LLAMA - Automatic Memory Allocations

An LLVM Pass and Library for Automatically Determining Memory Allocations

Derrick Greenspan
University of Central Florida
Orlando, Florida
derrick.greenspan@knights.ucf.edu

ABSTRACT

There is an expectation among experts that the memory hierarchy will be expanded to provide support for multi-level memory, enabling the use of new memory technologies living side-by-side with traditional DRAM. In doing so, the advantages of traditional DRAM can be retained alongside the advantages of these new memory technologies. However, in order to be effective, memory must be allocated to different memory levels, either manually by the programmer, or automatically.

This paper introduces a novel process which combines a custom LLVM Pass with a custom C library to automatically handle memory allocations performed by function calls without the need for programmer input or hardware/OS level changes. When utilizing a simulated multi-level memory architecture with dual non-volatile RAM and volatile DRAM, the results demonstrate that such a program can alleviate the burden on the programmer while still maintaining the performance advantages of DRAM and the higher density of non-volatile memory technology.

CCS CONCEPTS

• **Software and its engineering → Memory management**; **Allocation / deallocation strategies**; **Compilers**; **Source code generation**;

KEYWORDS

Compiler Construction, Source code Generation, Multi-Level Memory, Memory Allocation/Deallocation Strategies, Memory Management

ACM Reference Format:
Derrick Greenspan. 2019. LLAMA - Automatic Memory Allocations: An LLVM Pass and Library for Automatically Determining Memory Allocations. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357526.3357534

1 INTRODUCTION

Since the introduction of dynamic random-access memory (DRAM) in the 1970s, the memory hierarchy has generally had a single level of main memory. Today, however, there are several different memory technologies which look poised to replace DRAM long term, as they have performance close to or exceeding DRAM[10], or they have higher capacity and non-volatility[3, 10, 21, 35, 38]. However, in the short term, all of these memory technologies will likely serve as an additional layer in the memory hierarchy – in some cases, they will be below traditional main memory such as DRAM, in others, they may be above.

In order to best take advantage of multi-level memory technologies, programmers are expected to manually specify that a particular write be done in one level of memory or the other. For example, the Intel libpmem library[32], which supports non-volatile memories, allows the programmer to indicate if a particular dynamic memory allocation ought to be committed to non-volatile or volatile storage. However, there is a better method – rather than relying on the programmer to make a decision in advance as to where memory is stored, memory allocation functions can instead be intercepted and redirected to whichever memory level is most efficient as determined by the compiler. With a library and compiler working in tandem to perform these calculations as needed, the programmer can be alleviated of much of the difficult work in determining where dynamic memory allocations performed by functions such as `calloc()`, `malloc()`, and `realloc()` should ultimately perform their allocations.

This paper introduces LLAMA ("LLVM Pass (and) Library (for) Automatic Memory Allocations"), a new method for automatically placing memory allocations for multi-level memory systems in the correct memory levels. LLAMA combines a custom LLVM compiler pass[2, 19, 20, 33] with a custom C library which intercepts memory allocations, determines where the data should be allocated based on information determined by the compiler pass, and then performs the memory allocation. The goal of LLAMA is to enable multi-level memories while requiring as little user intervention as possible. By simply running the pass and including the library, LLAMA is able to perform memory allocations where needed without disruptive hardware or software level changes.

This work focuses on a two-level memory system comprised of conventional volatile dual data rate dynamic random-access memory (DDR DRAM), and "slow" non-volatile memory (NVRAM). This is likely to be the primary use of this technology for the near term and is elaborated in the "background" section below. Other scenarios, where the "slow" memory is DDR and the fast memory is a less dense but faster memory technology, is also briefly discussed, and is an avenue for future research[10].

The rest of this paper is organized as follows: Section 2 provides a brief background on historical and contemporary memory systems as they relate to the goals of LLAMA, and describes a hypothetical multi-level memory hierarchy. Section 3 provides an overview of related work in multi-level memory allocations. Section 4 describes the implementation of the pass and the library, while section 5

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA*, https://doi.org/10.1145/3357526.3357534.

describes a simulated multi-level memory system with LLAMA and provides performance results with an analysis. Section 6 discusses possible avenues of future work, and Section 7 concludes.

2 BACKGROUND

This section discusses main memory from an historical perspective, and provides an overview of new emerging memory technologies.

2.1 History of Main Memory Technologies

Since the popularization of metal–oxide–semiconductor field-effect transistors (MOSFETs) in the 1970s, dynamic random-access memory (DRAM), which is volatile, cheap, and of high density[14, 18], has been the most common type of main memory[6, 13, 14] used in computers. In contrast, static random-access memory (SRAM), which is extraordinarily fast but not dense, is used as CPU cache[22]. However, before DRAM and the advent of MOSFET technology, ferrite magnetic core memory (usually simply referred to as "core memory")[28], was most commonly used. Core memory is similar to DRAM in that it is random-access, however, unlike DRAM it is partially non-volatile and does not require any type of memory refresh[6, 23, 28].

Today, manufacturers are hitting a limit on scaling DRAM cells to smaller size[15]. In particular, when a DRAM cell is scaled down, it becomes harder to maintain the same capacitance of the DRAM cell[18, 25]. This results in higher failure rates of DRAM memories[18, 25] as the size of the DRAM cell decreases.

2.2 Emerging Memory Technologies

Effective and long-term solutions to the problem of high DRAM failure rates due to DRAM scaling may prove to be intractable[3, 18]. Because of this, main memory may once again transition to a new technology. In particular, there have been two classes of new memory technologies that are emerging which do not suffer from the same scaling issues. They are the non-volatile memory technologies, and the "fast" memory technologies. Both types of memory technologies have the potential to replace DRAM.

Non-volatile memory technologies (NVM) retain data upon power loss, and include Phase Change Memory (PCM)[18, 21], Memristor based Resistive RAM (RRAM)[18], Ferroelectric RAM (FeRAM)[18], Magnetoresistive RAM (MRAM)[3, 18, 22, 35, 38], Spin-Transfer Torque RAM (STT-RAM)[8], Intel's Optane Memory[4, 35], and others. Non-volatile memory tends to be slower than traditional DRAM[7, 22], but performance of non-volatile memory systems are still much closer to DRAM than to traditional block devices such as hard disks, which are orders of magnitude slower[9, 11].

"Fast" memory technologies include Zero capacitor RAM (Z-RAM)[27], Twin-Transistor RAM (TTRAM)[1], A-RAM[31], ETA RAM[22] as well as Hybrid Memory Cube (HMC)[18], and High Bandwidth Memory(HBM)[10]. These technologies are often faster and have orders of magnitude higher bandwidth than DRAM, but have smaller memory densities. In general, these technologies cost about $2-5$ times more per bit compared to traditional DRAM[16].

Just as DRAM replaced magnetic core memory, computer engineers hope that one of these technologies will eventually be able to satisfy all of the requirements of an "ideal memory", such as low-power, high-density storage, high speed, good endurance, and low cost[18]. If this happens, one of these technologies could ultimately completely replace DRAM as main memory in the long term. Regardless of whether such an ideal memory comes to fruition, in the short term, their potential is in complementing DRAM[35]. Therefore, in the short term, these technologies will only be useful to users if it does not require programmers to manually go through legacy code and add support for multi-level memory. This paper focuses on multi-level memory systems with two memory levels, but multi-level memory systems can have more than two levels. For example, it is possible to have "fast" memory, DRAM, and non-volatile memory together, in an attempt to get the "best of both worlds"[16].

2.3 A Hypothetical Memory Hierarchy

Figure 1 illustrates a hypothetical memory hierarchy with capacities and access times. The figure illustrates two levels of primary memory, a faster DRAM, and a slower but larger non-volatile random-access memory technology. Notice how the NVRAM is 32 times larger than DRAM, while only 3 times slower. In this diagram, the hypothetical NVRAM is able to surpass the scaling limit of DRAM and have much higher densities without a significant loss in performance. In addition, NVRAM, being non-volatile, can also act as secondary memory for longer term storage. This paper utilizes LLAMA on a hypothetical system with a similar hierarchy.

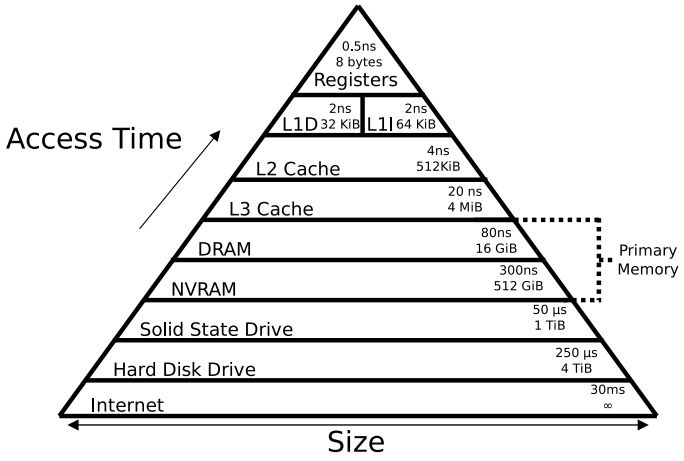

Figure 1: A hypothetical computer memory hierarchy with one level of non-volatile main memory and one level of volatile dynamic random-access memory. The access latency times are rough estimates.

3 RELATED WORK

There has been much work on cache policies and memory paging schemes, but while it is possible to treat one level of memory as a cache and another level of memory as memory (or, to treat one level of memory as traditional memory, and another as a backing store), multi-level memories do not have a performance difference as dramatic. Thus, these applications are not directly appropriate for multi-level memory systems.

While there has been a fair amount of work on multi-level memories, including replacement and addition policies for these systems[10], all require expensive hardware changes and additional hardware controllers for automatic allocations. Alternatively, there has been research in best practices for performing manual allocation[39], but this does not satisfy the requirement of alleviating the burden on the programmer. There has been work on a portable run-time interface for multi-level memory systems[12], but this requires the programmer to manually specify through an API which memory level they wish to write to. Other research, such as those done by Chen, et. al.[7], focuses on reducing writebacks from cache to non-volatile memories in order to counteract the issues of slow writes and limited write endurance. However, their research does not focus on multi-level designs.

There has been a modest amount of research on compiler support for automatically determining where any particular allocation of data should be written to without the involvement of expensive hardware changes. Khaldi and Chapman, in their paper "Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing"[17], describe a method of determining whether or not a particular set of data should be allocated to High Bandwidth Memory (HBM) or to DRAM, utilizing LLVM to modify source code in a method similar to LLAMA, however, they do not consider the size of the underlying variable nor the size of each memory level when determining where their variables should be allocated, instead they attempt to determine whether or not the particular set of data is bandwidth critical, and if it is, they then attempt to allocate the data to the HBM, without regard to the size of the data or even if such an allocation is possible. In addition, their work is specific to the Intel Knights Landing Architecture and their performance analysis only involves one use case.

Hammond, et. al., in their paper "Multi-Level Memory Policies: What You Add Is More Important Than What You Take Out"[10] describe hardware multi-level memory units responsible for deciding where memory pages should be placed, as compared to user-directed data placement, which they identify as imposing a heavy burden on the programmer. Although in the long term, such specialized hardware memory units will become the primary method of integrating multi-level memory systems, they do not propose a solution for current multi-level memory systems which do not utilize this specialized hardware and require software level solutions. For these systems, function calls based on the particular implementation must be used, for example, with Intel's libpmem[32], or the portable run-time interface for multi-level memory systems described above[12]. Rather than implementing these function calls manually, these systems can utilize LLAMA instead.

4 DESIGN AND IMPLEMENTATION

This section describes the two components which make up LLAMA, and goes into further detail about how each component works.

The main goal of LLAMA is to automatically allocate without programmer intervention larger data that is not often used in the "slow" memory, and to allocate smaller data or data that is used often in the "fast" memory. In order to do this, a compiler must perform analysis on the particular memory allocation to determine how often it is used, and pass this information along to a library which intercepts memory allocation calls. Therefore, the design behind LLAMA involves two components:

- The LLVM Pass portion is used to give a score to each memory allocation, based on the depth of the loop it is in, and how many instructions there are between two separate memory operations.
- The Library portion is used to compute *ad hoc* whether or not a particular memory allocation should occur in one memory level or another.

The library uses the data collected by the pass in making a determination of which memory level the data should be allocated to, and in turn the pass looks for the variable names declared by the library. A diagram depicting how the two parts work in tandem is found in figure 2.

Memory allocations that are not allocated via a memory allocation call, such as fixed-length arrays, variable-length arrays, or variable declarations, are not affected by LLAMA. They will be allocated to whatever the default memory level is, generally level 0.

It is important to note that LLAMA does not support whole-program analysis. LLAMA assumes that all child libraries are compiled so that all memory functions can be analyzed by the LLVM pass and intercepted by the library. This is because LLAMA's library portion can only operate if it already intercepts the memory allocation functions. However, LLAMA can support precompiled headers, so long as the header is compiled with LLAMA's library and pass.

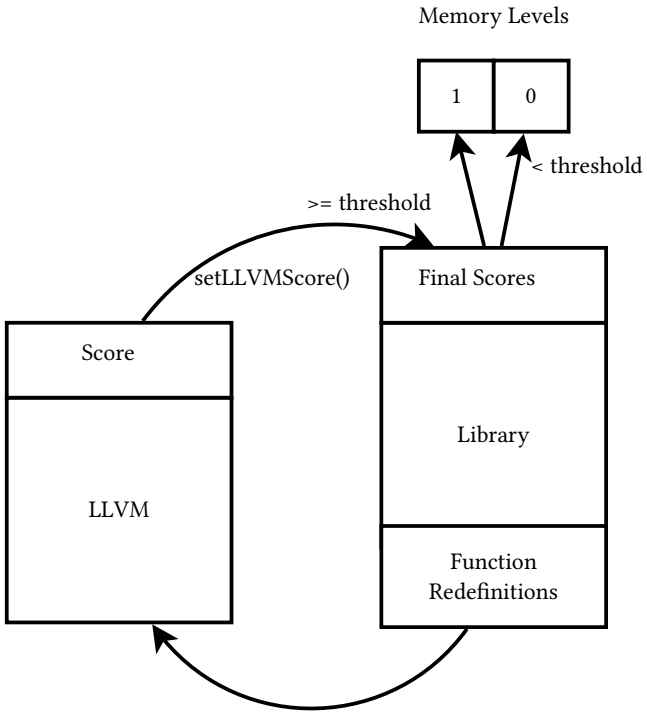

Figure 2: A broad overview of the two components of LLAMA

4.1 LLVM Pass

The LLVM Pass portion of LLAMA is its unique contribution. It is responsible for creating a score for each memory allocation invoked by `calloc()`, `malloc()`, or `realloc()`.

LLVM compiles to an *intermediate representation*, also known as *LLVM IR*, which is an idealized RISC architecture with an infinite number of registers[33]. This representation is what the LLVM pass analyzes and modifies.

The LLVM Pass described in this work has two steps: the analysis phase, and the modifying phase. The analysis phase is responsible for analyzing the code and scoring the individual memory allocations, while the modifying phase is responsible for informing the library of the LLVM score.

The entire pass happens after the LLVM common optimizations, specifically, it is the last step after loop rotation. This is done in order to allow the pass to have an accurate analysis of the depth of the loop (the common optimizations perform analysis which provides information such as the loop depth[1], and other optimizations can cause this depth to change), and to prevent the common optimizations from eliminating the changes performed by the modifying phase.

4.1.1 ***Analysis Phase.*** The *Analysis Phase* of the LLVM Pass scans each function of the pass and searches for the memory allocation function calls. When this happens, it adds both the location of the function call and the instruction itself to the `values` list, in preparation for the modifying phase.

It then looks for an LLVM `StoreInstruction`, and verifies that the `StoreInstruction`'s pointer operand is the instruction from a previous memory allocation function call by searching the list of saved memory allocations. In the case that the value operand is a `BitCastInstruction` (which will be the case if the variable's type larger than 8 bits) it compares the first operand of that instruction to the value saved in the list. In doing so, only those `StoreInstructions` which were invoked from a memory allocation call are considered.

Next it looks at the pointer operand of the `StoreInstruction`, which is the reference to the memory allocated by the function call. It is *this* value that other instructions refer to when they refer to data allocated by these calls. It checks to see the number of times the value is used, which is part of how the LLVM Pass generates a score. The more times the value is used (**T**), the more important it is that the memory is allocated in the lower memory level (the "fast" memory).

Then the pass gets every instruction which uses the pointer operand and iterates through them. For each, it checks the depth of the loop via the LLVM `getLoopDepth()` function. It is assumed that accesses inside a loop are more important than accesses outside of a loop, and therefore accesses inside a loop should be weighed more heavily as part of the LLVM score.

However, a large number of operations between accesses should decrease the importance of the access. Thus, for memory accesses in loops, each time an instruction (**inst**) between memory accesses in each basic block is found, the initial weight of 15 is decremented by one.

[1]The **loop depth**, which refers to the level of nesting the loop is in, is not to be confused with the **loop trip count** for each basic block, which refers to the minimum number of times a loop will execute.

```
%13 = add nsw i32 %12, 2
%14 = load i32*, i32** %3, align 8
%15 = load i32, i32* %2, align 4
%16 = sext i32 %15 to i64
%17 = getelementptr inbounds
        i32, i32* %14, i64 %16
store i32 %13, i32* %17, align 4
%br label %18
```

Figure 3: An example of LLVM IR used in referencing data from an array

The final weight of each individual instruction which is inside a loop and referencing the pointer operand is the difference between 15 and the number of instructions between each reference to the pointer operand raised to the power of the depth of the loop multiplied by 2. Mathematically, it is $(\mathbf{15 - inst})^{(\mathbf{2 \times depth})}$. However, if the difference is less than 0, then the weight is 0.

Figure 4 depicts the full formula used by the LLVM pass to calculate its own score for each memory allocation. This score is then used by the library at runtime as described in figures 6 and 7.

$$score = T + \sum_{n=1}^{T} \begin{cases} (15 - inst_n)^{(2 \times depth_n)} & \text{if } 15 - inst > 0 \\ 0 & \text{if } 15 - inst_n \leq 0 \\ 0 & \text{if } depth_n = 0 \end{cases}$$

Figure 4: The LLVM pass's score of each memory allocation. T is the total number of instructions which refer to the pointer to the memory allocation.

4.1.2 ***Modifying Phase.*** The score calculated in the analysis phase is used in the modifying phase to insert the "`setLLVMScore()`" function call. In particular, the LLVM pass iterates through every value stored in the `values` list, and sets the IRBuilder's insertion point to the location of the function call. At this point, the LLVM Pass performs its only modification: it inserts the `setLLVMScore()` function call directly above the internal memory allocation call, along with the calculated score.

4.2 Library

The library portion of LLAMA is relatively simple and straightforward, providing a degree of dynamic analysis that otherwise would not be possible with the LLVM pass alone. It utilizes the Structural Simulation Toolkit which is described in section 5. In particular, it utilizes the Ariel core (also described in section 5), which has support for multi-level memories.

4.2.1 ***Function Intercepts.*** The library consists of a header file (`intercept.h`) which redefines the traditional memory allocation functions `malloc()`, `calloc()`, and `realloc()`, as well as the traditional memory deallocation function `free()`. These function definitions are redefined to the internal memory allocation functions `_internal_malloc()`, `_internal_calloc()`, and

_internal_realloc() which the Structural Simulation Toolkit (described below) looks for.

4.2.2 ***The mlm_malloc() function call.*** Although the goal of LLAMA is to be useful in production environments, for the purposes of benchmarking, the library portion of LLAMA is currently designed to operate with the Structural Simulation Toolkit as a proof of concept. The ultimate goal is to have LLAMA be modular enough so that it is agnostic to the function calls used by different memory libraries, such as libpmem[32] or Hexe[26], and therefore can call different allocation functions based on the multi-level memory library to be used.

The Structural Simulation Toolkit includes the Ariel element, which is a processor core emulation component that dynamically streams instructions from a running application (via Intel PIN) and intercepts memory read/write requests[34] which are then forwarded to the memHierarchy element. The memHierarchy element simulates an intra- and inter-node directory-based cache coherency architecture[24], which is used to model different levels of the memory hierarchy, such L1, L2, and memory. The Ariel element also searches for certain functions, which tell Ariel to perform certain actions or do certain things. For example, `ariel_enable()` tells Ariel to enable memory tracing and interception. The function: `mlm_malloc(size_t,int)`, tells Ariel to allocate the particular block of data to whichever memory level is specified by its second argument. This work's particular usage of the Structural Simulation Toolkit is explored in more detail in the next section, "Performance Analysis".

4.2.3 ***Internal Memory allocation calls.*** The `mlm_malloc()` call only simulates a traditional `malloc()` function call. Therefore, to implement the internal `calloc()` and `realloc()`, they are re-implemented by the library. The internal memory allocation calls provide for a degree of dynamic analysis. In particular the library is able to tell at runtime how much data is to be allocated based on the arguments passed into `calloc()`, this is then used to make a final determination of which memory level the data should be allocated into, as described in the "Allocation Calculation" section below.

The internal `calloc()` function call performs a `memset()` operation which sets every value in the allocated memory to zero immediately after the `mlm_malloc()` call. The source code for this function is found in Figure 5. The internal `realloc()` function call performs a `memcpy()` operation, which copies the value in the old memory address to the value in the new memory address immediately after the `mlm_malloc()` call. After this call, the old memory address is freed with `_internal_free()`.

4.2.4 ***The setLLVMScore() function call.*** The LLVM pass inserts the function call `setLLVMScore(int thisScore)` immediately before the memory allocation function calls. A variable called LLVMScore is set based on the value of `thisScore`. This value is used in the allocation calculation below.

4.2.5 ***Allocation Calculation.*** The library performs a calculation based on the score given by the LLVM pass. The higher the score, the more likely that the data is allocated to the first memory level as described above. The score calculated by the LLVM pass is computed as part of a formula, shown in figure 6. The final score is

```
void * _internal_calloc(size_t nitems,
        size_t size)
{
    int level =
        _which_level(size * nitems,
        threshold);
    void *return_ptr =
        mlm_malloc(size*nitems, level);
    return_ptr =
        memset(return_ptr, 0, size);
    return return_ptr;
}
```

Figure 5: The source code for the *_internal_calloc*() function call in LLAMA's library component

the ratio between the size of the data to be allocated and the score given by the LLVM pass, multiplied by the ratio between the sizes of each memory level.

$$\textbf{finalscore} = \frac{\textbf{datasize}}{\textbf{LLVMscore}} \times \frac{\textbf{level0size}}{\textbf{level1size}}$$

Figure 6: A formula for calculating the final score for a particular memory allocation

The final score is then compared to a threshold. If the final score is greater than or equal to the threshold, then the write goes to the faster memory (level 0). If the final score is less than the threshold, the write goes to the slower memory (level 1). The entire formula for determining which level a particular memory allocation should be assigned is described in figure 7.

$$\textbf{level} = \begin{cases} 1 & \text{if } \frac{datasize}{LLVMscore} \times \frac{level0size}{level1size} \leq threshold \\ 0 & \text{if } \frac{datasize}{LLVMscore} \times \frac{level0size}{level1size} > threshold \end{cases}$$

Figure 7: The formula LLAMA uses to calculate which level a particular memory allocation will go to

5 PERFORMANCE ANALYSIS

The different threshold levels were tested using the Structural Simulation Toolkit (SST)[29, 30]. SST was configured to model four CPU cores with a private L1 and L2 Caches. Each L2 cache connects to an on-chip router. For simplicity, there is no L3 cache. The on-chip router connects to the separate directory controllers. Each controller connects to its own memory, either DRAM or non-volatile memory. The simulation model is depicted in figure 8.

This simulation model provides for simulating the memory hierarchy, and latencies between the different levels of the hierarchy. SST's memHierarchy, Merlin, and Ariel Components were used for the caches, on-chip network, and CPU cores, respectively. The simulation parameters and specified latencies can be found in table 1.

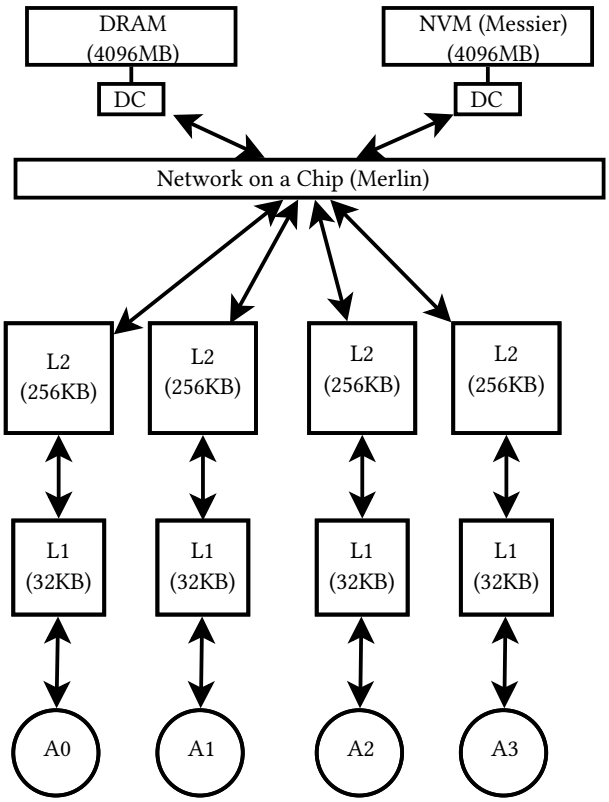

Figure 8: Proposed architecture and simulation model

Component	Parameters
Core	2 GHz, 3 issue / cycle, L1 and L2 per core
Coherency	MESI Protocol
L1	32KB, 8-way, 64B cache line, 4 cycles
L2	256KB, 8-way, 64B cache line, 10 cycles
Router	320GB/s, 72B flit, 4ns latency
DRAM	2GHz clock, 4096 MiB, DDR3
NVM	250MHz clock, 60ns access time, 4096MiB

Table 1: Simulation Parameters

5.1 Methodology

To test the effectiveness of LLAMA, three separate benchmark tests were run. Because the LLVM Portion of LLAMA currently does not support C++, all of the benchmarks tested are written in C.

The first benchmark is a custom simple memory allocation and operation test. This test performs memory operations that would be expected in common programs. A more thorough description is provided in the subsection below.

Because this paper forsees LLAMA being useful in high performance computing (HPC) systems with multi-level memory, the second and third tests are U.S. Department of Energy (U.S. DoE) miniapps: the XSBench Miniapp[37] and the SimpleMOC kernel[36].

5.2 Memory Allocator

This memory allocation test performs memory operations that mirror memory read/write patterns that are typical of common programs. In particular, the program performs a `malloc()` and `calloc()` system call on two pointers, allocating a set number of integers as determined by the program's first argument. The following memory read/write patterns are performed:

- Looping through the entire array and assigning each item a value.
- Performing a memory test on the entire array and checking if each value has been set as expected.
- Setting one array equal to another.
- Performing many operations between two memory accesses. Figure 9 depicts the source code for this particular operation.

The arguments passed to this application are found in table 2.

```
int l = 0;
/* Perform interleaving accesses */
for(i = 0; i < SIZE; i++)
{
        var1[i] = l;
        int j;
        for(j = 0; j < 1; j++)
        {
                int k;
                for(k = 0; k < 5; k++)
                {
                        l = j * k;
                }
        }
}
```

Figure 9: A segment of the source code for the memory allocator benchmark illustrating a loop. The variable *var1* was allocated earlier. Since there are a fair number of operations between accessing *var1* and the end of its basic block, the LLVMscore is lower than it otherwise would be.

5.3 Matrix Multiplier

The matrix multiplication benchmark allocates (via `calloc()`) two matrices of a size specified by standard input. The program then assigns a value to each item in each matrix. To avoid overhead from randomization operations, the program assigns a value to each item in the array in a deterministic fashion[2]. From there, the benchmark performs a multiplication operation on the two matrices, which creates a nested loop of depth two. The arguments passed to this application are found in Table 2.

5.4 XSBench

The XSBench Miniapp is an HPC miniapp which represents the key computational kernel found in the Monte Carlo neutronics application OpenMC[37]. The arguments passed to this application are found in Table 2.

[2]The value of each item in the array is $(row + column)$ mod 10.

5.5 SimpleMOC Kernel

The SimpleMOC-kernel miniapp represents the core computational kernel found in SimpleMOC. It performs the inner-loop of the larger SimpleMOC miniapp, that is, it is the attenuation of neutron fluxes across an individual geometrical segment. This kernel composes approximately 92% of the walltime of the full application[36] and thus is an accurate benchmark for SimpleMOC overall. The arguments passed to this application are found in Table 2.

Table 2: Application Parameters

Application	Options
Memory Allocator	100000
Matrix Multiplier	100 100 100 100
XSBench	-g 2 -p 500
SimpleMOC Kernel	-s 3 -e 3

5.6 Results

The simulations in this paper ran with two different thresholds. To establish a baseline, the simulations also test the performance of purely level 1 non-volatile memory writes, and purely level 0 DRAM. The performance of the two benchmarks are compared to different thresholds of varying levels. All the performance results can be found in figure 10.

The figure 10 results show a steady increase across the memory-heavy benchmarks (Memory Allocator and the two DOE benchmarks) in execution time as the threshold for saving data in the faster level 0 memory is raised. As more and more memory allocations are placed in the slower non-volatile memory, the execution time increases. Note however that the choice of allocation is binary – for every allocation it will either be entirely placed in the level 0 memory, or entirely in the level 1 memory. This is why the performance appears to rapidly drop off at the higher threshold levels.

5.7 Analysis

At first glance, the Matrix Multiplier benchmark exhibits counter-intuitive behavior - in particular, performance does not change based on threshold. This can be explained by the relatively small amount of memory being allocated, as well as the fact that most of the execution time is spent performing integer multiplication.

For the other three benchmarks, the results occur due to the memory heavy nature of the application. Of particular interest is what threshold a programmer using LLAMA would want to use.

For the SimpleMOC-Kernel, the threshold would be about **10000**. This is because the performance is degraded only 10% compared to purely allocating in DRAM, yet, most allocations are placed in the non-volatile memory. When the results are examined more closely by looking at the standard error (stderr) debugging outputted by the library, it is found that the vast majority of data can be stored in the non-volatile memory without affecting performance. Only the three largest memory allocations appear to affect performance at all – and these are allocations that with the specified arguments, are two 270KB allocations and one 54KB allocation. At a threshold of 10000, only the small allocations and the third largest allocation is impacted – the pass gives this allocation a score of 5400, which is why performance does not appear to change in a significant manner.

For XSBench, the threshold is **20000**. An examination of stderr illustrates that the largest individual memory allocation is about 1MB. Since this allocation is used quite often, the score ends up being **29652** – which is why performance dramatically declines once the 30000 threshold is reached. The rest of the memory allocations are placed in NVM because they have low scores, these do not appear to affect performance to a significant degree.

For the Memory Allocator, the correct threshold when aiming for performance is also **10000**. However, an examination of stderr shows that both memory allocations occur in DRAM, which is why there are no actual performance differences between DRAM and that threshold. At **20000**, a 30% performance loss is observed. Upon inspection of stderr, the reason is evident: the first memory allocation (the `malloc()`) is given a score of 11475, and it is 2.8MB large – thus, at a threshold of **20000**, that allocation is put in the first memory level. Since it is heavily used, this degrades performance dramatically. Figure 11 gives part of the stderr output for one Memory Allocator run, where the threshold is **40000**.

6 FUTURE WORK

Although running these benchmarks demonstrated that LLAMA is capable of alleviating the burden of manually deciding which memory level to allocate to, the memory footprints are small and thus do not perfectly represent real-world scenarios. Any future work should address this by increasing the memory footprints of each benchmark such that it will not fit solely in level 0. These results could then be compared to the results of a simple size threshold policy such as provided by Intel's memkind. As LLAMA's analysis is much more complicated than memkind, in theory it should be more effective, but research is required to demonstrate this.

Another avenue for future work is to extend LLAMA to support additional memory levels, rather than the dual memory levels that LLAMA currently supports. Theoretically this would merely require adding additional thresholds, that when met, places the data in a higher memory level. This would allow LLAMA to work with, for example, a three level memory system with "fast" memory as level 0, traditional DRAM as level 1, and non-volatile memory as level 2. This is relatively trivial to implement and is the most obvious avenue of future work.

Another option for future work would involve extending LLAMA to automatically determine the proper threshold for a particular program, so that the programmer does not need to decide the proper threshold themselves. This would be complicated – it would likely still involve the programmer telling LLAMA how much memory is being used by the program, as it's not possible to determine that before runtime. A profiling run which performs dynamic programming analysis could help.

Yet another avenue is to extend LLAMA to work with Intel's libpmem[32] library, which would allow implementing LLAMA on actual hardware. This would require also extending LLAMA to support memory mapping, and performing it whenever a memory allocation is performed. Such an extension could then be used to

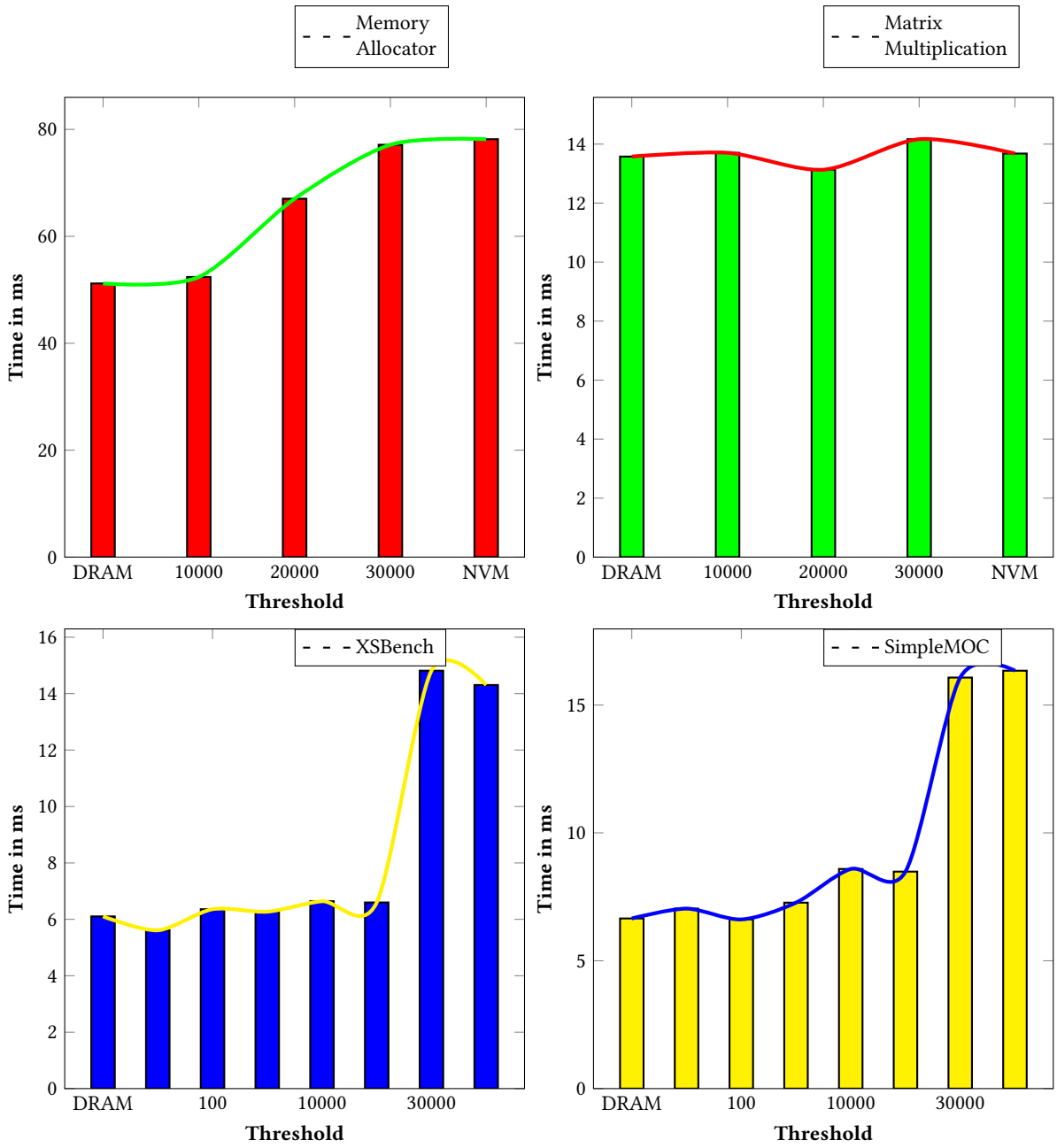

Figure 10: Performance by threshold of each benchmark. Note that the appropriate threshold to balance performance while maximizing memory allocations to the non-volatile store vary based on benchmark.

allow users to painlessly convert their source code to run on multi-level memory systems.

Another is to allow fixed-length arrays, variable-length arrays, and variable declarations (i.e., those arrays which are allocated on the stack) to be written to different memory levels in the same way memory allocations which are dynamically allocated with `malloc()`, `calloc()`, `realloc()`, (i.e., on the heap) are. This would require transforming these array allocations on the stack into dynamic memory allocations on the heap, which could impact performance, as normally the stack is much faster than the heap.

The algorithm could also be extended to support read/write leveling, having data which is read-heavy in the upper memory (non-volatile) level, with data which is read and write heavy in the lower memory (DRAM) level. This can be useful in certain situations such as in bioinformatics applications because most proposed forms

```
ARIEL: Enabling memory and instruction
tracing from program control.
Data size: 2800000, LLVM Score: 244
Threshold is 40000, score is 11475
0: Perform a mlm_malloc from Ariel 2800000,
level 1
Requested: 2800000, but expanded to: 2801664
(on thread: 0)
0: Ariel mlm_malloc call allocates data at
address: 0x2b8cf40b7000
Data size: 2800000, LLVM Score: 103
Threshold is 40000, score is 27184
0: Perform a mlm_malloc from Ariel 2800000,
level 1
Requested: 2800000, but expanded to: 2801664
(on thread: 0)
```

Figure 11: Standard-error for the Memory Allocator, at a threshold of 40000 (always NVM). Notice that the LLVM Score is 244 **and** 103**, and the data size is** 2800000 **bytes, which makes the final scores** $\frac{2800000}{244}$ = 11475 **and** $\frac{2800000}{103}$ = 27184 **respectively.**

of non-volatile memory has a slower write speed than read speed and suffer from a limited write-cycle capacity. This algorithm would require counting the number of times the pointer is written in addition to counting the number of times the pointer is used, and increasing the score appropriately as the ratio between the writes and total number of accesses increases.

Yet another is to support C++ and object oriented languages which LLVM supports. For memory allocation, allocating objects with the "new" keyword is the most important difference from imperative languages. In LLVM bytecode and x86 assembly, the function which performs the *new()* operator is `Znwm()`[5], it is foreseeable to have LLVM search for the `Znwm()`function call and then insert an `mlm_malloc_flag()`directly above, or alternatively to crate an equivalent function call which performs the same operations as the `new()` operator, but works with SST.

Finally, one last avenue for future work is to allow splitting memory allocations between different memory levels, such that one part of the data is allocated in one memory level, and the other part is allocated in the other. This would be done in order to store large blocks of data that will not fit in one particular memory level, and would essentially treat the lower memory levels as cache.

7 CONCLUSION

As multi-level memory become more common, developing tools to automatically manage data placement during dynamic allocation will become more important. Programmers cannot be expected to manually specify which memory level any particular data should be allocated to, and hardware changes to automatically determine which memory level data should be written to are often cost prohibitive.

To that end, this paper has presented a novel tool for performing these memory allocations without burdening the programmer – all the programmer must do is specify the desired threshold, link their program to the LLAMA library, add the llama header, and then run the optimization pass via clang. Everything else is done for them.

The initial performance analysis of LLAMA demonstrates that each application needs to have its own threshold specified, and that the algorithm works properly when determining whether to allocate any particular set of data in one memory level or another. While a programmer manually specifying which level of memory they wish to allocate a particular set of data to will generally be more efficient then relying on this tool, this tool works well for large programs and programs that have already been written, and where slightly worse performance is acceptable because the changes required to support multi-level memory would otherwise be too prohibitively large.

ACKNOWLEDGMENTS

The author acknowledges the University of Central Florida Advanced Research Computing Center for providing computational resources and support that have contributed to results reported herein. URL: https://arcc.ist.ucf.edu.

The author also acknowledges Adrian Sampson for providing his LLVM pass skeleton under the MIT license, which proved to be an invaluable reference when designing the LLVM portion of LLAMA. URL: https://github.com/sampsyo/llvm-pass-skeleton.

REFERENCES

[1] Arimoto, A. High-density scalable twin transistor ram (ttram) with verify control for soi platform memory ips, nov. 2007. *Solid-State Circuits*.
[2] Arthur Lattner, C. Llvm: An infrastructure for multi-stage optimization.
[3] Awad, A., Hammond, S., Hughes, C., Rodrigues, A., Hemmert, S., and Hoekstra, R. Performance analysis for using non-volatile memory dimms: Opportunities and challenges. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2017), MEMSYS '17, ACM, pp. 411–420.
[4] Branscombe, M. How intel's new optane persistent memory will change your data center, June 2018.
[5] Cao, Z. *Efficient Design and Implementation of Software Thread-Level Speculation*. PhD thesis, McGill University, 2015.
[6] Ceruzzi, P. E., Paul, E., et al. *A history of modern computing*. MIT press, 2003.
[7] Chen, D., Ye, C., and Ding, C. Write locality and optimization for persistent memory. In *Proceedings of the Second International Symposium on Memory Systems* (New York, NY, USA, 2016), MEMSYS '16, ACM, pp. 77–87.
[8] Chen, E., Lottis, D., Driskill-Smith, A., Druist, D., Nikitin, V., Watts, S., Tang, X., and Apalkov, D. Non-volatile spin-transfer torque ram (stt-ram). In *68th Device Research Conference* (June 2010), pp. 249–252.
[9] Feldman, J. M., and Retter, C. *Computer Architecture; A Designer's Text Based on a Generic RISC*. McGraw-Hill, Inc., 1993.
[10] Hammond, S. D., Rodrigues, A. F., and Voskuilen, G. R. Multi-level memory policies: What you add is more important than what you take out. In *Proceedings of the Second International Symposium on Memory Systems* (New York, NY, USA, 2016), MEMSYS '16, ACM, pp. 88–93.
[11] Heuring, V. P., Jordan, H. F., and Murdocca, M. *Computer systems design and architecture*. Addison-Wesley, 1997.
[12] Houston, M., Park, J.-Y., Ren, M., Knight, T., Fatahalian, K., Aiken, A., Dally, W., and Hanrahan, P. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 143–152.
[13] Itoh, K. The history of dram circuit designs – at the forefront of dram development –. *IEEE Solid-State Circuits Society Newsletter 13*, 1 (Winter 2008), 27–31.
[14] Jacob, B., Ng, S., and Wang, D. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.
[15] Jagasivamani, M., Walden, C., Singh, D., Kang, L., Li, S., Asnaashari, M., Dubois, S., Jacob, B., and Yeung, D. Memory-systems challenges in realizing monolithic computers. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2018), MEMSYS '18, ACM, pp. 98–104.
[16] Jayaraj, J., Rodrigues, A. F., Hammond, S. D., and Voskuilen, G. R. The potential and perils of multi-level memory. In *Proceedings of the 2015 International*

Symposium on Memory Systems (2015), ACM, pp. 191–196.

[17] Khaldi, D., and Chapman, B. Towards automatic hbm allocation using llvm: A case study with knights landing. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)* (Nov 2016), pp. 12–20.

[18] Kurinec, S. K., and Iniewski, K. *Nanoscale semiconductor memories: Technology and applications*. CRC press, 2014.

[19] Lattner, C. Llvm and clang: Next generation compiler technology. In *The BSD conference* (2008), pp. 1–2.

[20] Lattner, C., and Adve, V. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (March 2004), pp. 75–86.

[21] Lee, B. C., Zhou, P., Yang, J., Zhang, Y., Zhao, B., Ipek, E., Mutlu, O., and Burger, D. Phase-change technology and the future of main memory. *IEEE Micro 30*, 1 (Jan 2010), 143–143.

[22] Meena, J., Min Sze, S., Chand, U., and Tseng, T.-Y. Overview of emerging non-volatile memory technologies. *Nanoscale Research Letters 9* (09 2014), 1–33.

[23] Middelhoek, S., George, P. K., and Dekker, P. *Physics of computer memory devices*. Academic Press, Inc., 1976.

[24] Moore, B., Voskuilen, G. R., Rodrigues, A. F., Hammond, S. D., and Hemmert, K. S. Structural simulation toolkit. lunch & learn. Tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[25] Nair, P. J., Kim, D.-H., and Qureshi, M. K. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. *SIGARCH Comput. Archit. News 41*, 3 (June 2013), 72–83.

[26] Oden, L., and Balaji, P. Hexe: A toolkit for heterogeneous memory management. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)* (2017), 656–663.

[27] Okhonin, S., Nagoga, M., Carman, E., Beffa, R., and Faraoni, E. New generation of z-ram. In *2007 IEEE International Electron Devices Meeting* (2007), IEEE, pp. 925–928.

[28] Riley, W. B. *Electronic computer memory technology*. McGraw-Hill, 1971.

[29] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., Cooper-Balis, E., and Jacob, B. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev. 38*, 4 (Mar. 2011), 37–42.

[30] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., and Jacob, B. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev. 38*, 4 (Mar. 2011), 37–42.

[31] Rodriguez, N., Cristoloveanu, S., and Gamiz, F. A-ram: Novel capacitor-less dram memory. In *2009 IEEE International SOI Conference* (2009), IEEE, pp. 1–2.

[32] Rudoff, A. Persistent memory programming. *Login: The Usenix Magazine 42* (2017), 34–40.

[33] Sampson, A. Llvm for grad students, August 2015.

[34] Stelle, G., Olivier, S. L., Stark, D., Rodrigues, A. F., and Hemmert, K. S. Using a complementary emulation-simulation co-design approach to assess application readiness for processing-in-memory systems. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing* (Piscataway, NJ, USA, 2014), Co-HPC '14, IEEE Press, pp. 64–71.

[35] Steve Swanson. A Vision of Persistence, Jul 2018.

[36] Tramm, J. R., Gunow, G., He, T., Smith, K. S., Forget, B., and Siegel, A. R. A task-based parallelism and vectorized approach to 3d method of characteristics (moc) reactor simulation for high performance computing architectures. *Computer Physics Communications 202* (2016), 141 – 150.

[37] Tramm, J. R., Siegel, A. R., Islam, T., and Schulz, M. 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).

[38] Venkataraman, S., Tolia, N., Ranganathan, P., and Campbell, R. Consistent and durable data structures for non-volatile byte-addressable memory. pp. 61–75.

[39] Voskuilen, G., Frank, M., Hammond, S., and Rodrigues, A. Evaluating the opportunities for multi-level memory–an asc 2016 l2 milestone. *Sandia Report* (2016).