. Due to space restrictions, we do not describe these type safety constraints in detail in this paper.

3.9 Transactions

NVL-C enforces run-time safety constraints, such as the constraint against inter-heap NV-to-NV pointers, by terminating the application when a constraint is violated. Applications can terminate prematurely due also to other application or system failures, such as segmentation faults or power loss. Premature termination can leave NVM in an inconsistent state. For this reason, it is important that NVL-C support grouping NVM writes into transactions. Our current implementation of transactions has several limitations (see §3.9.3). As the hardware and system software improves, we will continue to improve NVL-C language capabilities.

3.9.1 ACID Properties

In this section, we describe how a NVL-C transaction addresses ACID properties [7] for its NVM writes.

Atomicity. If a transaction does not commit before application termination, then the next time any application accesses the same NVM heap, all the transaction's NVM writes are first rolled back to restore NVM data to a consistent state. Atomicity entails an overhead in both time and space: old data to be overwritten must be backed up.

Consistency. This property is addressed by the way in which NVM writes are grouped into transactions. For this purpose, there are two kinds of transactions:

1. *Implicit transactions* are either built into the NVL runtime or inserted into application code by the compiler to satisfy internal consistency constraints of NVL-C. Most notably, when storing a pointer into NVM, there is an implicit transaction that includes storing all the pointer's fields, updating NV-to-NV reference counts, and freeing NVM allocations whose reference counts then reach zero.
2. *Explicit transactions* are specified by the NVL-C programmer via the `nvl atomic` pragma to satisfy any additional consistency constraints imposed by application requirements. All NVM writes appearing within the attached C block are considered part of the transaction. For example, the transaction starting at Figure 2 line 21 guarantees that the `remove` function removes either all or none of the linked list nodes of value `k`.

Sometimes implicit transactions alone are sufficient to satisfy an application's consistency constraints. For example, in our linked list's `add` function, the implicit transaction for storing the pointer `b` to `a->next` on Figure 2 line 14 is sufficient because automatic reference counting ensures that the new node `b` is freed if the application terminates before then.

Isolation. In the future, we will extend NVL-C to automatically guarantee that transactions executing concurrently will leave NVM data in the same state as they would if they were to execute serially. We are confident NVL-C can be extended in this manner because NVL-C is conceptually based on NV-heaps, which offers this guarantee [4]. Nevertheless, it is already possible to combine NVL-C with existing parallel programming systems provided the programmer explicitly safeguards NVM data from concurrent access.

Durability. If a transaction commits before application termination, including power loss, then all its NVM writes are durably stored into NVM despite any processor caching or buffering. This property normally entails an overhead in time: any such caches or buffers must be flushed or synced.

clause	requirements	access patterns
`backup`	backup, durable, wlock	rw, wr, wo
`clobber`	durable, wlock	wr, wo
`readonly`	rlock	ro

Table 2: Transaction Data Clauses

3.9.2 Performance Hints

Consider a large matrix multiply that writes its result matrix to NVM. As a simple checkpointing mechanism to avoid losing significant progress in the event of premature application termination, the programmer could specify a transaction per N result rows, for some $N \geq 1$, as in Figure 4. That is, each transaction computes N rows, writes those rows to NVM, and increments an NVM-stored row counter N times so the application knows where to resume after premature application termination. Notice that the old data in the result matrix is irrelevant to the computation and thus to data consistency. Thus, the atomicity property in this case could be relaxed to say: if the transaction does not commit before application termination, then the next time the application accesses the same NVM heap, the transaction's writes to *only the row counter* are first rolled back to restore NVM data to a consistent state. Relaxing the atomicity property in this way is not necessary for correct behavior, but it avoids the backup overhead for the result matrix. However, result matrix writes still entail durability overhead. That is, row counter increments must not be committed if result matrix writes have not been durably stored.

We have devised several data clauses for the `nvl atomic` pragma that enable the programmer to provide performance hints for explicit transactions, such as eliminating backup overhead. These clauses are listed in Table 2. Each use of each clause specifies how the transaction is required to handle a specified segment of NVM. For example, the clause `backup(arr[4:2])` specifies that, for elements 4 and 5 of the array `arr`, old data must be backed up and writes must be made durable. In our matrix multiply example, the `backup`

clause is thus appropriate for the NVM-stored row counter. The `clobber` clause is the same as the `backup` clause except it does not back up old data, so it is appropriate for our matrix multiply example's result matrix. Notice that clause names reflect the handling of old data to be overwritten.

These clauses are also useful for aggregating actions on large contiguous NVM segments that the transaction accesses randomly. For example, in matrix multiply, rather than flush processor caches each time a transaction writes to a cell of its result matrix, it is usually more efficient to flush caches for all of the transaction's result matrix rows once at the end of the transaction. The `clobber` clause controls the granularity of such flushing, and the `backup` clause controls the granularity of backup logging. In the future when our transaction implementation supports the isolation property, the `readonly` clause will be useful for controlling the granularity of read locks on NVM segments that are read but never written by the transaction. The `backup` and `clobber` clauses will also control the granularity of write locks.

Table 2 also lists NVM access patterns to help guide the programmer's selection of appropriate clauses. The `backup` clause is the only clause that is appropriate for an NVM segment in which some bits are read before written (rw) by the transaction. That is, because old data there is used by the transaction, the old data must be backed up in case the transaction must be retried later. If no bits in an NVM segment are rw, but if some are written before read (wr) or only written (wo) by a transaction, then either the `backup` or `clobber` clause might be appropriate, depending on how the data there might be used elsewhere before the transaction is retried. The `readonly` clause is the only clause that is appropriate for an NVM segment that is only read (ro) by a transaction. Specifying a clause higher in this table for a NVM segment for which a clause lower in this table is actually appropriate will not harm behavior, but it might harm performance. The converse does not hold. The `nvl atomic` pragma also supports a `default` clause for specifying the default handling of an NVM segment not mentioned in the other clauses. If the `default` clause is not specified, the default is `backup`, which never harms behavior. NVL-C is free to ignore the default when, based on the access pattern for an NVM segment, it can safely improve performance.

3.9.3 Current Limitations

A couple of NVL-C's current limitations in its transaction support stem from its dependence on Intel's `libpmemobj` transaction support, which has similar limitations [1]. First, as mentioned above, NVL-C does not automatically guarantee the isolation property of ACID transactions. Second, an explicit transaction is limited to a single NVM heap, which must be specified explicitly. While `libpmemobj` does support dynamically nested transactions, we have not yet carefully addressed their relationship with the `nvl atomic` pragma's data clauses, which `libpmemobj` does not directly support.

4. IMPLEMENTATION

4.1 Compilation

As depicted in Figure 3, the compilation of NVL-C is divided into four stages, which we describe in this section.

Front end. In order to maximize analysis and optimization opportunities, all NVM-related constructs in the NVL-C programming interface, including all API function calls, are recognized specially by the front end as compiler builtins. The front end lowers NVL-C to *NVL LLVM IR*, which is standard LLVM extended with novel LLVM metadata, intrinsics, and address spaces that represent the same semantics as the NVM-related constructs from the NVL-C source. We implemented the reference front-end using an open-source C compiler called OpenARC [11].

NVL passes. Existing LLVM passes do not currently understand our LLVM IR extensions. However, we have created a new set of LLVM passes, which we call the *NVL passes*, to lower those extensions to standard LLVM IR before running any other LLVM passes. Lowering to standard LLVM IR is not the only responsibility of the NVL passes. That is, we have designed NVL LLVM IR to be semantically close to the source level so that the compiler front end's job is as minimal as possible, leaving the bulk of the NVM-related analysis and optimization work to the NVL passes. Thus, while we have already extended one C compiler front end to support NVL-C, we expect it to be straight-forward to extend other LLVM compiler front ends and high-level languages to support the NVL-C programming model as well.

LLVM. LLVM performs non-NVM-related optimizations and compiles the LLVM IR into target-specific object files.

Linking. NVL-C compilation does not require special linker support beyond linker support required by C. However, NVL-C applications must be linked with the NVL runtime. Currently, we have built the NVL runtime on top of Intel's `libpmemobj` library, version 0.4, in order to use its NVM allocation and transaction functionality [1]. However, we have encapsulated the interface of the NVL runtime in our `libnvlrt-pmemobj` library, and the compiler never targets `libpmemobj` directly. We have designed the NVL runtime in this way so that `libnvlrt-pmemobj` can be replaced with other versions of the NVL runtime that do not target `libpmemobj`, facilitating experimentation with other runtime implementations in the future.

4.2 Pointers

As with both NV-heaps and `libpmemobj`, NVL-C's pointers to NVM are wide. Specifically, NVL-C's V-to-NV pointers have three fields:

1. `heap`. This field contains the address of a volatile-memory-stored `nvl_heap_t` that contains volatile information about the V-to-NV pointer's target NVM heap, such as its base virtual address. In this way, the `heap` field implements what we described abstractly in §3.6 as the heap identifier, which we explained is redundant when the V-to-NV pointer is stored to NVM and thus becomes an NV-to-NV pointer. That's fortunate because, given that the `heap` field is a pointer into volatile memory, it will be invalid when the NVM heap is later reopened and a new `nvl_heap_t` is allocated.
2. `alloc`. This field is the heap-relative offset of the V-to-NV pointer's target NVM allocation. Because it is an offset not an absolute address, it never needs to be adjusted when an NVM heap is reopened and mapped to a different virtual address.
3. `obj`. This field is the allocation-relative offset of the V-to-NV pointer's target NVM object. As with the `alloc` field, because it's an offset, it does not need to be adjusted when an NVM heap is reopened.

The V-to-NV reference count for each NVM allocation is stored in a cuckoo hash table within the NVM heap's

nvl_heap_t. The hash table key is the alloc field used in V-to-NV pointers to the allocation. On the other hand, the NV-to-NV reference count for an NVM allocation is stored within the allocation itself. Thus, given a V-to-NV pointer, our implementation uses its heap and alloc fields to look up the V-to-NV and NV-to-NV reference counts for the target NVM allocation in $O(1)$ worst case time.

When both the V-to-NV and NV-to-NV reference counts for an NVM allocation a reach zero, a must be freed. However, a might contain pointers to other NVM allocations, and so the NV-to-NV reference counts for those allocations must first be decremented. Unfortunately, the last pointer into a that was destroyed to cause a to be freed might be a pointer to some object o within a. While a can be located via that o pointer's alloc field, there is no means to determine the type of a. Without the type of a, how do we locate all pointers within a? Our solution is to take advantage of the redundant heap field in NV-to-NV pointers. That is, for NV-to-NV pointers, we rename the heap field to nextPtrObj, and our implementation uses it to form a linked list of pointers. Within any pointer within a, the nextPtrObj field contains the offset of the next pointer within a. The nextPtrObj field of the last pointer in a is null. The offset of the first pointer within a is stored in a field within a's header. Our implementation is careful not to overwrite these nextPtrObj fields when overwriting an NV-to-NV pointer with another NV-to-NV pointer.

Because V-to-NV reference counts are stored in volatile memory, when an NVM heap is closed or when the application terminates, all that heap's allocations' V-to-NV reference counts are destroyed. In the case of closing the heap, their V-to-NV reference counts are first set to zero, and any required frees are performed. However, as Coburn et al. note [4], when an application terminates without closing an NVM heap, any of that heap's allocations whose V-to-NV reference counts are non-zero but whose NV-to-NV reference counts are zero are leaked. To address this problem, our implementation maintains an NVM-stored doubly linked list of such allocations so it can free them the next time the heap is opened. Our implementation adds an entry for an allocation to this list whenever the allocation's NV-to-NV reference count reaches zero while its V-to-NV reference count is non-zero. Moreover, in each NVM allocation's header, our implementation records a pointer to its entry in that list so our implementation can efficiently remove its entry from the list if its NV-to-NV reference count becomes non-zero again.

4.3 NVL Passes

In this section, we describe in more detail the NVL passes, which we described in general terms in §4.1.

4.3.1 NVLAddTxs

The NVLAddTxs pass instruments NVL LLVM IR to implement transactions that are not built into the NVL runtime. That is, for any store instruction that writes at least one pointer into NVM but that is not enclosed in a transaction, this pass encloses that store instruction in an implicit transaction. For any store instruction that writes anything into NVM, if that store instruction is contained either in an implicit transaction or in an explicit transaction whose clauses do not suppress backup at that store instruction, this pass inserts instructions to perform backup. At the start of each transaction, this pass inserts instructions to perform aggregated backup according to backup clauses. At the end of each transaction, this pass inserts instructions to ensure NVM writes are durable and to commit the transaction.

As described in §3.9, the programmer specifies an explicit transaction by attaching an nvl atomic pragma to a block of C code. The compiler front end must somehow mark the set of LLVM IR instructions generated from each such C block in order for the NVLAddTxs pass to properly identify the start and end of the transaction, identify contained store instructions, and implement the pragma's clauses. Using LLVM IR metadata and intrinsics to associate a C block's LLVM IR instructions with an attached pragma and its clauses is the purpose of our *basic block set* implementation for LLVM, which we described previously for FITL (Fault-Injection Toolkit for LLVM) [6]. We reuse that implementation in our NVL-C compiler front end and NVLAddTxs pass.

4.3.2 NVLAddSafety

The NVLAddSafety pass instruments NVL LLVM IR with run-time safety checks. To any store instruction that writes a pointer into NVM, this pass adds the interheap NV-to-NV pointer check we described in §3.4. To any NVM pointer comparison or subtraction instruction, this pass adds a check that the targets of the two pointer operands are not different allocations because such instructions are usually a sign of erroneous pointer arithmetic. This entire pass can be disabled in order to avoid the overhead of these run-time safety checks. Other run-time safety checks are performed by the NVL runtime, and we have not provided any means to disable them because their overhead is insignificant in comparison to the other operations performed by the runtime functions in which they appear.

4.3.3 NVLAddRefCounting

The NVLAddRefCounting pass instruments NVL LLVM IR with automatic reference counting. To any store instruction that writes a pointer into volatile memory or NVM, this pass adds an increment of the V-to-NV or NV-to-NV reference count for the target allocation of the pointer being written, and it adds a decrement for the target allocation of the pointer being overwritten. For V-to-NV reference counts, it also adds an increment where a new V-to-NV pointer is created in an LLVM register, and it performs a liveness analysis to determine where to add the corresponding decrement. To nvl_alloc_nv calls, it adds initialization of the nextPtrObj fields we described in §4.2. Other parts of automatic reference counting, including data structure extensions, are implemented within the NVL runtime and can be disabled via preprocessor macros when compiling the NVL runtime.

4.3.4 NVLLowerPointers

While the NVL passes mentioned in the previous sections instrument NVL LLVM IR with additional functionality, it is the NVLLowerPointers pass that finally lowers NVL LLVM IR to standard LLVM IR. Thus, NVLLowerPointers must be run after other NVL passes and before any non-NVL LLVM passes. The name of this pass is based on the fact that its primary function is to lower NVM pointers from typed LLVM IR pointers of special address spaces to their struct representation used within the NVL runtime.

Symbol	Description
ExM	Use persistent storage as if extended DRAM
ND	Skip runtime operations for durability
B	Basic NVL-C version w/o Safety, RefCnt, and transaction (T0, T1, ...)
S	Automatic pointer-safety checking
R	Automatic reference counting
T0	BSR + Enforce only durability of each NVM write
T1	BSR + Enforce ACID properties of each transaction
T2	T1 + aggregation using `backup` clauses
T3	T2 + skipping unnecessary backup using `clobber` clauses
T4	T3 at the granularity of each loop
CLFlush	Flush cache line to memory
MSync	Synchronize memory map with persistent storage

Table 3: Symbols Used in the Result Figures

4.4 Bare NVM Pointers

As explained in §4.2, our NVL-C implementation encodes an NVM pointer with multiple fields, which are required to enforce NVL-C's run-time safety constraints and to support automatic reference counting. Before a load or store operation can be performed on an NVM pointer, this encoding must be converted into an address encoding that is understood by the target architecture's load and store instructions. We refer to the resulting address encoding as a *bare NVM pointer* because it strips away all NVL-C metadata and exposes the same address encoding that is used for a normal pointer in C.

In our current implementation, the procedure to convert to a bare NVM pointer is simple: the base virtual address of the NVM heap is retrieved, the offset of the target allocation is added, and the offset of the target object is added. However, that procedure is many times longer than a single load or store instruction and is partially encapsulated in a `libpmemobj` API function that entails further overhead. For an application that spends much of its time performing NVM loads or stores, conversions to bare NVM pointers can then have a severe impact on performance. As part of our ongoing work, we have already found that seemingly minor tuning of this conversion procedure can dramatically alter the running time of some applications.

In applications like our matrix multiply in Figure 4, NVM loads and stores appear in a loop such that each conversion to a bare NVM pointer is a loop-invariant computation that can be hoisted before the loop so it is performed only once. We refer to this optimization as *NVM pointer hoisting* because it eliminates normal NVM pointers from the bodies of loops. Extending our NVL-C compiler to perform this optimization automatically is part of our future work. For our evaluation, we implement this optimization manually per application to measure its performance impact.

5. EVALUATION

5.1 Methodology

Because we are targeting future byte-addressable NVM systems, we have developed a multitier strategy to development and evaluation of NVL-C. To evaluate the performance of the proposed NVL-C system, we ported six applications into NVL-C versions: four kernel benchmarks (STREAM [12], MATMUL, JACOBI, and HASHTABLE) and two Department of Energy (DOE) proxy applications (XSBENCH [17] and LULESH [8]). The ported NVL-C applications were executed on a machine with two eight-core Intel Xeon E5-2643s, 32 GB of DRAM, and a Fusion-io 1.65 TB MLC ioScale2 SSD connected via PCIe bus, running Scientific Linux Version 6.5. As a target device, we used the SSD, which is block-addressable. Because the NVL-C runtime works with both block-addressable and byte-addressable devices, we can also measure the performance of the NVL-C applications on SSD as if it were byte-addressable (*Byte-addressable NVM* mode in result figures), which gives us a rough idea of the expected performance of the SSD if it were byte-addressable, while measurements in a block-addressable mode show the actual performance. For a comparison, the same NVL-C applications were also executed using a RAMDisk, which is byte-addressable but volatile.

As explained in §4.3, the modular design of the NVL-C system allowed us to measure the overhead of each major component (e.g., safety checking, automatic reference counting, transaction, etc.).We also measured the performance effects of various optimizations, such as NVM pointer hoisting (in §4.4), aggregation of transaction data (`backup` clauses in §3.9.2), skipping unnecessary backup (`clobber` clauses in §3.9.2), and increasing the granularity of a transaction region. As a reference, the original C versions of the tested applications were executed only on a DRAM, and all NVL-C performance was normalized against the original DRAM-only versions (100% indicates the same performance as the DRAM-only version.). Table 3 explains the symbols used in the figures.

To verify transaction correctness, we created a synthetic checkpointing-recovery benchmark using NVL-C transactions and performed a stress test by randomly killing the running benchmark with `SIGKILL`. The stress test didn't produce any any data corruption or deadlock, but merely killing the application may not cover all the possible transaction issues, such as problems caused by power failures.

5.2 Application Results

STREAM: STREAM [12] is a synthetic benchmark to measure sustainable memory bandwidth and corresponding computation rate. STREAM consists of memory-bound kernels testing the streamed accesses of array data from/to the target memory device, and thus is suitable for the basic performance study of the NVL-C system. Figure 5 shows the measured performance of its TRIAD kernel on two target devices (SSD and RAMDisk), normalized to the performance of its original C version on a DRAM. In the figure, *ExM* shows the case where the target NVM is used as a secondary memory partition (i.e., extended volatile memory) ignoring any persistence-related issues such as transactions and safety checking, where no NVM pointers are used, and NVL-C provides functions similar to what NVMalloc [3] does on a local NVM. The ExM results in Figure 5a show that the tested SSD achieves performance comparable to DRAM. Figure 5b shows the performance on RAMDisk; because RAMDisk is byte-addressable, *MSync* overhead is negligible, and thus performance in the block-addressable mode is similar to that in the byte-addressable mode. Figure 5 indicates that in all tested devices, 1) the relative overheads for safety checking (*S*) and automatic reference counting (*R*) are negligible, 2) *NVM pointer hoisting* optimization (*Hoisting*) is critical to reduce overhead, while 3) supporting transactions incurs a huge performance penalty, as expected. To mitigate the transaction overhead, we increased the transaction granu-

larity from per element write (*per elm*) to per whole array (*per loop*), which benefits more from transaction aggregation, achieving half of the DRAM performance on SSD in the byte-addressable mode (Figure 5a).

MATMUL: MATMUL is a locally developed dense matrix multiplication kernel. Figure 6a shows the normalized execution times on SSD, broken down into each major component. *T0* refers to a NVL-C version where only durability of its NVM writes is enforced (see 3.9.1). To enforce the durability, the NVL-C runtime performs two types of flush operations: flushing cache lines into the memory (*CLFlush*) and synchronizing buffers in the memory with the mapped file in the persistent storage (*MSync*). If the target NVM is byte-addressable, *MSync* may be skipped depending on the underlying file system supporting the NVM. *T1* shows the case where NVL-C transactions with ACID properties are enforced. As mentioned in §3.9.1, the current NVL-C runtime does not support the *isolation* property yet, but all the tested NVL-C applications are sequentially executed, and thus isolation is implicitly guaranteed. The figure shows that *T1* incurs more *MSync* overhead than *T0*, since *T1* involves more data synchronizations to durably store backup data to enforce atomicity. *T2* and *T3* show that transaction aggregation using `backup` or `clobber` clauses is quite effective in reducing *MSync* overhead. Figure 6b shows the overall performance on SSD when the *NVM pointer hoisting* optimization (*Hoisting*) is applied. The figure implies that the transaction aggregation is so effective that *T2* and *T3* perform similarly in both block-addressable and byte-addressable modes.

JACOBI: JACOBI is another locally developed benchmark that performs a stencil computation solving partial differential equations. JACOBI is more memory-intensive than MATMUL, and thus Figure 6c and 6d show that JACOBI incurs much more *MSync* and *CLFlush* overheads, resulting in significant slowdown and noticeable performance difference between block-addressable and byte-addressable modes. Nonetheless, transaction aggregation and skipping unnecessary backup are very effective, resulting in reasonable performance in the optimized version (*T3*).

XSBENCH: XSBENCH [17] is a mini-app representing a key computational kernel of the Monte Carlo neutronics application OpenMC, one of key DOE applications; XSBENCH calculates macroscopic neutron cross sections, a kernel which accounts for around 85% of the total runtime of OpenMC. XSBENCH deals with neutrons across a wide energy spectrum and materials of many different types, and thus it requires a very large read-only data structure holding cross section points for many discrete energy levels, which are suitable to be put on NVM. Figure 6e and 6f show the performance on SSD; because the major data structures are read-only, transactions have minimal overhead, resulting in overall performance comparable to the DRAM-only version in both block-addressable and byte-addressable modes.

LULESH: LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [8] is one of five challenge problems in the DARPA UHPC program and widely studied as an important DOE proxy application. Even though LULESH is a quite complex proxy application, it is well organized in a way that it distinguishes temporary variables that are initialized before use within a single iteration of the main loop from the variables with loop-carried true data dependencies or read-only variables, the latter of which are good targets to be put on NVM. Therefore, we could easily create its NVL-C version by changing the memory allocation codes for the persistent or read-only global variables and adding NVM qualifiers to their declarations. Figure 6g and 6h show the performance on SSD; while the base NVL-C transaction versions (*T1*) suffer from very large transaction overheads, transaction aggregation and NVM pointer hoisting optimizations work together to effectively reduce the overall overheads. In LULESH, the backup skipping optimization in *T3* is not applicable since all persistent data are read-first, while the backup skipping optimization is applicable only to either write-only or write-before-read data (see §3.9.2).

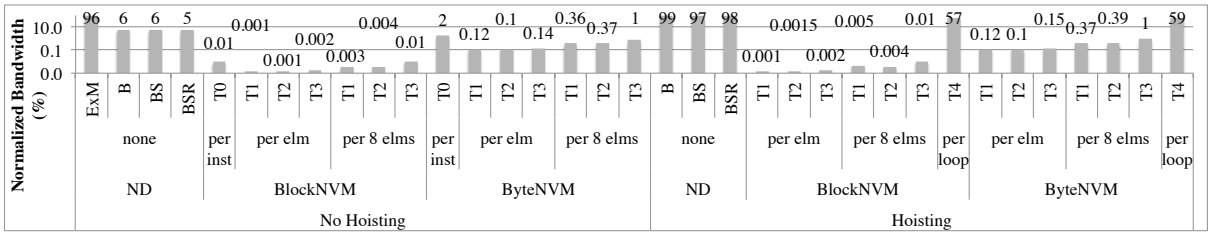
HASHTABLE: HASHTABLE is another locally developed benchmark to measure the performance of typical key-value mapping operations, which are common in database and other data analytics domains. The benchmark consists of four kernels performing key-value pair insertion, update, read, and delete 1000 times, respectively. Figure 7 shows the performance on SSD and RAMDisk. Because the benchmark accesses key-value pairs randomly, transaction aggregation is not applicable. Moreover, skipping backup data is not applicable either, to maintain the consistency of the mapped data structure, which is required for database applications. Therefore, the figure shows that the NVL-C version on SSD suffers from significant overheads, mainly from the durability overhead in transactions (*CLFlush* and *MSync*), while that on RAMDisk achieves reasonable performance. To reduce the durability overhead in the block-addressable devices, special hardware extension [5] or extended low-level Linux primitives [14] will be necessary.

6. RELATED WORK

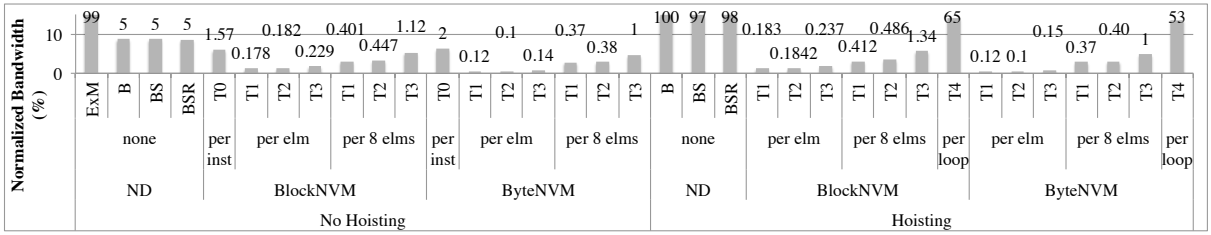
In anticipation of emerging NVM trends, previous research has described new programming systems that utilize NVM as fast and safe persistent storage, as our prior comprehensive surveys illustrate [13, 18]. In this section, we highlight several projects that bear interesting similarities to NVL-C.

First, NVL-C builds on the abstractions of NV-heaps, so we have compared NVL-C to NV-heaps throughout this paper [4]. A key focus of NV-heaps that drove our work is that, because data corruption persists across application termination and power loss, data corruption is inherently more dangerous for persistent NVM storage than for volatile storage, so extra safeguards are needed to protect data from application and system failures. We were unable to obtain the NV-heaps implementation for direct evaluation or extension.

Next, Mnemosyne is a compiler-supported, transactional, NVM programming system that targets future NVM technologies [19]. Unlike NVL-C, Mnemosyne modifies the Linux kernel, but it requires no hardware modifications like NV-heaps. Mnemosyne supports a type qualifier that is semantically similar to NVL-C's type qualifiers but that is used solely for compile-time warnings about mismatched pointer types. That is, the compiler does not instrument associated store instructions to provide automatic reference counting, and it does not enforce NVL-C's safety constraints, such as preventing NV-to-V pointers. A unique feature of Mnemosyne is that, in addition to dynamic allocations, it supports global variable declarations that persist across application termination. However, Mnemosyne assumes that traditional files are used for interchanging data among differ-

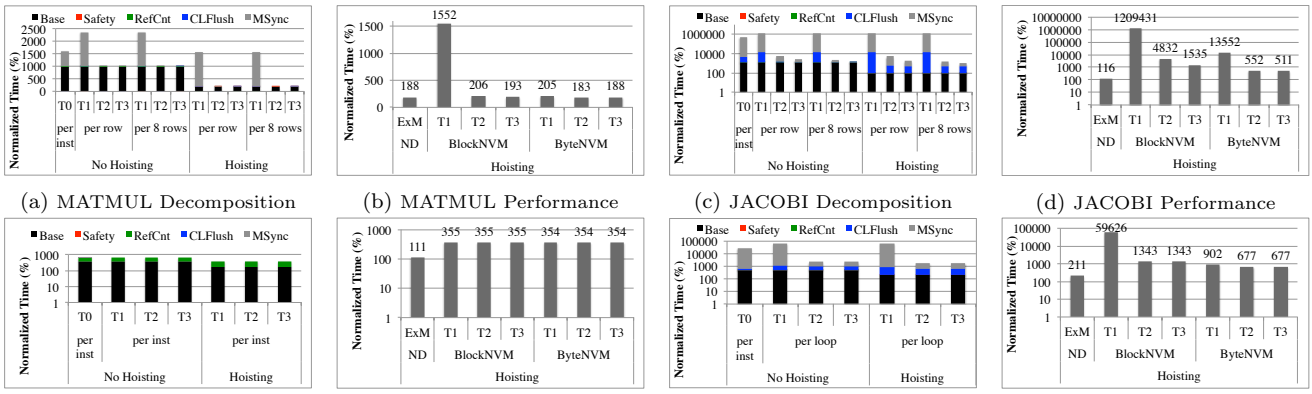


(a) SSD



(b) RAMDisk

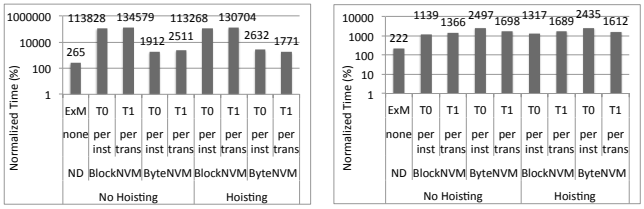
Figure 5: Bandwidth of STREAM TRIAD Normalized to DRAM-only Version



(a) MATMUL Decomposition (b) MATMUL Performance (c) JACOBI Decomposition (d) JACOBI Performance

(e) XSBENCH Decomposition (f) XSBENCH Performance (g) LULESH Decomposition (h) LULESH Performance

Figure 6: NVL-C Benchmark Performance on SSD Normalized to DRAM-only Version



(a) Performance on SSD (b) Performance on RAMDisk

Figure 7: Normalized HASHTABLE Performance

ent applications, and it does not offer the file system interface that NVL-C offers for accessing arbitrary NVM heaps.

Another effort, NVMalloc, is a library that enables out-of-core computations to access aggregate NVM storage as a secondary memory partition [3]. Its focus is the distribution of SSD across compute nodes in extreme scale machines. Additionally, it exploits persistence for checkpointing both DRAM and NVM, but it does not offer transactions or NVL-C's other safeguards related to persistence.

Several systems propose low-level primitives on which NVL-C could be built instead of `libpmemobj` to guarantee atomicity and durability of updates to NVM data in the face of application or system failures. BPFS is a file system that introduces a novel paging technique for targeting future NVM technologies, but it requires a set of theoretical hardware extensions, also assumed by NV-heaps [5, 4]. Park et al. extend low-level Linux primitives to create a simple interface for updates to `mmap`'ed files backed by any NVM device [14]. Specifically, they introduce a new `mmap` flag, `MAP_ATOMIC`, which specifies that writes to the `mmap`'ed file must not be durably stored to the underlying memory until an `msync` call completes successfully.

7. CONCLUSION

In this paper, we have presented NVL-C, a novel, transactional programming system that extends the familiar syntax of C to facilitate efficient and correct programming of emerging memory hierarchies including NVM. In comparison to existing NVM programming systems, NVL-C is more intuitive and flexible for the programmer, and it is more amenable to automated analysis, diagnostics, and optimizations because it is compiler-supported. The design is modular in order to maximize the ability to reuse implementation components among multiple compiler front ends, high-level languages, and runtimes in the future. The core of the compiler is an extension of LLVM, and we have currently built

the runtime on Intel's `libpmemobj`. We have evaluated our prototype implementation by porting a number of kernel benchmarks and DOE proxy applications to NVL-C to make them recoverable across application or system failure. Our results demonstrate that performance is reasonable when two optimizations are employed: NVM pointer hoisting and aggregation of transaction data. Future work includes developing LLVM passes to automate such optimizations and extending transaction support.

8. ACKNOWLEDGMENTS

The authors thank Philip Roth (ORNL) for his help in administering the evaluation testbed, and FusionIO (now Western Digital/SanDisk) for providing our ioScale card.

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the DOE. The United States Government (USG) retains and the publisher, by accepting the article for publication, acknowledges that the USG retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for USG purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

9. REFERENCES

[1] NVM Library. [Online]. Available: http://pmem.io/nvml/. (Accessed January, 2016).

[2] A. Badam. How persistent memory will change software systems. *Computer*, 46(8):45–51, 2013.

[3] W. Chao, et al. NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 957–968, 2012.

[4] J. Coburn, et al. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc 16th Intl Conf Architectural support for programming languages and operating systems*, pages 105–118, Newport Beach, 2011. ACM.

[5] J. Condit, et al. Better I/O through byte-addressable, persistent memory. *Proc. ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[6] J. E. Denny, S. Lee, and J. S. Vetter. FITL: Extending LLVM for the Translation of Fault-injection Directives. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[7] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[8] I. Karlin, et al. LULESH programming model and performance ports overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2012.

[9] B. Lee, et al. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143, 2010.

[10] B. C. Lee, et al. Phase change memory architecture and the quest for scalability. *Communications of the ACM*, 53(7):99–106, 2010.

[11] S. Lee and J. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing. In *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, june 2014.

[12] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[13] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[14] S. Park, T. Kelly, and K. Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proc 8th ACM European Conf Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

[15] S. Pelley, et al. Storage management in the NVRAM era. *Proc. VLDB Endow.*, 7(2):121–132, 2013.

[16] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. *Proceedings of the international conference on Supercomputing - ICS '11*, pages 85–85, 2011.

[17] J. R. Tramm, et al. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[18] J. S. Vetter and S. Mittal. Opportunities for nonvolatile memory systems in extreme-scale high performance computing. *Computing in Science and Engineering special issue*, 17(2):73–82, 2015.

[19] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, 2011.

[20] H. Zhang, et al. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.