Languages Must Expose Memory Heterogeneity

Xiaochen Guo
Lehigh University
xig515@lehigh.edu

Aviral Shrivastava
Arizona State University
aviral.shrivastava@asu.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

Gang Tan
Pennsylvania State University
gtan@cse.psu.edu

ABSTRACT

The last decade has seen an explosion in new and innovative memory technologies. While certain technologies, like transactional memory, have seen adoption at the language level, others, such as sandboxed memory, scratchpad memory, and persistent memory, have not received any systematic programming language support. This is true even though the underlying compiler-level mechanisms for these mechanisms are similar. In this paper, we argue that programming languages must be enhanced to expose heterogeneous memory technologies to programmers, so that they can enjoy the benefits of those technologies and be able to reason about programs that use the advanced features of novel memory technologies. We sketch a language design that allows programmers to specify memory requirements and behaviors, for both data and code. We further describe how a compiler can support such a language and suggest hardware improvements that can improve efficiencies of heterogeneous memories.

CCS Concepts

•**Software and its engineering** → **General programming languages;** *Access protection; Concurrent programming languages;* •**Information systems** → *Storage class memory;*

Keywords

Scratchpad Memory; Nonvolatile Memory; Security; Sandboxing; Transactional Memory; Concurrency

1. INTRODUCTION

While the end of Moore's law and Dennard scaling is largely credited with ushering in the multicore era, a second and more significant shift has also taken place. Increasingly, the value of a new microprocessor is coming not from the sophistication or number of processing pipelines, but novelty at the interface between the pipeline and memory. Innovation at this level takes many forms, to include physical structures, like non-volatile memory (NVM) [13] and uncached scratchpad memories [2], and virtual interfaces like coarse-grained cache-level atomicity (transactional memory [15]) and fine-grained, compartmentalized memory protection (e.g., Intel Software Guard Extensions [9]).

Memory innovations can increase programmer productivity (e.g., by eliminating the need to write code that serializes a data structure to disk), increase performance (e.g., by safely executing irregular codes in parallel), reduce energy (e.g., by favoring the use of low-power coprocessors), and enhance higher-order software properties like confidentiality and integrity. Unfortunately, programming languages have not evolved to facilitate the use of advanced memory features.

For the most part, today's languages assume that memory is a homogeneous array of untyped, volatile, shared memory. While prior developments in memory interfaces, such as caches and separate instruction memories, could hide invisibly behind this language-level interface, new features cannot. Today's languages rarely allow programmers to express what is confidential or persistent, nor do they make it easy for programmers exploit emerging memory-related technologies to reduce power consumption and avoid the overhead of pessimistic synchronization. Worse yet, the lack of an abstract interface to feature-rich memories means that software implementations of those features, suitable for execution on legacy hardware or with programs whose behavior exceeds hardware capacities, is cumbersome and error-prone, often resulting in inefficient code. As a result, the desire to exploit memory features encourages programmers to write machine-specific, unportable software that cannot be analyzed and verified statically.

In this paper, we take the position that programming languages must expose advanced memory features. The implementation of these features must be possible even in the absence of hardware support, and the semantics of these features must be specified independently from actual hardware implementations. Based on our experiences in four separate domains, Security, Persistence, Scratchpads, and Transactions, we propose a set of language features that serve as a starting point for moving beyond homogeneous byte arrays, to regions of feature-rich memory that can be exploited without cumbersome or machine-specific code. Through abstract instrumented memory interfaces at the compiler level, the syntax of our memory interface can be transformed into correct implementations on legacy hardware, and efficient implementations on emerging systems.

2. BACKGROUND AND MOTIVATION

In this section, we briefly review persistent memory, sandboxed memory, scratchpad memory, and transactional memory to motivate the need for language support for feature-rich memories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies 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. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS 2016 October 3–6, 2016, Washington, DC, USA

© 2016 ACM. ISBN 978-1-4503-4305-3…$15.00

DOI: http://dx.doi.org/10.1145/2989081.2989122

Persistent Memory.

Emerging storage-class memory (SCM) technologies have enabled system designers to re-architect main memory to exhibit persistence in addition to byte addressability. There are three leading technologies in emerging SCMs—phase-change memory (PCM), spin-torque transfer magnetoresistive RAM (STT-MRAM), and resistive RAM (ReRAM). These storage-class memories represent information using resistance rather than electrical charge, and retain the stored information after power off [5,8,13]. Persistence can be applied at any level of the memory hierarchy, from STT-MRAM based caches to ReRAM or PCM based main memories. A data structure in persistent memory may require position-independence, so that it can be relocated to different locations in process address spaces. To ensure durability, additional invariants (such as no pointers to volatile memory, or log-based atomic commit) may be required [4]. The programming model can, in turn, restrict when and how persistent memory can be accessed [3], and may introduce the need for complex logging to handle failures in the middle of a region that modifies a persistent heap.

Sandboxed Memory.

It is often necessary for an application to incorporate untrusted code (e.g., third-party libraries and plugins) and data. To protect the application, it is beneficial to isolate untrusted code and data so that their bad effects can be controlled. A sandboxed memory provides the abstraction layer that enables a memory region to hold untrusted code and data and also allows managed interaction with unsandboxed memory. We believe it prudent to take a broad view of sandboxing. For example, instructions in a sandbox might guarantee that the targets of their memory instructions cannot access memory outside the sandbox, and the targets of their return instructions cannot be modified at run time in a manner that violates the control-flow integrity of the program. Sandboxing techniques may be used to ensure both the integrity memory outside of the sandbox, and the confidentiality of certain regions of memory.

Scratchpad Memory.

Scratchpad Memory (SPM) is small, raw memory present close to the core. It is different from a cache in that SPM has a distinct address space, and the data movement in an out of the SPM is controlled by software, rather than performed automatically by hardware. As a result SPM is also called Software Programmable Memory. By eschewing coherence hardware, SPM uses less energy and is simpler to design and verify. SPMs are widely employed in embedded systems, where the system designer — by carefully managing the staging of the data to and from the scratchpad, through the use of buffered Direct Memory Access (DMA) calls — can achieve extremely efficient execution. SPMs are also often used in accelerators, such as GPUs and the IBM Cell processor, to facilitate high-performance, low-power computing, where the cores within the accelerator do not access the global, coherent memory of the host processor, but instead operate on a small private region of memory.

Transactional Memory.

Transactional memory provides atomicity without mutual exclusion, through the use of speculation. Accesses within a transactional region of code are logged, so that writes can be undone and reads can be ensured to remain valid throughout the duration of the transaction. Modern hardware TM systems, such as Intel TSX [15] and products from IBM [7,10,14], are "best effort", and cannot support transactions that perform system calls, exceed a single quantum, or access more data that fits (roughly) in some level of the cache. Transactions that the hardware cannot support can be run in a software mode, in order to still execute in parallel and monitor potential conflicts with concurrent hardware and software transactions.

In order for programmers to use these features, programming languages must expose appropriate interfaces. These interfaces typically introduce some amount of instrumentation to insert the appropriate instructions on individual memory references, and on the boundaries of code regions that employ these features. In some cases, such as confidentiality and persistence, the features do not make sense without the ability to attach their use to certain variables and objects. Furthermore, as we think of these features at a high level, it is increasingly desirable to compose them. For example, the isolation of a SPM could be used to provide a degree of sandboxing, or a transaction might operate over persistent confidential data. We take the position that languages must provide a unified interface that enables the use and continued creation of arbitrary abstract memory features. In addition to making the above four technologies easier to use, such an infrastructure will enable new technologies (encrypted memory, lossy memory, compressed memory, provenance-monitoring memory, etc.), both through software simulation and ultimately hardware support.

3. LANGUAGE DESIGN FOR ADVANCED MEMORY FEATURES

Rapid innovation in memory features presents a challenge to traditional language design. As an example, consider the addition of TM to C++ [1]: an idea from 1993, which gained traction beginning in 2001, led to the formation of a working group in 2007, a draft specification in 2012, and a C++ technical specification in 2015. The feature may be adopted in C++20. If each new memory feature requires new keywords, and complex interactions with existing language features, then each is likely to take the same trajectory.

One challenge for TM in C++ was that it introduced a need for substantial instrumentation machinery. Once that instrumentation is present in the compiler, its re-purposing for new features becomes less burdensome. However, properties like persistence and confidentiality expose an additional need: a feature may be a property of a variable or field, not of a region of code. In a manner similar to annotation types in C# and Java, we anticipate a relatively straightforward path to adding new type attributes and annotations. However, for clarity, we will focus on only four features in the discussion below. To that end, Table 1 depicts two classes of properties that must be exposed to languages: static data properties (SDPs) and code region properties (CRPs).

Static Data Properties	Code Region Properties
Persistence	Risk of Racy accesses
Confidentiality	Access restrictions
	Locality

Table 1: Two Classes of Properties.

3.1 Static Data Properties

SDPs allow a programmer to specify a feature required of an object or field of an object. As a simple example, consider a social graph in which we would like a user's name to be public, but the birthday confidential. An annotation on the birthday field is the most natural way to achieve this end: the programmer specifically does not want to have to identify every attempted use of the field;

instead, the aim is to have the compiler guarantee the field's secrecy. However, rigid annotations make code reuse more challenging. Consider if the graph is persistent: should fields representing pointers to other nodes within the graph be marked as persistent? If they were, then a programmer wishing to copy and then analyze a subgraph would be required to perform the analysis in persistent memory! Instead, we argue that it must be possible for SDPs to be dynamic properties. Dynamic SDP support will enable greater code reuse, as the same graph construction and traversal code can be used on both persistent and non-persistent graphs.

A complicating consequence of dynamic SDPs is that pointer types must become more complex. If a pointer may, at different times, reference memory of different features, then the type of the referenced object must be embedded in the pointer. In addition, certain memories, such as persistent memory, may favor a segmented addressing scheme. In this case, the size of a pointer field might depend on the features of the object it references.

3.2 Annotated Code Regions

Certain memory features, like SPM with non-coherent memory, are only profitable at the point where they are executing enough code to compensate for the cost of data movement; similarly, TM does not make much sense in the context of single memory accesses. Similarly, accesses to confidential data might be permitted only when made from a region that has been granted special permissions.

To support these aims, we propose that languages support arbitrary annotations on regions of code within a function. The definition of a feature includes rules that affect the relationship between an annotated variable access and the features of the enclosing code region. As an example, confidential data may only be accessed from a code region that is appropriately annotated, and such a region may not be nested within a code region explicitly annotated as forbidding confidential accesses.

Annotations should be as abstract as possible, to afford maximum flexibility to the implementation and to enable composition. For example, rather than specify that a region must run as a transaction, we specify that a region has the potential to race with concurrent regions. Alternatively, with a suitable race detector, the attribute could be inverted, and all regions run as transactions unless proven to be free of races. Similarly, rather than request that a region run on a scratchpad, the programmer should provide a prediction of the memory accesses made by the region. If the compiler can prove the prediction likely, it can generate code to offload the computation. This facilitates composition. Suppose that a region that may race, but has a predictable access pattern. It could be executed by (1) atomically fetching the input data, (2) offloading the computation to a scratchpad, (3) computing a diff to represent the writes by the region, and then (4) using a commit protocol to apply the diff only if the read set remained valid.

3.3 A Behavior Model

To unify emerging memory features, it is valuable to have a model for how computation unfolds in the face of pathology. In concurrent programming, a useful model is to imagine that threads run at arbitrary speeds (or, equivalently, that a thread may experience an arbitrary delay between any two instructions). In security, a strong adversary may be able to modify any data memory at any time (because of a overflown buffer, for example), and the system must ensure certain invariants (such as control flow integrity) nonetheless. We propose as a baseline model that a machine can be arbitrarily powered off at arbitrary points, but some invariants might hold when the machine is powered on (e.g., execution resumes from the last checkpoint, all persistent memory is unchanged, but all non-persistent memory might hold arbitrary values). This model subsumes both the "arbitrary delay" model of concurrency, and the "arbitrary memory overwrite" model in security, while also supporting persistence and accelerators.

4. AN EXAMPLE

We present an example to show potential use cases for the proposed language support for feature-rich memories. In this example, a bi-directional social graph consists of nodes representing users and edges representing friendship relations. As shown in Figure 1, a node holds information about a user, which includes sensitive information that needs to be protected via the "confidential" SDP.

The nodes themselves will be persistent. This is achieved at the allocation site, by indicating that the root pointer references a persistent node. Under the rule that a persistent structure cannot have references to volatile memory (which would become dangling references upon a power failure), the friends of the root node must, then, also be persistent. Thus in the event of a power failure, we should be able to retain a consistent graph without losing the most recent updates.

```
struct node {
  string name;
  int age;
  [confidential] date birthday;
  url picture;
  url collage;
  date joined;
  node* friends;
  date last_update;
  date collage_update;
};

// create an empty, persistent graph
[persistent] node* root = new node();
```

Figure 1: Data type for nodes in a social graph.

Code regions that need to access confidential data must be marked with the "confidential" CRP. This declaration simplifies static analysis, because the source of information leakage will be among the code regions that declare "confidential". In the example illustrated in Figure 2, we show a function that will be applied to nodes in the social graph to update users' ages. This requires accessing confidential birthday data. We assume that this is a system task, and that system tasks are performed by selecting a random set of root nodes, and launching a depth-first search from each selected node. The `last_update` field is used to terminate a recursive search, by revealing that the current node was visited by a concurrent system update task. To handle this concurrency, as well as concurrent accesses to satisfy user requests, this code region must be marked both "confidential" and "avoid_races". The runtime system may exploit TSX, SGX, or software libraries in order to achieve the desired behavior. Note, too, that if a researcher copied the graph to volatile memory, the same code should be able to operate effectively. When running on persistent memory, the end of the region should be instrumented by the compiler to ensure the durability and atomicity of any updates.

Instead of enabling confidential accesses, a system designer might wish to prevent certain codes from accessing confidential data. Consider a situation where the owner of the social graph wants to allow a third-party developer to run code that accesses the graph. In this case, the programming language should provide a simple mecha-

```
void dfs_on(node* n)
{
  [avoid_races,confidential] {
    if (too_old(n->last_update)) {
      n->age = year(now() - n->birthday);
      n->last_update = now();
      // request all friends be traversed
      return true;
    }
    else return false;
  }
}
```

Figure 2: Syntax for concurrent accesses to confidential data.

nism to restrict a region to accessing non-confidential data. With the specification, sandboxed memory will be able to manage restricted memory accesses. Decoupling data properties from code region properties allows a rich set of semantics to be expressed without excessive annotations, as shown in Figure 3. Regardless of what CRPs are specified within the supplied function `f()`, the calling context has forbidden access to confidential data, and the runtime system should be able to enforce that restriction.

```
void apply_each(void(*f)(const node*), node* n)
{
  [avoidraces, !confidential] {
    f(n);
  }
}
```

Figure 3: Syntax for restricting access to confidential data when executing untrusted code.

The above two examples show that certain run-time behaviors, like persistence, can be inferred from the dynamic access pattern of a computation. In the cases where the compiler can precisely predict this pattern, or the programmer can make a conservative estimate, we propose this information be provided as a CRP. The compiler can then choose to use an SPM, if the region exhibits high spatial or temporal locality. Programmers might even wish to give a hint that, while they haven't determined the exact access pattern, they expect high locality. In this case, a conservative runtime system could run the code on an SPM, but insert some DMAs into the code on the SPM to fetch data when the statically predicted working set is incorrect or too large for the SPM.

Figure 4 depicts one such case. In our hypothetical social network, a system update task creates a "collage" image for each user, by weighting the user's friends' profile pictures according to the predicted strength of the friendship. For users with few friends, the entire computation might fit on an SPM. The `make_collage` function also uses a `highlocality` CRP internally, so that the image processing task can be executed on an SPM even when the friend traversal cannot.

5. COMPILER SUPPORT

Given a program with annotations on data and code, a compiler translates it to executable code that uses the provided memory technologies in hardware or in software or in a combination of both. Ideally, the compiler should be portable across architectures, which have varying degrees of support for different memory technologies. What we envision is that the compiler takes in a description of hardware support and uses a mix of hardware and software to enforce

```
void update_collage(node* n)
{
  [avoidraces, !confidential, highlocality] {
    if (too_old(n->collage_update)) {
      set s;
      for (friend : friends)
        s.add(weight(n, friend), friend->picture)
      // a highlocality region
      n->collage = make_collage(s);
      n->collage_update = now();
    }
  }
}
```

Figure 4: Syntax for suggesting that execution be offloaded to an SPM.

the desired memory properties that are specified in the program. The description of hardware support tells, for example, if hardware TM is available. For a region of code that disallows data races, the compiler then uses hardware TM when it is available and, if not, uses a software implementation that relies on program instrumentation. The same can happen for sandboxed memory, where the absence of SGX might lead the compiler to emit code that encrypts certain fields or variables.

We propose that compilers adopt an abstract instrumented memory interface at the level of the compiler intermediate language to facilitate the instrumentation support of many memory technologies. The interface must provide a set of instrumentation primitives: (1) a region-begin instrumentation primitive that should be inserted at the beginning of a code region; (2) a region-end primitive that should be inserted at the end of a code region; (3) a memory-read instrumentation primitive that performs instrumented memory reads; (4) a memory-write instrumentation primitive that performs instrumented memory writes. Each primitive is also annotated with a tag about what memory technologies are applicable to the primitive. Our experience suggests that this abstract interface can accommodate transactional memory, sandboxed memory, persistent memory, scratchpad memory, as well as other emerging memory technologies. For a specific combination of memory technologies and architectures, the implementation of those primitives differs. The interface provides a nice separation between a compiler's view of instrumentation and the implementation details.

For instrumentation, the compiler inserts tagged instrumentation primitives at appropriate places according to the annotations in the program. For instance, if a code region should not have data races, the compiler inserts a region-begin primitive at the beginning of the code region, a region-end primitive at the end of the region, and replaces all memory reads (writes) with instrumented memory-read (memory-write) primitives. All the inserted primitives are tagged with transactional memory. At the back end of the compiler, those instrumentation primitives are lowered to appropriate instruction sequences depending on whether hardware transactional memory is available in the target architecture.

One benefit of exposing an abstract instrumented memory interface to the compiler is that the compiler can optimize the inserted instrumentation (for example, by removing unnecessary instrumentations). Due to the similarities among instrumentations, we believe it will be possible to have an optimization engine that can perform general optimizations on all aforementioned memory technologies, and compositions thereof. Figure 5 presents a diagram showing the inputs and outputs of the optimization engine.

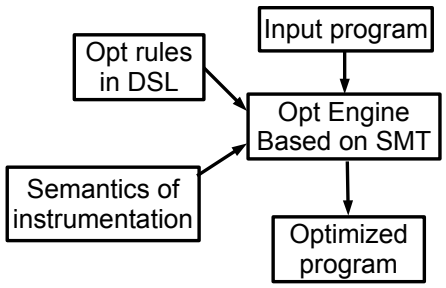

Figure 5: Diagram for the optimization engine.

One input to the engine is optimization rules in a Domain-Specific Language (DSL). Such rules are manually written in the DSL and tell how specific optimizations should be performed with respect to some preconditions, and what kinds of postconditions are satisfied after the optimization. For instance, one rule might specify that the translation of a memory address performed in an instrumented region can be lifted outside of a loop, given a set of conditions [16]. Another input to the engine is the semantics of instrumentation according to a particular memory technology. For instance, the semantics should tell us that a sandboxed memory address stays inside a predetermined address range, or that a conditional load cannot be hoisted above its condition when the transactional memory has weak semantics [12]. The optimization engine then takes an input program, applies optimizations according to the rules and the semantics of instrumentation, and outputs a program with optimized instrumentation.

We recognize that there are many research challenges when building such an optimization engine. First is the design of the DSL, where recent innovations by Lopes et al. provide a promising avenue [11]. Second, we may need to split an instrumentation of a memory access into multiple phases, such as pre-validation, actual memory access, and post-validation, noting that the ordering requirements within a single access, and across accesses, may be dependent on the memory feature [6]. The third challenge is how to write general optimization rules in the DSL. Clearly we should limit the number of rules that are specific to a single memory technology, though some will be unavoidable. Another challenge is the design and implementation of the optimization engine itself. We suspect that SMT (Satisfiability Modulo Theories) solvers may provide the best approach.

Note that when these obstacles are addressed, the outcome will be more than simply an ability for programming languages to use instrumented memories efficiently. Tooling will be possible, so that properties of a program can be analyzed statically. We leave this topic for future work.

6. HARDWARE SUPPORT

The focus of this position paper is on programming language support for programming feature-rich memories. We note, however, that generic language support will reveal broad opportunities for new hardware structures that can can be leveraged to further improve efficiencies within each memory. When many features use the same mechanisms, those mechanisms become more critical, and hardware acceleration more profitable. We briefly suggest ideas for improving hardware, which can provide further benefits.

Address Translation in Hardware.

One commonality across the four memory technologies is some form of address translation, or variable remapping. For transactional memory the true location of reads and writes may be different when write buffering. When persistent memories are mapped into a process address space, the pointers are represented as offsets. In sandboxing approaches, out-of-bound memory accesses may be remapped to in-bound address spaces. Scratchpad memories work their own separate address space from DRAM. Therefore as soon as some data is brought into the SPM, it must then be accessed using the new address. Thus some form of address translation is needed for all of the four example memory features. Implementing address remapping or address translation in hardware will reduce the number of software-level instrumentation instructions for language implementations. If the address translation is implemented in hardware, it will be simpler to implement a small table lookup using Content Addressable Memory (CAM) structures. To further reduce the hardware overhead, it is also possible to reuse and augment existing hardwares. The translation lookaside buffer (TLB) is already used for fast virtual-to-physical address translation. The hardware acceleration of the proposed address mapping can be implemented by augmenting the TLB with reconfigurable logic and additional states for each entry. Based on our initial studies, we suspect that a userspace-managed, TLB-like mechanism for fine-grained remapping into cache-line-sized base/offset regions will be broadly useful.

Metadata Management in Hardware.

The use of metadata is another commonality in all four memory features. All abstract memory features need to create, use, and modify metadata. The metadata typically stores the state of execution that must be maintained to implement the memory feature. For example, techniques to manage the code of an application on SPM work by partitioning the space reserved for code on the SPM into regions, and then map the functions in the application to the regions. Of the functions that are mapped to a region, only one function can be in the region at any point in time. When a function is called, its region is checked; if it contains some other function, then the code of the new function must be brought into the SPM (through DMA), otherwise, the execution can continue. The state of the SPM, specifically which functions are present in the regions, must be maintained as a metadata. This metadata must be updated as new functions are brought in. Access to metadata can be frequent, and therefore it will be beneficial to implement metadata storage and updates in hardware, rather than in software data structures. This will further reduce the cost of each instrumentation, making the implementation more efficient.

Prefetching in Hardware.

Prefetching is a general technique to reduce miss penalty in caches. With language support for emerging memory technologies, it will be possible to design reconfigurable domain specific hardware prefetchers for the supported memory features. A careful coordination of compiler and hardware prefetching mechanisms will significantly reduce data access latencies. A particularly interesting combination is the creation of a virtual scratchpad for caching metadata, and asynchronous transfers for populating the scratchpad with metadata needed to enforce control flow integrity (CFI). This would enable per-function CFI enforcement, without latency at function boundaries. Coupling this technique with complex strided prefetch, we may even be able to fetch the CFI information for a complete subgraph of a control flow graph, and then validate CFI for the whole subgraph. Naturally, these more aggressive prefetch techniques introduce new heuristics, since some of the fetched data may not be used. Introducing priority within the strides being prefetched, or pipelining techniques, can alleviate these problems. And while we

have focused this discussion on function-related metadata, the same techniques could be used to fetch the code of functions to execute on a scratchpad, metadata for a strided access within a transaction, or speculative fetching of metadata before pointer chasing.

7. CONCLUSIONS

Innovations at the interface between microprocessors memory are becoming increasingly important. On the one hand, emerging applications are continuously requiring richer features to be provided by the system, such as fine-grained isolation and atomicity; on the other hand, hardware innovations are constantly enabling new capabilities, such as byte-addressable persistent memory, scratchpad memories, and hardware-enforced data confidentiality.

We argue that programming languages must have a unified memory interface that can support these memory features. Furthermore, the interface must also be extensible, to support new semantics required by programmers, new capabilities enabled by hardware, or new memory features that do not exist yet. A unified interface will enable programmers to start exploring hardware features before they become available; it will enable hardware vendors to gain users for new features without having to make significant investments into compiler infrastructure; and it will enable researchers to explore means for composing multiple memory features, so that programmers can enrich their programs through the use of memory technologies that lead to less code, more secure code, code that can be more easily proven correct, and code that uses less energy while completing in less time.

As a step toward this goal, we introduced language extensions that allow programmers to request memory features in both data and code regions, and we demonstrated an example use case in social graphs. We showed that this language design can be simple and flexible. We also discussed potential optimizations of compiler supports and hardware accelerations. We believe that the proposed language will have a game-changing impact on both hardware and software designs.

Acknowledgments

This material is based upon work supported by the National Science Foundation at Arizona State University under Grants CCF 1055094 (CAREER), CCF-0916652, and CNS 1525855; at Pennsylvania State University under Grant CCF-1149211 (CAREER), and at Lehigh University under Grant CCF-1253362 (CAREER). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *Proc. of CODES+ISSS*, pages 73–78, 2002.

[3] D. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 29th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Amsterdam, The Netherlands, Oct. 2014.

[4] 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*, Newport Beach, CA, Mar. 2011.

[5] Everspin Technology. Ddr3 dram compatible mram - spin torque technology. http://www.everspin.com/ddr3-dram-compatible-mram-spin-torque-technology.

[6] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[7] IBM(R). *Power ISA(tm) Transactional Memory*, 2.07 edition, Dec. 2012.

[8] Intel Corporation. 3D XPoint^{TM} Unveiled—The Next Breakthrough in Memory Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html.

[9] Intel Corporation. Intel Software Guard Extensions (Intel SGX), 2015. http://https://software.intel.com/sites/default/files/332680-002.pdf.

[10] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[11] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.

[12] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[13] Micron Technology. Micron announces availability of phase change memory for mobile devices, 2012. http://investors.micron.com/releasedetail.cfm?releaseid=692563.

[14] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[15] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.

[16] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security (CCS)*, pages 29–40, 2011.