Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory

Panruo Wu
U. of California, Riverside
pwu011@ucr.edu

Dong Li
U. of California, Merced
dli35@ucmerced.edu

Zizhong Chen
U. of California, Riverside
chen@cs.ucr.edu

Jeffrey S. Vetter
Oak Ridge National Lab
vetter@computer.org

Sparsh Mittal
Oak Ridge National Lab
mittals@ornl.gov

ABSTRACT

The emergence of many non-volatile memory (NVM) techniques is poised to revolutionize main memory systems because of the relatively high capacity and low lifetime power consumption of NVM. However, to avoid the typical limitation of NVM as the main memory, NVM is usually combined with DRAM to form a hybrid NVM/DRAM system to gain the benefits of each. However, this integrated memory system raises a question on how to manage data placement and movement across NVM and DRAM, which is critical for maximizing the benefits of this integration. The existing solutions have several limitations, which obstruct adoption of these solutions in the high performance computing (HPC) domain. In particular, they cannot take advantage of application semantics, thus losing critical optimization opportunities and demanding extensive hardware extensions; they implement persistent semantics for resilience purpose while suffering large performance and energy overhead. In this paper, we re-examine the current hybrid memory designs from the HPC perspective, and aim to leverage the knowledge of numerical algorithms to direct data placement. With explicit algorithm management and limited hardware support, we optimize data movement between NVM and DRAM, improve data locality, and implement a relaxed memory persistency scheme in NVM. Our work demonstrates significant benefits of integrating algorithm knowledge into the hybrid memory design to achieve multi-dimensional optimization (performance, energy, and resilience) in HPC.

1. INTRODUCTION

Extreme scale HPC systems are characterized by power constraints, massive parallelism, and large working set size. As a result, the DRAM-based main memory has remained a crucial bottleneck because of its relatively low capacity and large power consumption. Non-volatile memories (NVM), such as resistive RAM (ReRAM), spin-transfer torque RAM (STT-RAM), and phase change memory (PCM), is expected to be able to revolutionize memory systems. They can have either better density (i.e., larger capacity per unit area) or lower stand-by power than DRAM, while being much faster than the traditional back-end storage. NVMs, however, have their own limitations: their write energy and access latency are typically larger than those of DRAM. Hence, NVM is usually combined with DRAM, forming a hybrid NVM/DRAM system to leverage the best characteristics of the two types of memories [15, 29, 32, 33, 34, 40, 37].

The hybrid main memory system, however, presents a challenge of effective data placement and movement across NVM and DRAM. Addressing this challenge is critical to optimize performance and energy efficiency of NVM. To exploit the persistence of NVM for resilience purpose, it is also desirable to manage data in NVM to support crash consistency, such that the application is resumable and re-computation is minimized after a system failure and reboot.

Most existing methods introduce hardware and software extensions to manage data placement in the hybrid main memory system. The hardware-based extensions treat DRAM as a transparent cache with customized caching policies to copy and migrate data [33, 32], or place DRAM and NVM side-by-side with the hardware-based monitoring mechanisms to capture memory access patterns and trigger data migration [40, 37, 34, 29, 15]. These hardware-based solutions, however, completely ignore application semantics. Due to the heuristic-based and reactive nature of their data movement algorithms, data migration in these works may trigger inefficient data movement with poor performance and low energy efficiency, and result in local optimum trap in adaptation.

On the other hand, the software-based extensions focus on implementing persistent semantics [30] for specific application domains, such as file systems [12], database [36], and key-value stores [11]. These designs, however, incur significant performance cost (120% increase in memory traffic and only 53% of the throughout of the pure DRAM-based system [41]) due to expensive logging [11, 36] or copy-on-write (COW) mechanisms [12, 35]. Although a recent work improves the performance [41], it cannot accommodate large-granularity persistent updates which are common in the HPC field. In general, most existing methods cannot work well for the HPC domain, because of the high-performance demands imposed in this domain.

In this paper, we re-examine the current hybrid memory designs from the HPC perspective, and aim to lever-

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

 ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907321

age the knowledge of numerical algorithms to direct data placement. Compared to other domains, the numerical algorithms in HPC are highly structured and formalizable. Hence, many research efforts have focused on using numerical algorithm knowledge to address the problems of fault tolerance [14, 18, 9], performance optimization [23, 38], and energy efficiency [20, 17]. In this paper, we extend the usage of numerical algorithm knowledge into a new territory: by introducing algorithm semantics into data management, we identify three opportunities to improve performance, energy, and resilience (PER) of the hybrid memory.

First, algorithm-directed data placement (ADDP) enables optimization for specific numerical operations, commonly found in computational kernels of HPC applications. These operations include matrix transpose, regular scatter/gather using an indirection vector, and strided memory accesses. The current common practice to implement these operations can cause expensive data copy or low cache utilization because of data over-fetching for sparse data items. ADDP avoids those problems by explicitly controlling how data are accessed and cached in hybrid memory systems. This data control happens as a *side effect* of data movement in the hybrid memory system which avoids the data copy overhead.

Second, ADDP results in more aggressive data management than the existing solutions. Guided by the algorithm structure and bounded algorithm performance, ADDP is able to prefetch data from NVM to DRAM. Coupled with a direct memory access (DMA) mechanism for bulk data copy, data prefetching can be overlapped with computation, minimizing the negative impact of data management on performance. Furthermore, ADDP can trigger data migration from DRAM to NVM proactively to save limited DRAM space and save DRAM static energy.

Third, ADDP enables a relaxed memory persistency scheme to implement system resilience on NVM. The strict persistency approach, implemented by recent works [26, 30], maintains the program order of every write request. This, however, is also one of the fundamental reasons for its poor performance. By leveraging the algorithm knowledge, we demonstrate that the data persistency on NVM is only needed to be guaranteed at certain algorithm phases with the quantifiable cost of recomputation at system crashes. Hence, the relaxed persistency approach brings performance benefits manifested by high instruction execution performance and memory level parallelism. Also, the recomputation cost of ADDP is much smaller than that of a common HPC resilience technique (i.e., the checkpoint/restart).

We demonstrate the effectiveness of ADDP using three representative computational algorithms (conjugate gradient, fast Fourier transform, and LU decomposition). In this paper, we use PCM as an example of NVM, although our methodology is applicable to other hybrid NVM/DRAM systems also. In summary, this work makes the following contributions.

- We provide an *algorithm-managed* hybrid NVM/DRAM system to optimize across multiple dimensions (performance, energy, and resilience). We show that algorithm features, common numerical operations, and algorithm structures can be leveraged to direct data placement without extensive hardware changes. This result is important for building next-generation extreme-scale HPC systems where one often needs to strike a balance between performance target and ownership cost.

- We propose a new relaxed memory persistence execution scheme on NVM. We reveal the correlation between data persistency and recomputation, and provide a new angle to examine the emerging memory persistency model [30] for NVM.

- We identify the necessary hardware support to implement ADDP. This hardware support includes a customized DMA mechanism for bulk data movement. In combination with the aggressive data management at the software level, the hardware support enables flexible algorithm control and productive programming experience.

- Our detailed evaluations show that ADDP provides higher performance (up to 49%) and energy efficiency of main memory (up to 25%), compared to a common data placement scheme without algorithm semantics. Further, write operations to PCM are significantly reduced (by up to 13x). Our approach incurs less than 2% performance overhead while achieving the same resilience as the checkpoint technique.

2. SYSTEM DESIGN

To enable algorithm-directed data placement, hardware and software must work in concert to support algorithm-specific optimizations. To avoid performance loss while improving energy efficiency, we must place write-intensive, frequently accessed data in the volatile memory (DRAM). Further, given the limited capacity of the volatile memory and large working set size of HPC applications, data in the volatile memory should be properly managed and timely migrated to the persistent memory (NVM) to save space and maximize the benefits of the volatile memory.

2.1 Hardware Design

The hardware design must accommodate three requirements to facilitate algorithm-directed data placement. *First*, the hardware design must support efficient, massive data movement. The massive data sets represented as data structures in a numerical algorithm often display uniform access pattern. For example, in the LU factorization algorithm, as the input matrix is iteratively decomposed, specific rows and columns of the input matrix can experience the same intensive update operations. These rows and columns can take 20% memory footprint of the algorithm. If they are identified as the candidates to migrate, massive data movement can occur. Similar examples can also be found in a large range of algorithm implementations based on well-structured stencils. *Second*, while the algorithm can direct critical data placement at the application level, we must relieve programmer from the burden of handling all data placement. Hardware is a solution to automatically direct data that cannot be guided by the algorithm. *Third*, to enable algorithm-directed data placement, hardware working with software must provide easy and direct access to the volatile memory and persistent memory. We organize DRAM and PCM to meet the second and third requirements, and introduce a DMA mechanism to meet the first requirement.

2.1.1 Main Memory Organization

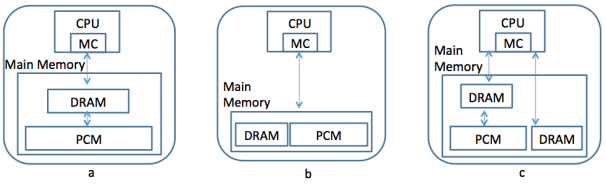


Figure 1: A logical view of main memory organizations

Figure 1.c logically shows the memory organization in our hybrid memory system. This is in contrast to the convention memory organizations shown in Figure 1.a [33, 32] and Figure 1.b [40, 37, 34, 29, 15]. In Figure 1.a, DRAM is organized similar to an inclusive hardware cache invisible to the OS. The limitation of this approach is that DRAM space does not add to the overall memory capacity. Also, the hardware-managed DRAM cache is generally applied to all workloads, even those with poor locality (e.g., the sparse matrix vector multiplication), potentially losing performance and energy efficiency. In Figure 1.b, DRAM and PCM are combined into a large flat memory. To optimize performance and energy efficiency, the data placement in DRAM and PCM is determined by hardware or OS. Hardware or OS continuously monitors temporal memory access patterns to direct data placement. Notice that the architecture shown in Figure 1.a loses application semantics, and does not meet the third requirement to facilitate algorithm-directed data placement. Also, the architecture shown in Figure 1.b imposes significant burden of data management at the software side, and does not meet the second requirement.

Figure 1.c logically displays our design. With this design, DRAM is divided into two parts separately managed by software and hardware. In particular, one part of DRAM is used as an exclusive software-managed memory (SMM), and shares the same physical address space as PCM (but with different addresses). With the support of system software (Section 2.2), the application can explicitly direct which data blocks should be copied from PCM to SMM where the PCM-unfriendly computation will occur, or which data blocks should be migrated from SMM to PCM when the computation finishes or the SMM runs out of space. The other part of DRAM is used as an inclusive hardware LRU cache for PCM, similar to Figure 1.a. This hardware-managed DRAM cache is inevitable, because some data cannot be controlled by the application or is difficult to control by the application (e.g., stack frames and a third party library). The placement of these data must rely on conventional hardware-based solutions without the need of application intervention.

To implement the hardware-managed DRAM cache in the above memory organization, we use a hardware design similar to [29, 33, 32]. In particular, the hardware-managed DRAM cache is organized similar to a traditional hardware cache. The hardware-managed DRAM cache is managed by an on-chip memory controller (MC) [29]. MC tracks whether data are located in the hardware-managed DRAM cache based on metadata (including tag, LRU, valid, and dirty bits). Each hardware-managed DRAM cache block has metadata. All metadata are stored in DRAM alongside data, but cached in MC to minimize metadata lookup latency [29]. Based on the retrieved metadata, a memory request is placed in either a DRAM scheduler or a PCM scheduler in MC. After data arrive at MC, if they should be cached into (or evicted from) the hardware-managed DRAM

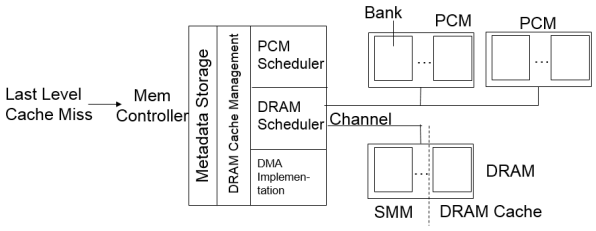


Figure 2: Our hybrid memory organization

cache, a migration request is inserted into the destination scheduler (either the DRAM scheduler or the PCM scheduler). The scheduler then writes data into the destination device.

Different from the existing hardware-based work, we treat SMM and PCM equally as the regular main memory. We use separate memory schedulers (one for PCM and the other for SMM and hardware-managed DRAM cache), because of the difference of memory timing constraints in DRAM and PCM. The hardware-managed DRAM cache and SMM are still composed of multiple banks organized as rows and columns of memory cells in the traditional manner, but they are separated at the granularity of memory bank. Figure 2 depicts important implementation details.

2.1.2 *Massive Data Copy and Migration*

To support algorithm-directed data placement and meet the first requirement, we introduce a DMA mechanism into MC to facilitate massive data copy and migration while minimizing data management overhead. We call data copy and migration collectively as *data management* in the later discussion.

DMA has been traditionally used to transfer data directly from the host memory to any input/output device without the host CPU intervention. We introduce DMA here to enable asynchronous data management between PCM and SMM. Asynchronous data management is a crucial performance optimization, especially when massive and frequent data management is desired in hybrid memory systems. In combination with algorithm direction at the user level, data management can overlap with computation, removing the data management overhead from the critical path.

An alternative for asynchronous data movement between memory devices without a dedicated DMA mechanism is to use helper threads. Although using helper threads for data movement is more flexible, we prefer the in-memory DMA for the following reasons. First, DMA frees CPU cores from moving the data by loading and storing, thereby increasing the system concurrency and possibly the performance. To use the DMA to move and transform data, CPU first initiates operations (in table 1) in the DMA and later checks the status of the operations. The DMA subsequently fulfills the data movements by scheduling memory access requests to MC. In contrast, the helper thread moves data by a series of load and store instructions executed by a CPU core. Those loads and stores go through the cache hierarchy, and the cache misses would result in memory access requests to MC. With bulk, non-sequential data movement, a high cache miss rate is expected. Whether DMA or a helper thread will perform better depends on the performance bottleneck of the application. For computation-intensive applications such as dense matrix operations and FFT where

the computations are likely to be the performance bottleneck, offloading data movements to DMA would minimize the negative performance impact of using the helper thread. For other less computation-intensive applications where the bottleneck would likely be memory latency or bandwidth, the available cores may not be fully utilized, and the performance between DMA and the helper thread may not be significantly different. Second, using the helper thread, there are some negative side effects of moving data by loading into the processor and then storing into the memory. Both loads and stores may pollute the caches, and the stores may incur unnecessary fetches of cache lines from the memory. Non-temporal store with write-combining and non-temporal load with load buffer mitigates the problem, but can only work effectively when reads and writes are sequential. For non-sequential memory accesses, the useless bytes in a cache line would have negative performance and energy impacts. Third, the DMA within MC can be specialized to schedule memory accesses better using dynamic request information. For example, a larger internal buffer can be employed to support data transformations in the memory. In [7], the memory copy by DMA is demonstrated to be 20% faster than the best copy algorithm using CPU cores. In summary, we use in-memory data movement based on DMA rather than the helper thread for performance and energy optimization.

To implement the DMA operations, a set of registers are introduced into MC, similar to existing designs [21, 7]. In particular, we add a set of DMA control registers to record memory address, byte count, operation type (e.g., scatter and gather), and operation specific parameters (e.g., data distribution for scatter and gather) for data management. We also introduce status registers to track data management status for the DMA controller. The DMA controller performs data management. After being configured with the control registers, the DMA controller fetches a block of data from the source device to a copy/migration buffer, and then inserts a copy request into the destination scheduler which writes data from the copy/migration buffer into the destination device. The above process is repeated until the entire transfer is completed. Afterwards, the scheduler notifies the DMA controller of the completion of data management. The DMA controller then updates the control registers to indicate the DMA completion.

Different from the existing designs [21, 7], all DMA related registers are mapped into memory address space to fully enable DMA programmability at the application level (with the assistance of OS for virtual memory management). In particular, before triggering data management with DMA, the application initializes the control registers and launches DMA operations. The data involved in the DMA transfer is not available to use until the DMA completion notification is provided by the status register and read by the application. This application-managed DMA mechanism is critical for algorithm-directed data placement. Different from prior hardware-based data management in conventional hybrid memory systems, the statuses of massive data management (e.g., start and completion) are exposed, opening new opportunities for performance optimization at the application level. These opportunities include prefetching (from PCM to SMM) and proactive eviction (from SMM to PCM) to overlap data movement and computation. To implement these two optimizations without DMA, one has to introduce thread-level parallelism (e.g., using a helper thread to manage data) and maintain thread synchronization.

Furthermore, motivated by vectored I/O DMA in networks, we introduce specific operations in DMA, including matrix transpose, strided memory accesses, and scatter/gather. These numerical operations are extremely common in HPC applications. For matrix transpose, the transpose happens as data are copied/migrated to the destination device, hence the transpose is treated as a *side effect* of data management, avoiding memory copy overhead in the traditional transpose implementation. For strided memory accesses, when sparse data items in PCM are copied from PCM to SMM, they can be collectively copied and packed in SMM. This improves on-chip cache utilization and improves performance. For scatter/gather operations, data copy from multiple memory areas can occur in a single DMA transaction, improving performance of data copy by leveraging memory level parallelism. These operations are initiated by specifying operation type in the DMA control registers.

Hardware Cost. The primary hardware cost includes the metadata storage for the implementation of the hardware-managed DRAM cache, and the DMA implementation (e.g., the DMA controller and registers). The metadata storage cost is the same as that in the existing hardware-based approach [29] (e.g., 8KB SRAM in MC), which is acceptable. For the DMA implementation, many previous works have proposed DMA with similar levels of complexity and sophistication [22, 3, 21, 7] with manageable area size. Our hardware cost is comparable to them.

2.2 Software Design

At the system software level, the DRAM cache is invisible and does not need any software support. PCM is treated as the main memory and managed by the traditional virtual memory management (VMM) in OS. SMM shares the same physical memory space with PCM, and is managed by VMM without paging. Paging is not supported by SMM, because the capacity-limited SMM tends to hold performance critical data and paging could significantly impact performance. From the programmer's view, SMM is just a pre-allocated memory space. The virtual address of the SMM space is mapped to the physical address of SMM by OS. In common programming practices, a global pointer can be employed to track the boundary of free SMM memory.

We also introduce a set of nonblocking memory operation APIs to enable algorithm direction and leverage DMA. These APIs are shown in Table 1. Here, *dma_memcpy* launches regular memory copy operations with DMA; *dma_stride*, *dma_gather*, *dma_scatter*, *dma_transpose*, and *dma_lacpy* perform optimized DMA operations as discussed in Section 2.1. Each of these DMA operations returns an ID. The request ID is used later to query the status of DMA or wait for its completion with *dma_wait* or *dma_test*. In addition, these DMA operations allow the programmer to customize data type with *dma_datatype*, especially for non-contiguous data in memory. The DMA primitives can be implemented in an operating system as system calls to read/write DMA registers. In general, we introduce limited extensions to the system software while introducing rich DMA semantics to enable algorithm directed data placement.

3. ALGORITHM-DIRECTED DATA PLACEMENT

Table 1: APIs for managing and operating on SMM

API	Description
int **dma_memcpy**(void **src*, void **dest*, int *count*, DMA_Datatype *datatype*)	dest[i]=src[i] (i=0:count-1)
int **dma_stride**(void **src*, void **dest*, int *count*, int *stride*, DMA_Datatype *datatype*)	dest[i]=src[stride*i] (i=0:count-1)
int **dma_gather**(void **src*, void **dest*, int *count*, int **idx*, DMA_Datatype *datatype*	dest[i]=src[idx[i]] (i=0:count-1)
int **dma_scatter**(void **src*, void **dest*, int *count*, int **idx*, DMA_Datatype *datatype*)	dest[idx[i]]=src[i] (i=0:count-1)
int **dma_transpose**(void **src*, void **dest*, int *dim1*, int *dim2*, int *m*, int *n*, DMA_Datatype *datatype*)	dest[j+dim1*i]=src[i+dim2*j] (i=0:m-1,j=0:n-1)
int **dma_lacpy**(void **src*, void **dest*, int *dim1*, int *dim2*, int *m*, int *n*, DMA_Datatype *datatype*)	dest[i+dim1*j]=src[i+dim2*j] (i=0:m-1,j=0: n-1)
int **dma_wait**(int *request*, DMA_Status **status*)	Wait for a DMA request to complete
int **dma_test**(int *request*, DMA_Status **status*)	Test if a DMA request is completed

In this section, we discuss details of the algorithms. While the algorithm details can vary from one algorithm to another, there are several common approaches for using algorithm knowledge to direct data placement. In this section, we first provide a general description of these approaches, and then show three case studies to explain how these approaches can be utilized in specific algorithms.

3.1 General Description

To improve performance and energy efficiency of the hybrid memory system, we rely on effective use of three techniques: (1) strategically placing data structures in SMM and PCM to avoid unnecessary thrashing in hardware-managed DRAM cache and expensive write in PCM; (2) using the asynchronous data movement primitives between SMM and PCM to overlap data movement and computation; (3) using hardware supported special data movement primitives (transpose, gather and scatter, etc) to efficiently move and transform data.

To implement (1), we leverage algorithm knowledge to determine the dynamic behavior of program based on the analysis of algorithm complexity and data criticality. Using algorithm knowledge, we always place performance- and algorithm-critical data into SMM. This is different from the existing approaches that direct data placement largely based on temporal access patterns. The direction of data placement based on the temporal access pattern can result in hardware-managed DRAM cache thrashing when the data access pattern changes across the execution of algorithm. Although the compiler-based data flow analysis and liveness analysis can also provide hints to implement (1), the complexity of inter-procedure analysis and alias analysis often underestimate data usage and skew the decision. As we show later, the effectiveness of (1) is especially pronounced in the first case study (i.e., the conjugate gradient algorithm).

To implement (2), we leverage algorithm knowledge to determine the best point to trigger proactive data movement. Using algorithm knowledge, we maximize the overlap between computation and data movement without impacting execution correctness. The effectiveness of (2) is especially pronounced in the second case study (i.e., the Fast Fourier Transform). To implement (2), we can also use general compiler and runtime techniques to direct data placement. For example, using a task-based programming model, we encapsulate computation and data movement into different tasks, and rely on runtime to determine the task scheduling and rely on task-level parallelism to improve performance. However, this method has two limitations. First, to ensure execution correctness, the task scheduling largely depends on data dependency analysis. However, a coarse-grained data dependency analysis (e.g., at the level of data array) can serialize task execution and reduce concurrency, while a fine-grained data dependency analysis can bring large runtime overhead to track data dependency. Second, an algorithm implementation based on a task-based programming model sometimes requires restructuring of applications to address synchronization and consistency issues. By comparison, working at the algorithm level of abstraction provides succinct knowledge of data dependency at fine granularity. Further, it does not incur runtime overhead, and the cost of algorithm optimization can be easily amortized over frequent use of algorithm in large-scale applications. Note that the feasibility of the algorithm-level optimization is already demonstrated by well-known algorithm-level work (e.g, ScaLAPack [2] and PLASMA [1]).

To implement (3), we locate appropriate data movements in an algorithm, and then replace them with special hardware supported data movement primitives. While the compiler, with the assistance of user-annotation, may also implement (3), the identification of numerical operations and their optimization on a hybrid memory system still requires sufficient algorithm knowledge.

Implementation of persistency semantics for NVM implies that we exploit the non-volatility property of NVM to resume computation after system crashes. The key challenge in doing so is that after failure, the data state in PCM must be guaranteed to be *consistent*. Consistency means that the data in PCM represents a valid state in the fault free program execution. The data in PCM do not always constitute a consistent state due to the out-of-order processor and memory system and data buffering in volatile caches. Upon failure, the data in PCM could be in an invalid state. Implementing the persistency semantics for the HPC domain must meet two requirements: minimizing recomputation after application restarts and minimizing runtime overhead during fault free execution.

To meet the above requirements, we introduce a relaxed persistency scheme. We maintain the data consistency only at algorithm well-defined points. Hence, the persistency is not guaranteed between those points, forming the relaxed persistency [30]. In particular, as many numerical algorithms follow an iterative structure, we choose the end of each iteration to maintain persistency. This method avoids extensive logging or COW in the existing approaches [11, 36, 12, 35], and greatly reduces the runtime overhead. Furthermore, recomputation is bounded by only one iteration of the algorithm, much smaller than that of existing checkpoint/restart techniques that have recomputation at the granularity of a whole algorithm or a complete application phase with multiple algorithms.

Maintaining persistency at the end of each iteration means we ensure that the critical algorithm data is consistent in PCM at the end of each iteration. Furthermore, there should be one resumable state available at any time (not just at the end of each iteration), in case of application failures. To implement the above goal, we can simply employ a local checkpoint to create a persistent data copy in PCM. However, this will create significant performance overhead due to frequent checkpoint in the critical path of computation.

Different from checkpointing techniques, we leverage algorithm knowledge to implement data persistency as the algorithm updates the data. Using this method, we achieve the same resilience as provided by checkpoint, albeit with much smaller overhead.

To further explain our methodology, we categorize common numerical methods into three classes based on their iterative structure.

1. In-place streaming: data are processed in a streaming manner and transformed in-place and there is no dependency between iterations. Examples include 2D/3D FFT, and some structured grids.

2. Iterative without history: one iteration only depends on its last iteration, and the output of the iteration is overwritten by its next iteration (i.e., history is not preserved). This class includes most iterative solvers (CG, GMRES, MultiGrid) and time-step solvers.

3. Iterative with history: one iteration only depends on its last iteration, but different from the iterative without history, a part of the output of the iteration must be preserved across iterations. Many dense linear algebra methods fall into this class, such as right-looking LU decomposition.

For the first class of algorithms, since there is no inter-iteration dependency, as long as data is committed into PCM to ensure persistence at the end of iteration, the application is resumable from the data in the last iteration.

For the second class of algorithms, at each iteration we employ two versions of the data for read-write data structures[1]: one version is read-only and represents the most recent resumable state from the last iteration; the other version (also named as the *working version*) is read and written, and used for the computation of the current iteration. Once the current iteration is done, the working version is committed into PCM to ensure persistency, and then becomes the read-only version for the next iteration, while the other version becomes the working version. For this algorithm class, since history is not preserved across iterations, we can switch the two versions across iterations. This method is fundamentally different from checkpointing, since checkpointed data are never involved in computation and the application has to suffer from data copy overhead. In contrast, our method integrates the maintenance of data persistency into computation, and completely removes explicit data copy. We name our method, the *twin data* technique in the rest of the paper.

For the third class of algorithms, a similar two version scheme can be devised, but we must avoid overwriting the history when switching the two versions in PCM. We use algorithm knowledge to ensure that the essential data in the current working version has committed into PCM without overwriting the history data.

To implement the above relaxed persistency scheme, we must commit appropriate data to PCM. The data commit operation includes writing back dirty, on-chip cached data to PCM and writing dirty SMM data into PCM. This creates runtime overhead, but the overhead is bounded and limited, given the small cache size and SMM size. Furthermore, as we switch the two versions, we pollute the on-chip caches, which creates further overhead. However, comparing with intensive computation within each iteration, the overhead at the end of each iteration can be easily amortized. We quantify the overhead in Section 5.

[1]Read-only data always has persistency in NVM, and write-only data can leverage the method for the first algorithm class to maintain persistency.

1: **ConjGrad**
2: Input: The sparse matrix A, right hand side b, and initial guess x
3: Output: The solution to $Ax = b$
4: Initialization: $q \leftarrow 0, z \leftarrow 0, r \leftarrow b - Ax, p \leftarrow r, \rho \leftarrow r.r$, where x is some initial guess
5: Data placement: p, q, r, z are placed in SMM.
6: **for** i=0,1,... **do**
7: (commit $q, r, \alpha, \rho_0, \rho, p, z$)
8: (q') $q \leftarrow Ap$
9: (α')$\alpha \leftarrow \rho/(p.q)$
10: (ρ_0')$\rho_0 \leftarrow \rho$
11: (z')$z \leftarrow z + \alpha p$
12: (r')$r \leftarrow r - \alpha q$
13: (ρ')$\rho \leftarrow r.r$
14: (p')$p \leftarrow r + (\rho/\rho_0)p$
15: Check convergence: $r = A.z$?
16: (switch $q, r, \alpha, \rho_0, \rho, p, z$)
17: **end for**

Figure 3: Conjugate gradient algorithm. Capital letters such as A represent matrices; lowercase letters such as x, y, z represents vectors; Greek letters α, ρ represent scalar numbers. The purple operations inside parentheses are what happen in a version switching iteration. Blue text is the data placement optimization for performance and energy.

Discussion: The algorithm knowledge can estimate dynamic behaviors, identify data criticality, ensure execution correctness, and model performance. Based on the algorithm knowledge, we can perform optimizations that cannot be achieved by compiler and runtime. This has been demonstrated by the success of several highly optimized linear algebra packages [1, 2]. In fact, we expect compiler and runtime can help even further in identifying opportunities (e.g., read/write patterns and data dependency across iterations) to efficiently apply algorithm knowledge.

3.2 Case studies

In this section, we present cases studies on three representative numerical applications: conjugate gradient, fast Fourier transform, and LU decomposition for a matrix.

3.2.1 *Conjugate Gradient*

The conjugate gradient (CG) is one of the most commonly used iterative methods to solve the sparse linear system $Ax = b$ when the coefficient matrix A is symmetric positive definite. Figure 3 lists the algorithm pseudocode.

CG involves two sparse matrix vector multiplication (SpMV, lines 8 and 15), three vector updates (lines 11,12, and 14) and two vector inner products (lines 9 and 13) in every iteration of the method. In general, the algorithm computes successive approximation to the solution, computes residuals corresponding to the approximate solutions, and determines search directions to update both the approximate solutions and the residuals. Vector updates and vector inner products are lightweight, cache friendly and fast. The dominant kernel of CG is SpMV, which is also the kernel of many other iterative Krylov solvers. SpMV usually involves indirect indexing of an array. The access pattern of indirect indexing of an array exhibits very limited spatial and temporal locality.

To optimize performance and energy consumption, we note that in every iteration, the entire matrix A, which consumes the major memory footprint, can be read into the hardware-managed DRAM cache. As a result, the performance-critical vectors p, q, r, z will be evicted out of hardware-managed DRAM cache and written back to PCM. This unexpected data eviction can be triggered by the prior hardware-based data placement solutions based on temporal locality analysis, which causes performance loss. To prevent this undesirable effect, we pin the frequently updated vectors into SMM so that they will never be evicted to PCM and benefit from the fast read/write speed of DRAM.

To implement the relaxed persistency without checkpointing, we examine the read-write data structures within each iteration (i.e., the vectors q, p, r, z), and implement the twin data technique. As shown in Figure 3, in the original algorithm these vectors are updated at each iteration, and they are only dependent on the output from the last iteration. Hence CG is the second class of algorithm discussed in Section 3.1.

With the twin data technique, we create two sets of the vectors, and alternate between them when updating them within the iteration. In particular, in each iteration we read from one set of the vectors and write the other set; in the next iteration we read from the other set and write the first set. By doing so, at any given time, there is always a set of consistent vectors available, i.e., the read-only set of the vectors. Using these consistent vectors and the read-only data structure (i.e., the sparse matrix A), the CG implementation is always resumable with the recomputation less than one iteration.

As a further optimization, we could switch the two versions every multiple iterations instead of every iteration to reduce runtime overhead. Also, using algorithm knowledge we notice that the vectors q, r can be re-generated from p, z. Hence we can exclude q, r from the implementation of the two versions to further reduce runtime overhead. However, the above optimization will result in larger recomputation after application failures.

3.2.2 Fast Fourier Transform

Fast Fourier Transform (FFT) is one of the most popular and widely used spectral methods [5]. It is used to compute the Discrete Fourier Transform (DFT) and its inverse. Given an input data grid $X(n1, n2, n3)$, a typical 3D FFT performs transformation and outputs the transformed X. In general, 3D FFT performs 1D FFT transformation on each of the three dimensions of the 3D data grid and 2D transpose. 2D transpose aims to transpose non-consecutive dimensions into consecutive dimensions to improve data locality. The pseudocode of 3D FFT is shown in Figure 4.

To improve performance and energy efficiency of 3D FFT, we optimize it using aggressive data management and the transpose DMA. In particular, the basic operations for 3D FFT are 2D transpose and 1D FFT, each of which has intensive write operations on the input data grid. Hence, the data grid must be copied into SMM for performance and energy efficiency reasons. When copying data from PCM to SMM, we leverage transpose DMA to implement data transpose as a side effect of data copy.

Furthermore, we overlap data copy (i.e., transpose) with computation (1D FFT). Specifically, there are three loops in 3D FFT shown in Figure 4. According to the algorithm

```
1: FFT3D
2: Input: a 3-D array X(n1, n2, n3), column major
3: Output: 3D DFT'ed X(n1, n2, n3)
4: FFT along the x-axis:
5: for k = 1, ..., n3 do
6:    FFT along 1st dimension on plane X(:, :, k)
7: end for
8: FFT along the y-axis:
9: for k = 1, ..., n3 do
10:   Transpose the plane X(:, :, k) into P
11:   FFT along 1st dimension on plane P
12:   Transpose the plane P back to X(:, :, k)
13: end for
14: FFT along the z-axis:
15: for j = 1, ..., n2 do
16:   Transpose the plane X(:, j, :) into plane P
17:   FFT along 1st dimension on plane P
18:   Transpose plane P back to X(:, j, :)
19: end for
```

Figure 4: 3D Fast Fourier Transform

knowledge, within the second loop, the 1D FFT can start to work on the transposed data, even before the first 2D transpose is completed; the second 2D transpose can start to transpose the data processed by 1D FFT, even before the 1D FFT is completed. Hence, we can implement a pipeline of the two transposes and 1D FFT by partitioning SMM into three parts. Each part is in charge of either transpose or 1D FFT operation in a round-robin manner. Figure 5 generally depicts this execution paradigm for the second and third loops.

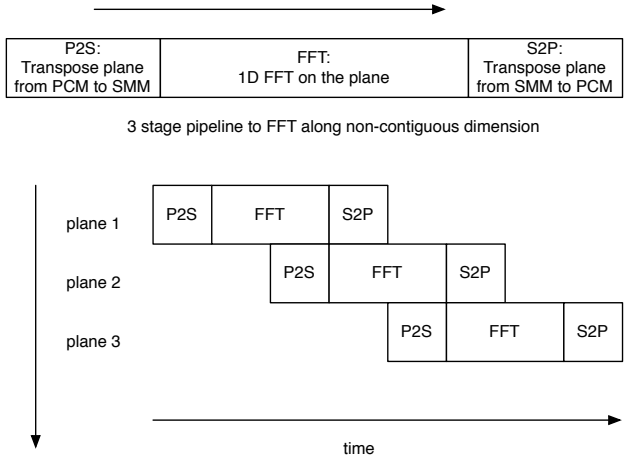
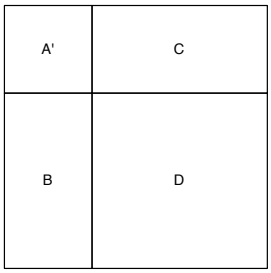


Figure 5: Optimizing 3D FFT with transpose DMA and aggressive data management

To implement the relaxed persistency without checkpointing, we examine the read-write data structure, the data grid X. Since the data planes in X are processed one by one, FFT falls into the first algorithm class discussed in Section 3.1. At the end of each iteration in the three loops, we maintain persistency of X as a data plane is moved or transposed (using transpose DMA) from SMM to PCM. We also use a single variable to bookkeep the data plane position within X. Upon failure and reboot, the computation will be resumed based on the bookkeeping variable and persistent data planes. Note that implementing the above relaxed persistency does not increase data movement and computation, and does not alter the overlap between computation and data movement, therefore the run time overhead is min-



1: **while** A is not empty **do**
2: Factorize the left NB columns A, B and update the top NB rows (C).
3: Use updated panels from the previous step to update the trailing (right-bottom) matrix $D \leftarrow D - BC$.
4: Move to work on the trailing matrix: $A \leftarrow D$
5: **end while**

Figure 6: Blocked right-looking LU factorization without partial pivoting

imized.

3.2.3 LU Decomposition

The LU decomposition is the standard algorithm to solve a general dense linear system. LU decomposition factors a general matrix A into the product of a lower triangular matrix (L) and upper triangular matrix (U) such that $A = LU$. To improve numerical stability, some LU algorithms employ partial pivoting. In the discussion below we do not consider partial pivoting because of page limitation, but our method is applicable to partial pivoting. Figure 6 generally describes the popular blocked right-looking LU algorithm.

For the right-looking LU algorithm, we do not have opportunities to improve performance and energy efficiency, however we can employ the twin data technique to achieve resilience improvement. Figure 7 depicts the memory snapshots when applying the twin data technique. Each memory snapshot corresponds to one step (i.e., one iteration) of LU. For the LU algorithm, the matrix A is a read-write data structure, and falls into the third algorithm class discussed in Section 3.1. At the beginning, we create a copy of the matrix A (see the memory snapshot 0), creating two instances of A (the new copy and the original one). Then the panel update is performed in one instance of the matrix A (the left one), and the updated area is highlighted in the figure (see the version 1 in the memory snapshot 1). If a failure happens during the transition from the memory snapshot 0 to 1, we can safely rollback to the version 0 (see Figure 7) from one instance of the matrix A (the right one particularly). After the step 1 is finished, the left matrix is committed to PCM to maintain persistency, and the right matrix becomes the working version. Afterwards, in the step 2 we read the panel from the left matrix but write to the right matrix shown in the snapshot 2. The algorithm continues the above process by alternating matrix update between the left and right instances.

With the above implementation, at any moment there is a resumable state existing in one of the two matrix instances. The recomputation after the system crashes is limited to one iteration. Note that we achieve the above resilience without intensive copying operations and with minimal changes to the original algorithm. In addition, at the end of LU, the

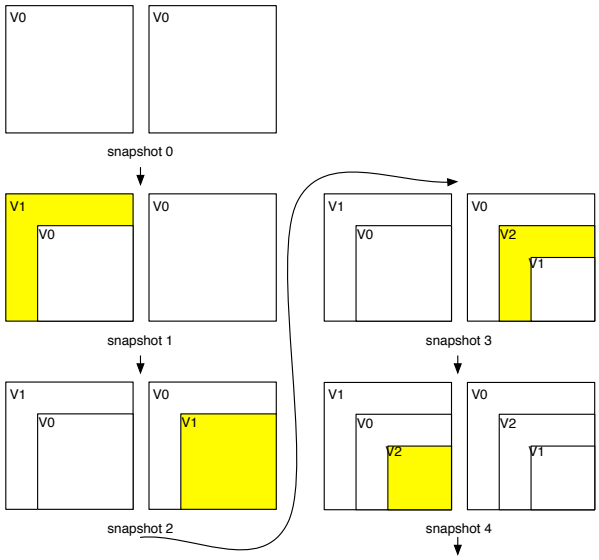


Figure 7: The implementation of the twin data technique for LU factorization. The updated areas are highlighted with yellow color.

final result is scattered between the two instances of the matrix. We must merge the two instances, but the merging operation is simple and the overhead is marginal comparing to the whole LU factorization.

4. EXPERIMENTAL METHODOLOGY

Our experiments are based on McSim [4], a PIN [27] based multi- and many-core cycle accurate simulation infrastructure. McSim provides event-driven timing simulation and models cores, caches, directories, on-chip networks and memory channels. We enhance the main memory model in McSim to support the hybrid DRAM/PCM simulation. The implementation of the new memory model allows configurable PCM/DRAM ratio and configurable memory organization. The new memory model also accounts for the microarchitecture of the DRAM and PCM devices. We further extend the implementation of the memory controller in McSim to implement DMA functionality.

Table 2 lists the detailed parameters and architecture configurations for the processor and memory system in our simulation. DRAM timing parameters are based on the Micron specification [28]; PCM timing parameters are based on [31, 10]. We calculate DRAM and NVM energy consumption based on the number of memory accesses broken down into row buffer hits, misses and the memory energy parameters listed in 2. This method is also utilized in [24, 41]. The memory energy parameters are listed in Table 2, and these parameters are based on [25, 8]. We collect performance and energy consumption results of the running phase of the numerical kernels, skipping the initialization phase.

To compare with the existing hardware-based solutions for data placement, we implement a hardware-managed DRAM cache in [33] as our baseline machine. This hardware-managed DRAM cache mechanism is one of the most common hardware-based data placement solutions. To distinguish this hardware-managed DRAM cache mechanism with the DRAM cache employed in our system, we name the DRAM cache mechanism in [33] as the *pure DRAM cache*. Note that while

Table 2: Simulation parameters

Processor	8-way super scalar, ROB size 64, instruction queue size 128
Memory organization	2 memory channels, 1 (DRAM) or 4 (PCM) ranks per channel, 8 banks per rank PCM 1GB, DRAM 32MB, SMM:DRAM cache=1:1 by default,
L1 Cache	split I/D caches, each 16KB, 4 ways, 64B block, private cache
L2 Cache	a unified 4MB cache, 16 ways, 64B block, shared cache
Memory Controller	64-entry Transaction Queue, 16-entry Command Queue, FR-FCFS, closed-page
Timing (cycles) [31, 10]	DRAM - tRCD: 14, tRAS: 34, tRP: 14, tRR: 1, tCL: 14, tBL: 4 tRRDact: 5 PCM - tRCD: 37, tRAS: 50, tRP: 14, tRR: 1, tCL: 10, tBL: 14, tRRDact: 3
Energy (pJ/bit) [40]	DRAM - Array read: 1.17, Array write: 0.39, row buffer read: 0.93, row buffer write: 1.02, background power: 0.08 PCM - Array read: 2.47, Array write: 16.82, row buffer read: 0.93, row buffer write: 1.02, background power: 0.08

the DRAM cache studied in [33] is abstract and geared towards exploiting the high density of PCM to reduce page faults, we are more interested in evaluating the hardware-managed DRAM cache idea in an HPC environment where the page fault is not a significant issue but the concurrency and latency of the hardware-managed DRAM cache are critical. Thus, we employ a highly optimized hardware-managed DRAM cache as our baseline. This DRAM cache has high hit concurrency (up to 64 concurrent requests), high associativity (32 way) without extended latency for tag matching, high miss concurrency (up to 64 concurrent requests). The implementation of such cache could be expensive but achieves excellent performance for HPC applications which are sensitive to the concurrency and latency of the cache. Hence, our hardware-managed DRAM cache is a high bar for evaluation, representing the best we can obtain from a transparent cache.

5. EVALUATION

We use CG (class B, CG.B) and FT (class A, FT.A) from NAS Parallel Benchmark (NPB3.3) suite, and LU from LAPACK [2] (the DGETRF input with a square matrix of size 1000) as algorithm implementation. Table 3 shows the memory system configurations for evaluation. Besides the baseline cache and ADDP, we also evaluate pure DRAM and pure PCM to reveal the implication of our designs and algorithm characteristics. All results are normalized to those of the baseline hardware-managed DRAM cache.

Table 3: Configurations of the memory system

Name	hardware-managed DRAM cache	DRAM SMM	Main memory	DMA
Baseline	Yes	No	PCM	No
ADDP	Yes	Yes	PCM	Yes
Pure DRAM	No	No	DRAM	No
Pure PCM	No	No	PCM	No
ADDP DRAM	Yes	Yes	DRAM	Yes

Performance: Figure 8 shows the execution time of CG.B and FT.A. The figure does not show the results for LU, because its implementation is highly optimized and bounded by computation, and can achieve 90% of the peak performance. The main memory system plays little role in determining the performance and hence, we do not optimize it. However, we will discuss the ADDP runtime overhead of LU in Figure 11.

Figure 8 shows that for CG our optimization scheme (i.e., placing the frequently updated vectors in DRAM SMM) reduces run time by 9% compared against the baseline. The performance improvement comes from eliminating the hardware-managed DRAM cache thrashing problem analyzed in Section 3.2.1. . The performance of ADDP is very close to that of the pure DRAM system, and is much better than that of the pure PCM system. This demonstrates the effectiveness

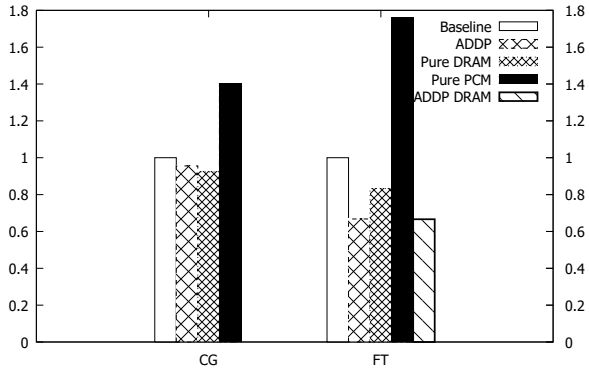


Figure 8: Execution time

of ADDP for performance optimization.

For FT, we notice 49% performance improvement with ADDP over the baseline pure hardware-managed DRAM cache system. The significant performance improvement comes from successfully exploiting parallelism between processor and memory system with the DMA engine. ADDP in this case performs even better than a pure DRAM main memory system without DMA. To investigate the reason, we compare the performance of ADDP in the hybrid memory with that of ADDP in the pure DRAM system (labeled as *ADDP DRAM* in Figure 8 and Table 3). We found that the performance of these two approaches is very close. This result suggests that ADDP effectively removes the high latency of PCM access from the critical path by overlapping computation and data movement. Also, given the better performance of ADDP DRAM over the pure hardware-managed DRAM cache, we infer that there is significant parallelism untapped by the pure DRAM memory system. Hence, using DMA, we open new opportunities to improve performance.

To further understand the reasons why ADDP performs better than the baseline, we profile the data migration volume and hardware-managed DRAM cache miss rate. For ADDP, the hardware-managed DRAM cache miss refers to the cache miss happening in the inclusive hardware DRAM cache. From the figure 9, we notice that for CG the data migration volume reduces by 90%, which explains the performance benefit of CG. However, we do not see a significant difference between the baseline and ADDP for the cache miss rate. This is because the memory references to the read-only, input matrix A of CG (see the pseudocode in Figure 3) with a large memory footprint account for the major cache misses, while ADDP for CG does not optimize the data placement of A. For FT, we notice the significant reduction in both data migration and cache misses, which explains the big perfor-

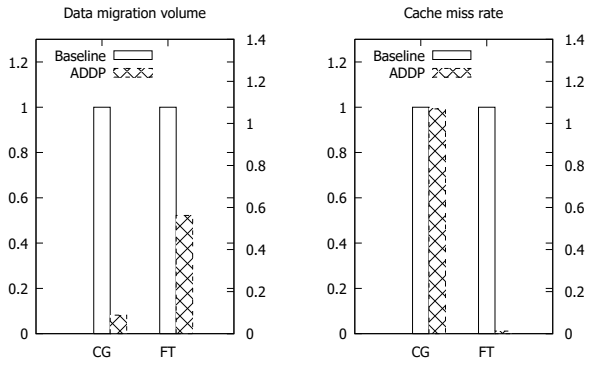


Figure 9: Data migration from DRAM to PCM and DRAM cache miss rate

mance benefits of ADDP. The reduction in data migration and cache misses comes from our pipelined data management and optimized data transpose operations that avoid unnecessary data movement.

For both CG and FT, the performance difference between ADDP and pure DRAM is less than 12%. This performance gap is smaller than that achieved by the existing work [11, 36], demonstrating the feasibility of using ADDP for HPC.

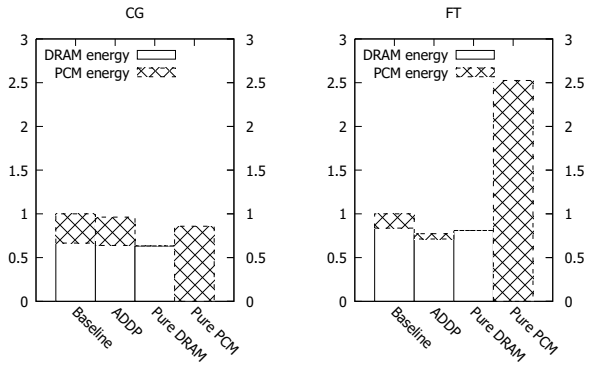


Figure 10: Dynamic energy of memory system

Energy: Figure 10 shows energy consumption of the main memory including both DRAM and PCM. For CG, we notice that in terms of energy consumption, the pure PCM is close to ADDP and the baseline cache, and is only second to the pure DRAM. In combination with the performance results, we conclude that for CG, we improve performance at the cost of slightly increased energy consumption on a hybrid memory system. However, comparing with the baseline, ADDP still successfully reduces energy consumption by 9.2%.

For FT, ADDP consumes the least energy, comparing with other three cases using PCM (25% energy saving comparing with the baseline case). With ADDP, we also have a high ratio of DRAM energy to PCM energy. This result is an indication of good caching effects—most memory accesses are served by DRAM. In contrast to CG, the pure PCM memory in FT consumes much more energy than the other cases, because FT is a write intensive application and writing PCM consumes much more energy than reading. In this case, a hybrid DRAM/PCM system is beneficial for energy saving.

Resilience: To quantify the overhead of the relaxed persistency scheme, we compare the performance with and without the twin data technique. Figure 11 shows the results. As discussed in Section 3.1, the twin data technique can incur overhead when switching the two versions. In particular, the program must wait for the data in various volatile buffers to commit and become persistent in PCM; switching versions may also change the caching behavior because of the temporary expansion of working set. Note that employing the two versions does not change the locality property of the algorithm.

Figure 11 shows that the version switching overhead for the three algorithms is negligible (~1%), even if we switch the two versions at every iteration. This low overhead is due to the limited cache size and infrequent version switching. In the following analysis we assume there is negligible overhead in version switching, with up to one iteration re-computation when recovering from failures.

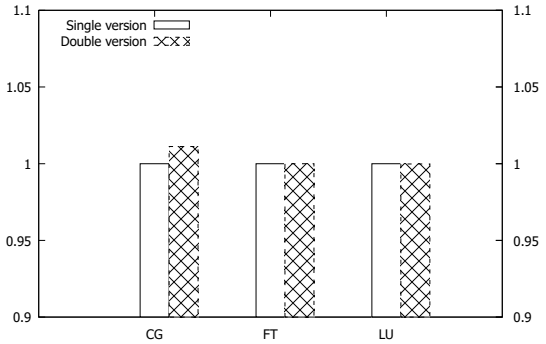


Figure 11: Version switching overhead of CG.B and FT.A based on ADDP

To further evaluate the effectiveness of our approach, we compare the twin data technique with a checkpointing technique. Given the lack of simulation capabilities for HPC checkpointing, we build an analytic model to make the comparison. The state-of-art checkpointing technique [16] employs PCM to implement a two level scheme. The first level is a global synchronous checkpoint that saves a consistent global state into stable global storage such as disks or neighboring PCM. The second level is a local synchronous checkpoint that saves the local state into local PCM. The paper leverages 3D stacking between DRAM and PCM to accelerate the local checkpointing while avoiding expensive global checkpoint with overhead around 6% [16].

The effective use of the twin data technique can serve the *functionality* of the local checkpointing, while avoiding the checkpointing overhead or the requirement of 3D stacking technology. Hence, we expect to see a dramatic reduction in checkpointing overhead.

To make the comparison relevant to HPC, we use the optimal checkpoint interval in [13]. The expected execution time of an application with checkpoint/restart can be decomposed into the following components:

$$\begin{aligned} T_{\mathrm{C/R}}(\tau) &= \text{solve time} + \text{dump time} + \text{rework time} + \text{recovery time} \\ &= \frac{M}{p} e^{pR/M} (e^{(\tau+\sigma)p/M} - 1) \frac{T_s}{\tau} \end{aligned} \tag{1}$$

where the solve time (T_s) is the original execution time of the computation, the dump time is the time to perform pe-

riodical dumping the state of the program to stable storage, the rework time is the work lost since the latest checkpoint, equivalent to the time elapsed when failure happens since last checkpoint, and the recovery time is the time required to be able to recompute from the latest checkpoint including reading the stored checkpoint data back, rebooting, re-initialization, etc. In the quantitative representation, M is the mean time to interrupt (MTTI) of a system component, p is the number of components, τ is the optimal checkpoint interval, σ is the time to do one dumping, and R is the time for one restart. The optimal checkpoint interval τ that minimizes $T_{\mathrm{C/R}}$ can be found in [13].

With the twin data technique, since there is no checkpointing overhead, the execution time will consist of only the original solve time (T_s) and the recovery time (R). Using the same fault probability model as the checkpointing scheme, we calculate the expected execution time with the twin data technique as

$$\begin{aligned} T_{\mathrm{twin}} &= \text{solve time} + \text{recovery time} \\ &= T_s e^{pR/M} \end{aligned} \tag{2}$$

Based on the above modeling, we compare the expected execution time of the state of the art checkpointing mechanism and the twin data technique. The ratio of the expected execution time is as follows, assuming the recovery times are the same for both techniques.

$$\frac{T_{\mathrm{C/R}}}{T_{\mathrm{twin}}} = \frac{M}{\tau p}(e^{(\tau+\sigma)p/M} - 1) \tag{3}$$

Note that the ratio is always larger than 1, as $e^{(\tau+\sigma)p/M} - 1 \geq \frac{(\tau+\sigma)p}{M}$. As the scale ($p$) continues increasing exponentially, the ratio will grow very fast, even if we assume the optimal checkpoint interval τ for C/R. The above analysis demonstrates the performance benefits of introducing algorithm knowledge into the resilience design.

The above analysis assumes that the recomputation time is the same for ADDP and checkpointing. This is very conservative, because the recomputation with ADDP is bounded by one iteration, much smaller than that of checkpointing.

We further quantify and compare the performance of ADDP and the checkpointing mechanism with an example. We assume that a single dump (including the necessary coordination) is 0.5 minute, the MTTI of a single component (processor or socket) is 10 years, and the recovery time (including rebooting, re-initialization) is 1 minute. We use the model (1) to calculate the normalized execution time of C/R using the optimal checkpoint interval from [13], and model (2) to calculate the normalized execution time with the twin data technique. Figure 12 shows the performance. The figure reveals the lower execution time with the twin data technique and the rapidly increasing performance gap between the twin data and the checkpointing mechanism as the system scale becomes larger.

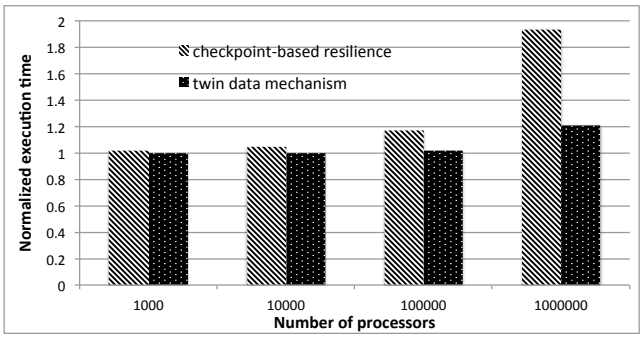


Figure 12: Overheads comparison: twin data vs. checkpoint

6. RELATED WORK

Some studies have considered hardware-based data placement for the hybrid memory system. Ramos et al. [34] rely on MC to monitor write intensity and popularity of memory pages, which is used to migrate pages between DRAM and PCM. Bivens et al. [6] and Qureshi et al. [33, 32] use DRAM as a set-associative cache that is logically placed between processor and PCM. Yoon et al. [40] place data based on row buffer locality in memory devices. Wang et al. [37] rely on static analysis and MC runtime monitoring to determine data placement on GPU. A key limitation of these approaches is that they rely heavily on hardware-based monitoring mechanisms and caching policies to direct data placement without awareness of application semantics. Hence, they can result in inefficient data copy/migration with poor performance and low energy efficiency, because the data management algorithms are generally based on memory access patterns monitored within a user-defined time period. Depending on the duration of the time period and other heuristic parameters to trigger data movement, these algorithms may not work well for a range of workloads and need to be disabled to avoid performance loss. By contrast, our technique takes a holistic view of algorithm structure and application data-access pattern and hence, it avoids the above problems in the traditional solutions. A few other works use software-based extensions to implement persistency semantics [12, 36, 11, 35]. However, these techniques cannot work well for HPC domain, due to their large overhead and limited support for massive data movement.

Numerical algorithms play crucial role in HPC and hence, several researchers have explored techniques for utilizing algorithm knowledge to improve application fault tolerance (e.g., [39, 9, 14, 18]), performance [23, 38, 19], and energy efficiency [20, 17]. Different from previous efforts, this paper demonstrates the significant benefits of using algorithm knowledge to direct data placement in the future hybrid memory system.

7. CONCLUSIONS

In this paper, we demonstrate that using algorithm knowledge, the data placement for hybrid memory can be optimized. Our approach provides valuable insights for using NVM for the future HPC systems. Based on algorithm direction, we reveal many opportunities to improve performance, energy efficiency, and implement a relaxed persistency scheme. The benefits are significant, demonstrating the feasibility to introduce algorithm semantics to address critical challenges of using the future hybrid memory systems.

Acknowledgement

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions. This work is partially supported by the U.S. Department

of Energy, Office of Science, Advanced Scientific Computing Research, the NSF grants CCF-1553645, CCF-1305622, ACI-1305624, CCF-1513201, the SZSTI basic research program JCYJ20150630114942313, and the Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase).

8. REFERENCES

[1] The parallel linear algebra for scalable multi-core architectures (plasma). http://icl.cs.utk.edu/plasma/overview/index.html/.

[2] Scalable linear algebra package. http://www.netlib.org/scalapack/.

[3] Intel quickdata technology software guide for linux. http://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf, 2008.

[4] J. Ahn et al. McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling. In *ISPASS*, 2013.

[5] K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report EECS-2006-183, UC, Berkeley, 2006.

[6] A. Bivens et al. Architectural Design for Next Generation Heterogeneous Memory Systems. In *Int. Memory Workshop*, 2010.

[7] M. Calhoun. Characterization of Block Memory Operations. *Master Thesis, Rice University*, 2006.

[8] J. Chen et al. Energy-Aware Writes to Non-Volatile Main Memory. In *Workshop on Power-Aware Computing and Systems*, 2011.

[9] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. *PPoPP*, 2013.

[10] Y. Choi et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *ISSCC*, 2012.

[11] J. Coburn et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[12] J. Condit et al. Better I/O Through Byte-Addressable, Persistent Memory. In *SOSP*, 2009.

[13] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gen. Comp. Syst.*, 22(3):303–312, 2006.

[14] T. Davies and Z. Chen. Correcting Soft Errors Online in LU Factorization. In *HPDC*, 2013.

[15] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *DAC*, 2009.

[16] X. Dong et al. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. In *SC*, 2009.

[17] R. Dorrance et al. A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs. In *FPGA*, 2014.

[18] P. Du et al. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *PPoPP*, 2012.

[19] M. Faverge, J. Herrmann, J. Langou, B. R. Lowery, Y. Robert, and J. Dongarra. Designing LU-QR hybrid solvers for performance and stability. In *IPDPS*, 2014.

[20] E. Garcia et al. Optimizing the LU Factorization for Energy Efficiency on a Many-Core Architecture. In *LCPC*, 2013.

[21] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *ISCA*, 2005.

[22] IBM. Cell broadband engine processor dma engines, part 1: The little engines that move data. http://www.ibm.com/developerworks/library/pa-celldmas.

[23] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS*, 1991.

[24] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecturing Phase Change Mmemorhy as a Scalable DRAM Architecture. In *ISCA*, 2009.

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[26] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-Ordering Consistency for Persistent Memory. In *ICCD*, 2014.

[27] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*, pages 190–200, 2005.

[28] Micron Technology. Calculating memory system power for ddr3. Technical Report TN-41-01, 2007.

[29] J. Meza et al. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. In *IEEE CAL*, 2012.

[30] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory Persistency. In *ISCA*, 2014.

[31] M. Poremba et al. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *ISVLSI*, 2012.

[32] M. K. Qureshi et al. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *MICRO*, 2009.

[33] M. K. Qureshi et al. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.

[34] L. Ramos, E. Gorbatov, and R. Bianchini. Page Placement in Hybrid Memory Systems. In *ICS*, 2011.

[35] S. Venkataraman et al. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *FAST*, 2011.

[36] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS*, 2011.

[37] B. Wang et al. Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design. In *PACT*, 2013.

[38] S. Williams et al. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC*, 2007.

[39] P. Wu et al. Fault Tolerant Matrix-Matrix Multiplication: Correcting Soft Errors On-line. *Scalable Algorithms for Large-Scale Syst.*, 2011.

[40] H. Yoon et al. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[41] J. Zhao et al. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *MICRO*, 2013.