Runtime-Guided Management of Stacked DRAM Memories in Task Parallel Programs

Lluc Alvarez
Barcelona Supercomputing Center
lluc.alvarez@bsc.es

Marc Casas
Barcelona Supercomputing Center
marc.casas@bsc.es

Jesus Labarta
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
jesus.labarta@bsc.es

Eduard Ayguade
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
eduard.ayguade@bsc.es

Mateo Valero
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
mateo.valero@bsc.es

Miquel Moreto
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
miquel.moreto@bsc.es

ABSTRACT

Stacked DRAM memories have become a reality in High-Performance Computing (HPC) architectures. These memories provide much higher bandwidth while consuming less power than traditional off-chip memories, but their limited memory capacity is insufficient for modern HPC systems. For this reason, both stacked DRAM and off-chip memories are expected to co-exist in HPC architectures, giving raise to different approaches for architecting the stacked DRAM in the system.

This paper proposes a runtime approach to transparently manage stacked DRAM memories in task-based programming models. In this approach the runtime system is in charge of copying the data accessed by the tasks to the stacked DRAM, without any complex hardware support nor modifications to the application code. To mitigate the cost of copying data between the stacked DRAM and the off-chip memory, the proposal includes an optimization to parallelize the copies across idle or additional helper threads. In addition, the runtime system is aware of the reuse pattern of the data accessed by the tasks, and can exploit this information to avoid unworthy copies of data to the stacked DRAM. Results on the Intel Knights Landing processor show that the proposed techniques achieve an average speedup of 14% against the state-of-the-art library to manage the stacked DRAM and 29% against a stacked DRAM architected as a hardware cache.

CCS CONCEPTS

• **Hardware** → Memory and dense storage; • **Software and its engineering** → Memory management;

KEYWORDS

Stacked DRAM memories, runtime systems, task-based data-flow programming models

ACM Reference Format:
Lluc Alvarez, Marc Casas, Jesus Labarta, Eduard Ayguade, Mateo Valero, and Miquel Moreto. 2018. Runtime-Guided Management of Stacked DRAM Memories in Task Parallel Programs. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205312

1 INTRODUCTION

For many years, the trend of improving the performance of High-Performance Computing (HPC) architectures by increasing the frequency and the number of cores has created a growing gap between compute capacity and memory bandwidth. As a consequence, one of the main challenges in large scale multi-core architectures is to provide enough memory bandwidth to supply sufficient data on time to the cores for highly parallel processing. In the last years, memory specifications like High Bandwidth Memory (HBM) [18] or Hybrid Memory Cube (HMC) [31] have adopted three-dimensional integration as a promising solution to the memory bandwidth problem. This technology allows to stack multiple layers of DRAM inside the package, forming a memory structure that provides significantly higher bandwidth than off-chip memories while consuming less power. Stacked DRAM memories are already present in commercial processors like the Intel Knights Landing (KNL) [40], and are expected to be a key element of next-generation HPC architectures.

The main drawback of stacked DRAM memories is the limited capacity they offer, insufficient to satisfy the memory requirements of modern HPC systems. For this reason, current and upcoming processors with stacked DRAM memories still maintain an off-chip memory, forming a memory system with two types of memories: a high bandwidth, low capacity stacked DRAM and a low bandwidth, high capacity off-chip memory. This opens the door to different approaches for architecting the stacked DRAM in the system, with clear trade-offs. On the one hand, the stacked DRAM can be architected as a large hardware cache that lays between the processor caches and the off-chip memory. This solution capitalizes on the benefits of stacked DRAM memories without modifying any software layer, but reduces the total amount of memory available for the software and complicates the hardware design of the processor. On the other hand, the stacked DRAM can be integrated in the system as a second memory visible to the software, forming a *heterogeneous memory system*. This solution reduces the hardware

complexity and maximizes the memory capacity offered to the software. However, it increases the programming complexity of the system, since it requires the programs or some software layer to explicitly manage the two types of memories.

To overcome the programmability difficulties of complex heterogeneous memory systems and architectures, task-based programming models such as OpenMP 4.0 [1] have emerged in the last years. In these programming models the programmer exposes the available parallelism of the program by dividing the code in tasks and by specifying the data and control dependencies between them. With this information the runtime system manages the parallel execution following a data-flow scheme, scheduling tasks to cores and taking care of synchronization between tasks. Decoupling the application from the architecture eases programmability and allows to leverage the runtime system information to drive optimizations in a generic and application-agnostic way [2, 10, 11, 25, 30, 41].

The goal of this paper is to exploit the benefits of stacked DRAM memories without affecting the programmability of the system. To do so, this paper proposes to leverage the characteristics of task-based programming models to give the runtime system the responsibility of managing the stacked DRAM memory, without any intervention from the programmer nor any change in the source code of the applications. The data dependence annotations of these programming models specify the data that is accessed by the tasks, so the runtime system can copy to the stacked DRAM the data needed by the tasks before they are executed. This allows to maximize the number of memory accesses that are served by the stacked DRAM, making the most of its high bandwidth and low power consumption. In addition, since copying data between the stacked DRAM and the off-chip memory has a significant cost, this paper proposes two techniques to accelerate the copy operations. The first optimization parallelizes the copies of the data across threads, using threads that are idle or additional helper threads. The second approach leverages the runtime system information on the reuse pattern of the data accessed by the tasks to decide when it is worth copying the data to the stacked DRAM and when it is better to avoid the copy and to operate with the data in the off-chip memory. The main contributions of this paper are:

- A novel runtime system design to transparently manage a heterogeneous memory system formed by a stacked DRAM and an off-chip memory. These extensions allow task parallel programs to benefit of such memory organization without introducing any programming burden.
- Two techniques to mitigate the cost of copying data between the stacked DRAM and the off-chip memory. The first solution parallelizes the copy operations using idle threads and helper threads, while the second approach leverages the information of the data reuse pattern in the runtime system to avoid unnecessary costly data copies.
- A complete evaluation on the KNL with relevant HPC benchmarks. The results highlight the importance of the data reuse pattern for the performance of state-of-the-art techniques. The proposed runtime approach is able to adapt to the characteristics of the applications and always achieves the best performance, reaching average speedups of 14% against other software approaches and 29% against hardware caches.

This paper is organized as follows: Section 2 introduces stacked DRAM memories and task-based programming models. Section 3 explains the proposed runtime system techniques to manage the stacked DRAM. Section 4 describes the experimental framework for the evaluation, which is presented in Section 5. Section 6 discusses the related work, and Section 7 draws the conclusions of this work.

2 BACKGROUND AND MOTIVATION

This section explains the main properties of stacked DRAM memories and the ways to architect them in a system. Then it describes the characteristics of task-based programming models and their opportunities for managing these memories.

2.1 Stacked DRAM Memories

In recent years stacked DRAM memories have become a promising solution to aid the bandwidth problems of parallel architectures. Die-stacking provides significant bandwidth and power consumption benefits compared to conventional off-chip memories. The *Through-Silicon Via (TSV)* interface of die-stacking circumvents the pin-count limitations of off-chip memories, so the bandwidth to the stacked DRAM is limited by the parallelism in the memory itself and not by the interface with the processor. This allows stacked DRAM to provide a much higher bandwidth than off-chip memories. In terms of energy per bit, the TSV interface provides an improvement of two orders of magnitude over DDR3 interfaces [29]. However, the capacity of stacked DRAM modules is much lower than the capacity of off-chip memory modules, so both types of memories co-exist in current and, expectedly, future systems. In such scenario, the stacked DRAM can be architected in the system in different ways.

One approach is to architect the stacked DRAM as a large hardware cache between the last-level cache of the processor and the off-chip memory. This approach, known as *DRAM cache*, is totally transparent to the software, so no modifications to the software stack are required. However, DRAM cache designs face the challenge of making affordable the access latency, storage and energy consumption overheads associated with tag management of such a big cache. Although different designs have been proposed in the literature[1], the most simple and effective approach is to organize the DRAM cache as a direct-mapped cache with the same block size as the processor caches, as proposed by the state-of-the-art Alloy Cache [34] and implemented in the KNL.

Stacked DRAM memories can also be architected as *part of memory*. In this approach the stacked DRAM is mapped to a range of the physical address space, visible to the software. This increases the total memory capacity of the system and avoids the tag storage overheads of DRAM caches. However, the main challenge of this approach is that the software has to manage an *heterogeneous memory system* formed by two memories with different characteristics.

Although some works propose to manage heterogeneous memory systems in the operating system[1], current systems rely on the programmer to do so. The KNL offers the possibility to expose the stacked DRAM as a NUMA memory node that is managed with NUMA libraries (i.e. `memkind` [16]). These libraries offer command line options and environment variables to specify the memory in which the data of the program has to be allocated, but does not

[1]These proposals are discussed in detail in Section 6

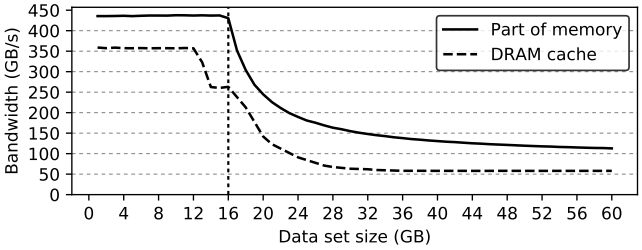


Figure 1: Memory bandwidth measured with the Stream Triad benchmark on a KNL when configuring the stacked DRAM as part of memory or as a DRAM cache.

allow to select in which memory each individual data object is allocated. To do so, the programmer has to allocate the variables with a special form of `malloc` in the source code. These solutions manage the heterogeneous memory system *statically*, that is, allocating data in the stacked DRAM or in the off-chip memory at the beginning of the program and not changing the placement of the data during the execution. In HPC programs, with input sets that exceed the size of the stacked DRAM, static solutions are not optimal because only a small portion of the data is served by the stacked DRAM, while the rest of data is served by the off-chip memory.

A better solution for HPC workloads is to *dynamically* copy to the stacked DRAM the data that is going to be accessed by the tasks, so all the memory accesses in the computation are served by the stacked DRAM and its benefits are maximized. Unfortunately, this requires to transform the code of the application, typically applying tiling to split the computation in blocks and to copy to the stacked DRAM the data of the block that is going to be computed. These code transformations are not trivial, and relying on the programmer to do them imposes a clear programming burden.

2.2 Stacked DRAM Characterization

The existing solutions to manage stacked DRAM memories suffer from some inefficiencies that are exacerbated when the data set of the application does not fit in the stacked DRAM.

Figure 1 shows the memory bandwidth measured with the Stream Triad benchmark [27] on the KNL for different input set sizes. The capacity of the stacked DRAM in the KNL is 16 GB. The two lines show the measured bandwidth when the stacked DRAM is configured as a DRAM cache and when it is configured as part of memory, using the `memkind` library to allocate as much data as possible in the stacked DRAM. It can be observed that, when the stacked DRAM is configured as part of memory, the measured bandwidth reaches 435 GB/s for any data set size smaller than 16 GB, while the DRAM cache achieves 360 GB/s for data sets of up to 12 GB. The performance differences happen because the latency of the stacked DRAM varies depending on the configuration. When the stacked DRAM is configured as part of memory, its latency is slightly higher than the off-chip memory latency [35] and, when it is configured as a DRAM cache, the latency is even higher due to the extra circuitry [34].

Figure 1 also shows that, when the data set is bigger than the size of the stacked DRAM, the performance drops. This happens because, when the stacked DRAM is configured as part of memory, only a portion of the data can be allocated in the stacked DRAM. So, the bigger the data set, the more memory accesses are served by the off-chip DRAM, causing the performance degradations. The performance of the DRAM cache also drops gradually as data sets get bigger. When the data set exceeds 12 GB conflict misses start to happen in the DRAM cache. The number of DRAM cache misses augments as the data set gets bigger, reaching a point where all the memory accesses miss in the DRAM cache and the performance is limited by the bandwidth of the off-chip memory serving misses.

An important aspect of the stacked DRAM as part of memory is the cost of copying data between the stacked DRAM and the off-chip memory. In the KNL, the only way to copy data between the memories is by means of a `memcpy`. Thus, copying data requires one thread to execute loads and stores in a loop to read blocks of data from one memory, bring them to the CPU, and store them in the other memory. This mechanism has a significant cost, since it heavily utilizes resources of the core and the cache hierarchy.

2.3 Task-Based Programming Models

Task-based programming models have become an appealing alternative to face the programmability challenges imposed by the heterogeneity of current and, expectedly, future HPC architectures. This tasking model is currently supported by the OpenMP 4.0 [1] shared memory programming standard.

Task-based data-flow programming models conceive the execution of a parallel program as a set of tasks with dependences between them. The programmer adds code annotations to split the serial code in *tasks* and to specify what data is read (*inputs*) and written (*outputs*) by each task. Typically, the compiler encapsulates the tasks in functions that receive by parameter the addresses of the inputs and outputs. The runtime system manages the execution of the tasks by means of a *Task Dependence Graph (TDG)*, a directed acyclic graph where the nodes are tasks and the edges are dependences between them. Following a decoupled execution model, threads first execute the application code (creating all the tasks they encounter) until they reach a global synchronization point, and then they execute tasks asynchronously. The runtime system dynamically schedules tasks when all their inputs are ready and, when the execution of a task finishes, its outputs become ready for the next tasks. This model decouples the hardware from the application, enabling many optimizations at the runtime system level in a generic and application-agnostic way [2, 10, 11, 25, 30, 41].

This paper proposes to exploit the characteristics of task-based programming models to manage stacked DRAM memories in the runtime system without affecting programmability. A key characteristic of task-based programming models is that the inputs and outputs of the tasks specify the data they access. The runtime system can leverage this information to copy to the stacked DRAM the data that is accessed by the tasks before they are executed. As a result, the memory accesses to the inputs and outputs of the tasks are served by the stacked DRAM, maximizing its bandwidth and power consumption benefits. Another characteristic of task-based programming models is that the runtime system is in charge of managing the threads that participate in the parallel execution. This allows the runtime system to accelerate the copies of the data by parallelizing them, using either threads that are idle or additional helper threads specifically devoted to perform the copy operations. The runtime system is also aware of the reuse pattern of the data

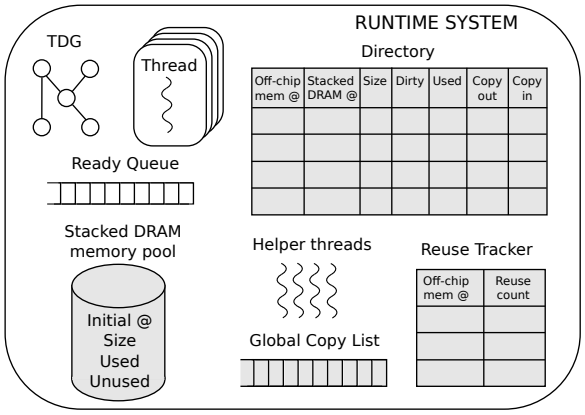


Figure 2: Components of the runtime system and extensions to manage stacked DRAM memories (shaded in gray).

that is accessed by the tasks, which can be used to avoid unworthy data copies. In particular, the runtime system can decide to avoid copying to the stacked DRAM data that is only going to be accessed by a single task. Instead, the task performs the computation accessing data in the off-chip memory, saving a copy operation that is more costly than the off-chip memory accesses themselves.

3 STACKED DRAM MANAGEMENT IN TASK RUNTIME SYSTEMS

This section presents the design of a task-based runtime system to manage the stacked DRAM of a heterogeneous memory system.

3.1 Overview

The goal of this paper is to efficiently manage the stacked DRAM of a heterogeneous memory system without affecting programmability. To achieve this goal, the runtime system dynamically maps to the stacked DRAM the data that is going to be used by the tasks. This approach maximizes the amount of memory accesses served by the stacked DRAM, making the most of its advantages in performance and power consumption. The proposal does not introduce any programming burden, as no modifications in the source code of the applications are required.

The main data structure of the design is a directory that tracks the data mapped to the stacked DRAM, enabling the runtime system to manage this memory as a software cache. Before a task is executed, the runtime system is in charge of copying the inputs and outputs of the task to the stacked DRAM. The directory is used to check if the data is already present in the stacked DRAM or if it has be copied from the off-chip memory. This may require replacing some data of the stacked DRAM to make room for the new data and triggering copies of data between the stacked DRAM and the off-chip memory.

Since the copies of data can have a significant cost, we propose two optimizations to mitigate their associated overheads. The first optimization consists on parallelizing the data copies among multiple threads, using threads that are idle or additional helper threads. The second optimization avoids mapping to the stacked DRAM data that is not reused, as the cost of copying the data between the two memories is higher than the cost of accessing data in the off-chip memory during the execution of a single task.

3.2 Data Structures

Figure 2 shows the data structures of the proposed runtime system, with the extensions to manage the stacked DRAM shaded in gray. The non-shaded data structures are the main parts of a generic runtime system for task-based programming models, which consists of a representation of the TDG, the threads that participate in the execution, and a ready queue where the threads request tasks.

We add a memory pool to represent the stacked DRAM in the runtime system. The memory pool is allocated in the stacked DRAM when the program starts, and the runtime system stores its initial address, its size in bytes, and the amount of used and unused bytes.

The directory is the most important data structure for the management of the stacked DRAM. The directory is formed by a set of entries that associate blocks of data in the off-chip memory with its corresponding blocks of data in the stacked DRAM. To do so, each directory entry contains the address of the data in the off-chip memory, the address of the data in the stacked DRAM, the size of the data, a dirty flag to indicate whether the data in the stacked DRAM has been modified, a used counter to indicate how many executing tasks access the data in the stacked DRAM, and two pointers to two *copy descriptors* that describe the parameters of the copies of data into the stacked DRAM (CopyIn) and out of the stacked DRAM (CopyOut). We implement the directory with a C++ hash map using the off-chip address as key.

The copy descriptors store the attributes of the copies of data that need to be performed. Each copy descriptor contains the source address, the destination address, the size in bytes, a finished flag to indicate that the copy has finished, and a chunk offset that keeps the amount of bytes that has already been copied. This last field is used to split the copy in multiple chunks, so it can be parallelized.

Additional data structures are introduced in the runtime system to perform optimizations. The helper threads and the Global Copy List are used for the parallelization of copies of data, while the Reuse Tracker is used for the reuse-aware bypass of the stacked DRAM. These optimizations and their data structures are explained in detail in Sections 3.4 and 3.5. In the next section, we describe the basic operational model without such optimizations.

3.3 Operational Model

The aforementioned data structures are operated during diverse phases of the execution of a task parallel program.

When the program starts, the runtime system allocates and initializes all its internal data structures. To initialize the stacked DRAM memory pool the runtime system assumes that the whole capacity of the stacked DRAM is available, although the system administrator or the user can use an environment variable to adjust the amount of stacked DRAM memory that the runtime system can manage. A buffer of the specified size is allocated in the stacked DRAM and used as the stacked DRAM memory pool. The initial address of the memory pool is set to the starting address of the buffer, the size and the unused bytes fields are set to the specified size, and the used bytes field is set to zero.

When a task is scheduled for execution, its inputs and outputs are mapped to the stacked DRAM. For each input and output of the task, its address is looked up in the directory, which can lead to several situations. Figure 3 shows a scheme of the possible cases.

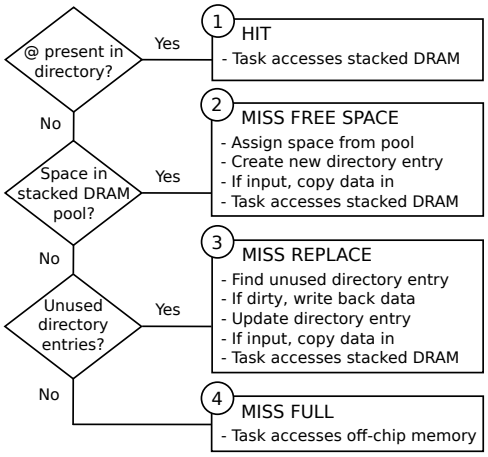


Figure 3: Scheme of the possible situations that can arise for data being mapped to the stacked DRAM.

1 - *HIT*: If a matching directory entry is found, the stacked DRAM already contains the data being mapped. The used counter of the directory entry is incremented by one and, if the data being mapped is an output, the dirty flag is set. The stacked DRAM address is passed by parameter to the task, so it will access the stacked DRAM.

2 - *MISS FREE SPACE*: If a match is not found in the directory and the size of the data being mapped is smaller than the unused bytes of the stacked DRAM memory pool, a new directory entry is created. The fields of the directory entry are initialized as follows: the off-chip memory address is set to the address of the data being mapped; the stacked DRAM address is set to the stacked DRAM memory pool starting address plus the used bytes; the size is set to the size of the input or output being mapped; the dirty flag is set if the data being mapped is an output; and the used counter is set to one. Then, if an input is being mapped, a copy descriptor is created to copy the data from the off-chip memory to the stacked DRAM. The source address of the copy descriptor is set to the address of the input being mapped, the destination address is set to the stacked DRAM address of the directory entry, the size is set to the size of the input being mapped, and the finished flag and the chunk offset are set to zero. A pointer to the copy descriptor is stored in the CopyIn field of the directory entry. Then, the used and unused bytes of the stacked DRAM memory pool are respectively incremented and decremented by the size of the data being mapped. Finally, the stacked DRAM address is passed by parameter to the task.

3 - *MISS REPLACE*: If a match is not found in the directory and the size of the data being mapped is larger than the unused bytes of the stacked DRAM memory pool, a directory entry is replaced from the stacked DRAM to make room for the data being mapped. The directory is sequentially traversed, and the first entry with the same size as the data being mapped and with the used counter set to zero is selected as a victim for the replacement. If the directory entry has the dirty flag set, a copy descriptor is created to write back the data from the stacked DRAM to the off-chip memory. The source address, the destination address and the size of the copy descriptor are respectively set to the stacked DRAM address, the off-chip memory address, and the size of the directory entry, and a pointer to the copy descriptor is stored in the CopyOut field of the directory entry. Then the fields of the directory entry are modified to reflect the mapping with the new data. To do so, the off-chip memory address is set to the address of the input or output being mapped, the dirty flag is set if an output is being mapped, and the used counter is set to one. Finally, if an input is being mapped, a copy descriptor is created to copy the data from the off-chip memory to the stacked DRAM. The source address of the copy descriptor is set to the address of the input being mapped, the destination address is set to the stacked DRAM address of the directory entry, the size is set to the size of the input being mapped, and the finished flag and the chunk offset are set to zero. A pointer to the copy descriptor is stored in the CopyIn field of the directory entry and the stacked DRAM address is passed to the task.

4 - *MISS FULL*: If a match is not found in the directory, all the directory entries ares used, and there is no space available in the stacked DRAM pool, the data cannot be mapped to the stacked DRAM because its whole capacity is devoted to data being accessed by running tasks. In this situation the data is not mapped to the stacked DRAM and the off-chip memory address is passed by parameter to the task, so it will access the data in the off-chip memory.

After mapping the data of a task to the stacked DRAM, the copies of data specified in the copy descriptors take place. To do so, each input and output of the task is looked up in the directory, and the two copy descriptor pointers of the matching entry are retrieved. The CopyOut is processed before the CopyIn. For each copy descriptor a `memcpy` takes place using the source address, the destination address and the size specified in the copy descriptor. The chunk and the finished flag of the copy descriptor are used only when the copies are parallelized across threads, as explained in the next section. When the copies of data finish the copy descriptors are destroyed and the execution of the task starts.

Finally, when a task finishes its execution, the address of each input and output is looked up in the directory and the used counter of the matching directory entry is decremented.

3.4 Parallelization of Copies of Data

Copying data between the stacked DRAM and the off-chip memory is a costly operation, as explained in Section 2.2. An approach to mitigate the overheads of the copies of data is to parallelize them across multiple threads. To do so, the copies of data are split in chunks and threads are allowed to copy different chunks in parallel. Chunks can be requested by threads that are idle or by additional helper threads that can only perform this specific duty. To enable this technique, several extensions are introduced in the runtime system. Two environment variables are also added to adjust the size of the chunks and the number of helper threads.

The Global Copy List is added to the runtime system to keep the in-flight data copies. When data is mapped to the stacked DRAM and a copy descriptor is created, a pointer to the copy descriptor is inserted in the Global Copy List.

The behaviour of the threads is slightly changed to enable the parallel data copies. As explained in the previous subsection, before a task is executed, a `memcpy` is performed for every copy descriptor of the inputs and outputs of the task. In order to parallelize the copies, instead of performing a single `memcpy` for each copy descriptor, the thread iterates on a loop while the finished flag of the copy

descriptor is unset. In every iteration of the loop, the thread requests a chunk of the copy, which consists on computing the source and destination addresses for the chunk (adding the chunk offset to the source and destination addresses of the copy), incrementing the chunk offset by the chunk size, performing the `memcpy` of the chunk and, if the copied chunk is the last chunk of the copy, setting the finished flag. Once the finished flag is set, the thread exits the loop and is ready to proceed with the next copy descriptor.

Idle threads and helper threads can request chunks of any in-flight copy, effectively parallelizing it. To do so, idle and helper threads request a chunk of the copy descriptor at the head of the Global Copy List. If the requested chunk is the last chunk of the copy, the idle or helper thread removes the copy descriptor from the Global Copy List, but does not perform any action on the copy descriptor pointers of the directory entries. This is done by the thread that will execute the task, as explained in the previous paragraph, and is used as a synchronization mechanism to ensure that the parallel copies of data have finished before the task is executed.

3.5 Reuse-Aware Bypass

Another way to mitigate the overheads of the copies of data is to avoid the copies that will not be amortized. In particular, if the data is only accessed once, the cost of copying the data is higher than the benefits of accessing the stacked DRAM instead of the off-chip memory during the computation. For this reason, we propose a mechanism that allows tasks to access data in the off-chip memory when copying the data to the stacked DRAM is not worth.

The Reuse Tracker is added to the runtime system to track the data reuse pattern of the application. The Reuse Tracker keeps, for every block of data declared as input or output of any task, a reuse counter that counts the number of tasks that read or write it.

The Reuse Tracker is updated when tasks are created and executed. At task creation, the address of each input and output is looked up in the Reuse Tracker and its corresponding reuse counter is incremented by one. When a task finishes its execution, its input and output addresses are looked up in the Reuse Tracker and the reuse counters of the matching entries are decremented by one.

The information in the Reuse Tracker is used when the inputs and outputs of a task are mapped to the stacked DRAM, introducing minor changes to the basic operation explained in Section 3.3. An input or output is always mapped to the stacked DRAM if the data is already there (case 1, HIT) or if the data is not present in the stacked DRAM but there is space available in the stacked DRAM memory pool (case 2, MISS FREE SPACE). If these two cases do not happen, the Reuse Tracker is consulted before trying to replace any unused data in the stacked DRAM to make room for the new data. The address of the data being mapped is looked up and, if the reuse counter of the matching entry is equal to one, the stacked DRAM is bypassed and no replacement is attempted. The same address being mapped is passed by parameter to the task, so it will access the data in the off-chip memory during the execution. If the reuse counter of the data being mapped is higher than one the basic operation continues normally, trying to replace some unused data from the stacked DRAM (case 3, MISS REPLACE) and not mapping the data to the stacked DRAM if all its capacity is occupied with data that is being used by other running tasks (case 4, MISS FULL).

3.6 Additional Considerations

The proposed runtime system mechanisms assume that the blocks of data specified as task inputs and outputs present either complete or no overlap. This is done in accordance with the OpenMP 4.0 specification, that does not allow inputs and outputs with partial overlap. Other task-based programming models like OmpSs allow partially overlapped inputs and outputs, and propose runtime system mechanisms to support them in systems with multiple address spaces [8]. Similar mechanisms could be added to our design.

The proposed design does not include any defragmentation mechanism for the stacked DRAM. Note that, in HPC applications, the computation is usually split in equally-sized blocks, so the size of the inputs and outputs of the tasks is very regular. Thus, when a block of data is mapped to the stacked DRAM, it is very likely that an unused block of the same size will be found in the directory, so the equally-sized blocks can be replaced without causing any fragmentation. Our design leaves the data in the off-chip memory in case there are no unused blocks with the same size as the data being mapped, but defragmentation mechanisms could be added.

4 EXPERIMENTAL FRAMEWORK

The proposed ideas have been evaluated on an Intel Xeon Phi 7250 processor [40]. Its memory system consists of a 96 GB DDR4 off-chip memory at 1200MHz and a 16 GB MCDRAM stacked memory at 7200MHz. The processor contains 68 cores running at 1.4GHz, and each core has a 2-wide out-of-order pipeline with SMT support for 4 hardware threads, 512-bit vector units (AVX-512), and private L1 data and instruction caches of 32 KB each. The KNL is organized in 34 tiles, where each tile contains 2 cores, a 1 MB L2 cache shared by the 2 cores of the tile, and a portion of the distributed tag directory for the cache coherence protocol. The 34 tiles are interconnected with a 2D mesh configured in Quadrant clustering mode.

The KNL runs a Linux operating system with kernel version 3.12.64. The runtime system for the task-based programming model is Nanos [13] 0.12a (rev. 46307e2), which natively supports the OpenMP 4.0 task constructs. The benchmarks are compiled with the Mercurium [4] 2.0.0 (rev. 76f98c4) source-to-source compiler, using Intel ICC 17.0.1 as a backend compiler. The Intel MKL library is used for the mathematical kernels of the benchmarks.

A set of representative HPC benchmarks, shown in Table 1, are used in the evaluation. The benchmarks are programmed in OpenMP 4.0, using annotations to specify the tasks and their inputs and outputs. Chol calculates a Cholesky decomposition of a triangular matrix from a symmetric and positive definite matrix. DGEMM performs a dense matrix multiplication of two square matrices. LU calculates a LU decomposition of a sparse matrix as the product of a lower triangular matrix and an upper triangular matrix. QR computes a factorization of a matrix A as a product $A = Q \cdot R$, where Q is orthogonal and R is upper triangular. SMI computes the inverse of a real, positive definite, symmetric matrix. Gauss solves the stationary heat diffusion problem using the Gauss-Seidel method with a 4-element stencil, following a wave-front parallelization strategy. Jacobi solves the stationary heat diffusion problem using the Jacobi method with a 5-element stencil, using two matrices and an embarrassingly parallel algorithm. RedBlack solves the stationary heat diffusion problem with a 4-element stencil using two sub-iterations

Table 1: Benchmark parameters

Benchmark	Reuse	Small input			Large input		
		Size	Tasks	Scal	Size	Tasks	Scal
Chol	✓	8	5984	64	72	1198144	64
DGEMM	✓	6	4352	64	54	112896	64
LU	✓	8	91018	256	32	91018	256
QR	✓	6	93536	64	80	597620	64
SMI	✓	8	17952	64	72	3594432	64
Gauss	x	8	208192	256	72	237792	192
Jacobi	x	9	210176	64	64	430592	64
RedBlack	x	8	208192	192	72	237792	192
Stream	x	12	10496	64	30	10496	64

with embarrassing parallelism. Stream is a benchmark to measure memory bandwidth, from which we evaluate the Triad test.

Table 1 shows the setup of the benchmarks. The table contains a Reuse column that specifies if the data accessed by the tasks is reused inside the parallel regions of the program. Chol, DGEMM, QR, SMI and LU contain a single parallel region where the whole computation takes place, and the same piece of data is accessed multiple times by different tasks inside the parallel region. Gauss, Jacobi, RedBlack and Stream are iterative algorithms where every iteration consists of a parallel region, and the same piece of data is only accessed once by a single task inside each parallel region.

Table 1 also shows the benchmark parameters for the two input sets used in the evaluation, including the size of the data set (in GB) and the number of tasks. The large input set does not fit in the stacked DRAM, which is the most representative case for HPC applications, while the small input set fits fits in the stacked DRAM.

For each benchmark and input set we explore different task granularities running with different numbers of threads (16, 32, 64, 128, 192 and 256). We study the parallel executions with visual performance analysis tools and we select the best scaling task granularity. The scalability is reflected in the *Scal* column of Table 1, which shows the best performing number of threads for each benchmark and input set. It can be observed that all the benchmarks achieve good scalability, reaching its highest performance at 64 threads in 6 of the 9 benchmarks. The other 3 benchmarks benefit from the SMT capabilities of the cores and scale up to 192 or 256 threads.

5 EVALUATION

5.1 Performance Results

This section evaluates the performance of the proposed techniques. We first consider input sets that do not fit in the stacked DRAM.

Figure 4 shows the speedup of the proposed mechanisms with respect to state-of-the-art solutions. Six bars are presented per benchmark: *Off-chip* allocates all the data in the off-chip memory and does not use the stacked DRAM; *Cache* manages the stacked DRAM as a hardware cache; *PoM* configures the stacked DRAM as part of memory and the benchmarks use the `memkind` library to allocate as much data as possible in the stacked DRAM; *Runtime* uses the ideas proposed in this paper to manage the stacked DRAM in the runtime system without including any optimization; *Runtime+ParCopies* manages the stacked DRAM in the runtime system with support for parallel data copies; and *Runtime+Reuse* manages the stacked DRAM in the runtime system and bypasses the stacked DRAM for non-reused data. All results are normalized against *PoM*.

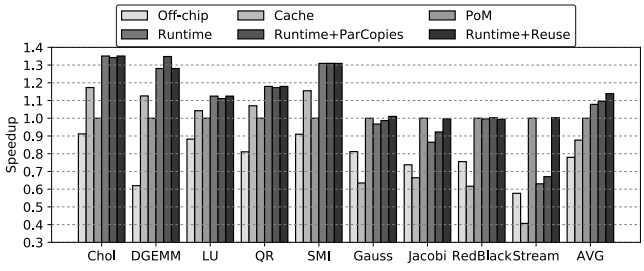


Figure 4: Performance when allocating all data in the off-chip memory (Off-chip), configuring the stacked DRAM as a cache (Cache) or part of memory (PoM), and managing it in the runtime system (Runtime) with parallel data copies (Runtime+ParCopies) or with reuse-aware bypass of copies (Runtime+Reuse). All results are normalized to PoM.

Off-chip achieves worse performance than *PoM* in all the evaluated benchmarks (22% on average) due to the lower bandwidth of the off-chip memory. In the case of *Cache*, it outperforms *PoM* in all benchmarks with reuse (11% on average), but suffers significant performance degradation in the benchmarks without reuse (42% on average). As shown in Section 2.2, the attained bandwidth in cache mode is below the bandwidth of the external memory for large data sets with no reuse, as discussed in Section 2.2 (Figure 1).

For the three runtime-based approaches, Figure 4 shows two clear trends depending on the data reuse pattern of the benchmarks. For those with data reuse (Chol, DGEMM, LU, QR, and SMI), the three runtime-based approaches clearly outperform *PoM*, achieving speedups between 11% and 35%. These speedups are granted by the ability of the runtime system to dynamically copy to the stacked DRAM the data that is going to be accessed by the tasks, reducing their execution time. The first time an input or output is accessed by a task, the runtime system maps it to the stacked DRAM, paying the cost of the data copy. Then the following tasks that reuse this data already find it in the stacked DRAM and can capitalize on its benefits without any added cost. These operations add very modest performance overheads, as less than 1% of the total execution time is spent mapping inputs and outputs to the stacked DRAM, and less than 6% is spent copying data between the two memories. In addition, the overheads of the copies are reduced when they are parallelized, providing performance gains of up to 6% over *Runtime* in DGEMM. The next subsection evaluates in depth the effect of parallelizing the copies and adding helper threads.

For benchmarks without data reuse (Gauss, Jacobi, RedBlack and Stream), Figure 4 shows that *PoM* and *Runtime+Reuse* achieve the same performance, while the other two runtime-based approaches (*Runtime* and *Runtime+ParCopies*) perform significantly worse, up to 37% in Stream. This happens because, when data is not reused, there is no benefit from copying them to the stacked DRAM. As such, the *Runtime* approach is slower than *PoM* due to these unworthy copies of data, that add significant overheads in Jacobi (17%) and Stream (63%). Parallelizing data copies achieves some performance improvements in these benchmarks, 4% on average, but it is still slower than *PoM* and *Runtime+Reuse*. The best option in these benchmarks is to bypass the stacked DRAM and avoid any data copy. The ability of *Runtime+Reuse* to detect non-reused data allows it to identify such situation without any programmer intervention.

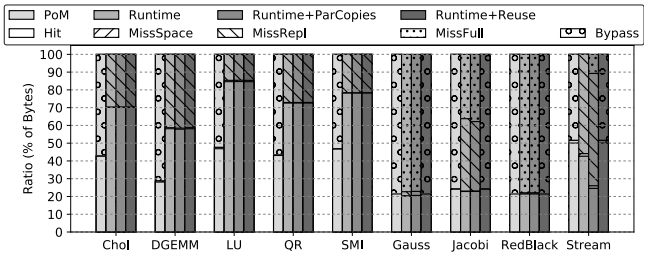


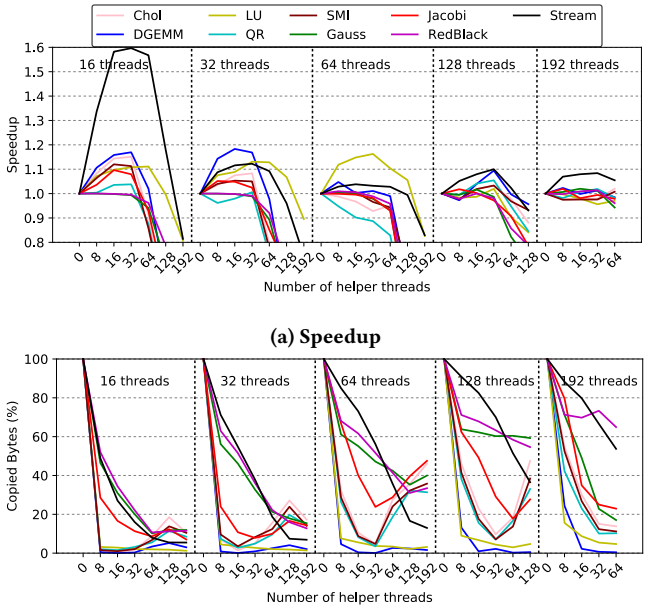
Figure 5: Stacked DRAM hit, miss and bypass ratios. Misses are mapped to the stacked DRAM if there is available space (MissSpace) or with a replacement (MissReplacement). If the whole stacked DRAM is in usage (MissFull) or the data is not reused (Bypass), accesses are served by the off-chip memory.

In order to better explain the presented performance numbers, Figure 5 reports the percentage of bytes specified in the inputs and outputs of all the tasks that are served by the stacked DRAM. This figure characterizes the different situations that can happen during the map operation in the runtime system (described in Section 3): the data already is in the stacked DRAM (*Hit*, case 1); the data is not in the stacked DRAM but there is space available (*MissSpace*, case 2); the data is not in the stacked DRAM and some other data is replaced (*MissRepl*, case 3); the data is not in the stacked DRAM and the whole capacity is occupied with data used by running tasks (*MissFull*, case 4); and finally the data has no reuse and the stacked DRAM is bypassed (*Bypass*). The figure includes results for the three *Runtime* variants and for *PoM*. To obtain the measurements for *PoM*, we emulate its behaviour in the runtime system. To do so, inputs and outputs are mapped to the stacked DRAM as long as there is space available in the pool and, when the stacked DRAM becomes full, the runtime system never replaces any data. Instead, inputs and outputs are served by the stacked DRAM if they hit in the directory, and are served by the off-chip memory otherwise. This effectively emulates the behaviour of *PoM* and achieves the same performance in all the benchmarks.

Figure 5 shows that, for the benchmarks with data reuse, *PoM* presents hit ratios between 28% and 48%, while the rest of the data is served by the off-chip memory. The three runtime-based approaches achieve higher hit ratios, from 59% to 85%. For these benchmarks, *Runtime+Reuse* never bypasses the stacked DRAM because all the data is used by multiple tasks.

For benchmarks without data reuse, *PoM* and *Runtime+Reuse* offer the same performance, while the other approaches perform worse. The best option in these cases is to always bypass the stacked DRAM to avoid any copy of data. As shown in Figure 5, *Runtime+Reuse* and *PoM* bypass the stacked DRAM for any data that does not hit in the directory. In contrast, the other two runtime approaches do unworthy replacements of data in the stacked DRAM.

Gauss and RedBlack scale up to 192 threads with coarse task granularities, so the inputs and outputs of the in-flight tasks exceed the capacity of the stacked DRAM. For this reason, in *Runtime* and *Runtime+ParCopies*, 79% of the inputs and outputs are not copied to the stacked DRAM due to lack of space. As a consequence, the three runtime approaches behave like *PoM* in Gauss and RedBlack, achieving the same performance. We observe that, with suboptimal task granularities that fit in the stacked DRAM, results for Gauss and RedBlack follow the same trend as Jacobi and Stream.



(a) Speedup

(b) Percentage of copied bytes

Figure 6: Speedup and percentage of bytes copied by the compute threads with parallel copies and helper threads.

All together, *Runtime+Reuse* achieves the best performance results with an average 1.14x speedup over *PoM*. One the one hand, *PoM* delivers good performance in benchmarks without reuse, as copying the data to the stacked DRAM is not worth in these cases. On the other hand, *Cache* and *Runtime* dynamically copy the data accessed by the tasks to the stacked DRAM, which significantly improves performance of benchmarks with reuse. *Runtime+ParCopies* reduces the cost of the copy operations and further increases the performance gains of *Runtime*. Finally, *Runtime+Reuse* appears as a general and efficient solution for all the benchmarks, given its ability to efficiently manage the stacked DRAM according to the characteristics of any application.

5.2 Performance of Parallel Data Copies

Figure 6 presents an extensive experimental campaign to evaluate the parallelization of data copies. The two plots show results for executions with 16, 32, 64, 128 and 192 compute threads, separated by vertical lines. For each number of compute threads, the X axis considers adding 0, 8, 16, 32, 64, 128 and 192 helper threads (without exceeding the maximum 272 threads of the KNL). Figure 6a shows the speedup of each configuration with respect to the same number of compute threads and no copy threads, and Figure 6b shows the percentage of bytes copied by the compute threads.

The parallelization of the copies of data and the addition of helper threads achieve important performance improvements, specially for executions with up to 64 threads in total. Results show that, when 16 or 32 compute threads are used, adding up to 32 helper threads achieves average speedups of 14% and 7%, respectively. In these cases, the performance of Chol, DGEMM, Jacobi, LU and SMI improves by 10% to 20%, and Stream achieves a maximum speedup of 1.6x with 16 compute threads and 32 helper threads. As shown in

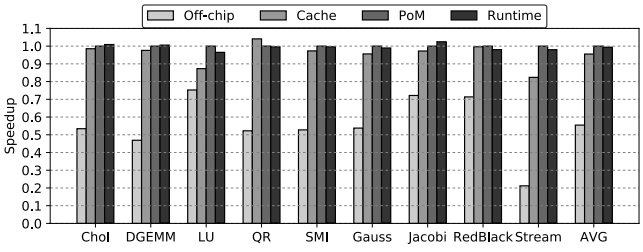


Figure 7: Performance with input sets that fit in the stacked DRAM. All results are normalized to PoM.

Figure 6b, these speedups are obtained because most of the copies are done by the helper threads, from 70% to 99% of the total copied bytes in most benchmarks. Gauss and RedBlack have very little copies of data so the performance gains are very low.

The performance improvements are not so relevant when the total number of threads exceeds 64. This can be observed in the results that use 64 compute threads with 32 or more helper threads, or 32 compute threads with 64 or more helper threads. In these cases the parallel copies provide very little benefits and can even degrade performance, even though the percentage of bytes copied by the helper threads is still large. This happens because the KNL has 68 cores with support for 4 hardware threads per core so, when the total amount of threads grows past 68, the threads start competing for functional units, cache ports and other microarchitectural resources of the cores. This slows down all the phases of the program, including the execution of the tasks, the copies of data and other runtime system phases. When using 128 or 192 compute threads, the effects of adding helper threads are not very relevant. This happens because, even without adding helper threads, the resources of the cores are already contended. Moreover, some benchmarks do not scale past 64 threads, so some executions are dominated by runtime system and synchronization overheads.

As a conclusion, parallelizing the data copies achieves important performance gains, from 10% to 20% in many benchmarks when using up to 64 threads. These speedups are granted by the ability of the helper threads to perform a large percentage of the copies, from 70% to 99% of the total bytes. However, this optimization is not so effective in executions with more than 64 threads due to the contention in the microarchitectural resources of the cores.

5.3 Performance with Small Input Sets

Next we study the performance when the input sets fit in the stacked DRAM. Figure 7 shows the results of *Off-chip*,*Cache*, *PoM* and *Runtime*. The other runtime solutions (*Runtime+ParCopies* and *Runtime+Reuse*) show the same performance as *Runtime*, and are obviated for clarity. All results are normalized to PoM.

The most important conclusion from Figure 7 is that our proposals achieve the same performance as *PoM*. *PoM* statically allocates the whole data set in the stacked DRAM with no overhead, while *Runtime* only adds a 1% overhead to automatically manage the stacked DRAM at runtime. *Cache* obtains an average slowdown of 5% due to the reduced performance in LU and Stream. *Off-chip* is the worse approach, with an average 44% slowdown with respect to *PoM*. Thus, even in the most advantageous scenario for *PoM*, the proposed runtime system solutions provide the same performance.

6 RELATED WORK

6.1 DRAM Caches

DRAM caches can be categorized in two classes.

Block-based DRAM caches like the one in the KNL store data and tags using a cache block granularity (i.e. 64 bytes), aiming to maximize the effective capacity of the cache and to efficiently exploit temporal locality. The main problem of this class of DRAM caches is the storage requirements for the tags, that are too large to be placed in SRAM, so they have to be stored in the stacked DRAM. This limits the final capacity of the DRAM cache and imposes serial tag-data accesses, increasing the hit latency and complicating the design of set-associative caches. To overcome the latency problem, the Loh-Hill Cache [24] stores both tags and cache blocks in the same DRAM row, so a single row access can serve the tag and the block access. The Alloy Cache [34] collocates the tag and the block together to further reduce the access latency. Sim et al. [39] introduce a hit/miss predictor for the DRAM cache and the ATCache [17] accelerates tag accesses by means of a SRAM-based tag cache.

Page-based DRAM caches use a page granularity to reduce the storage overhead of tags. However, multi-gigabyte DRAM caches still require a tag store of tens of megabytes, so the tags still need to be stored in the stacked DRAM. Page-based DRAM caches suffer from very high miss latencies, as replacing data at a page granularity is a costly operation, and from making a suboptimal utilization of the cache capacity and of the off-chip bandwidth if only parts of the pages are accessed. To tackle these problems, CHOP [21] improves the efficiency of a page-based DRAM caches by caching hot pages only. The Footprint Cache [20] and the Unison Cache [19] track which blocks of the page have been accessed and selectively fetch only those blocks that are likely to be used in the future.

6.2 Software Management of Heterogeneous Memory Systems

Some works propose to manage heterogeneous memory systems at the Operating System (OS) level. A naive policy is to allocate pages in the stacked DRAM and, once it is full, allocate pages in the off-chip DRAM. A more sophisticated approach is to migrate pages from the off-chip DRAM to the stacked DRAM based on some heuristic, which arises some problems. First, the OS has very limited information on how the pages are accessed, as it only has a single accessed bit per page in the page table and does not know how recently or how many times the pages are accessed. This problem can be solved by extending the page table and the TLBs to measure the page hotness [28]. The second problem is the cost of page migrations, which require to trigger an interrupt, to copy the page to the stacked DRAM (potentially copying a dirty page back to the off-chip memory), to update the page table, and to invalidate the TLB entries for that page. Sim et al. [38] introduce an extra level of address translation from physical addresses into DRAM addresses, which allows the hardware to swap pages between the stacked DRAM and the off-chip memory without OS intervention.

Khaldi et al. [22] propose to automate the allocation of data objects at compile time. In this approach the compiler analyses the code to detect frequently used data and memory access patterns, it

uses this information to calculate a priority value for each data object, and then it generates the appropriate functions calls to allocate the data objects in the stacked DRAM or in the off-chip memory. Other works propose to automatically allocate the most suitable data objects to the stacked DRAM at runtime. Servat et al. [37] present a framework of runtime libraries and tools that generates a profile of the execution, calculates the best distribution of data objects in the memory system, and allocates them accordingly in future executions. RTHMS [32] uses a similar approach to provide programmers with recommendations on data placement.

6.3 Task-Based Programming Models

The ideas proposed in this paper apply to runtime-managed task-based programming models that specify data dependencies between tasks. It has been shown in the literature that these programming models offer great potential for managing complex memory organizations with multiple address spaces, such as heterogeneous systems with accelerators or multi-node clusters. OmpSs [8, 9, 33] extends OpenMP with pragmas to specify the device where tasks can be executed and the inputs and outputs of the tasks that require copies of data. The address spaces of the nodes and the GPUs are managed by the runtime system as software caches, and a centralized directory tracks all the data in all the caches. Asynchronous data transfers are supported by the underlying libraries (GASNet for multi-node clusters and the CUDA runtime for GPUs). StarPU [3] applies a very similar model to support accelerators such as GPUs or the Cell SPUs, but offers a lower-level API to the programmer instead of pragmas. CnC-HC [36] and CnC-CUDA [14] extend CnC with constructs to specify the data accessed by the tasks, and the underlying runtime systems Habanero-C and Habanero-Java perform the data transfers [15]. Sequoia [23] generates code for the data transfers between tasks at compile time, so it does not need a directory to manage the data at runtime. Legion [5] extends Sequoia by providing a mapping interface for the programmer to control the data transfers when tasks start and finish. Similarly, PaRSEC [7] allows the programmer to describe the distribution of data and tasks across the system as a way to control the amount of communication. In Cilk [6] tasks communicate data by means of explicit calls in the application code, and the runtime system uses system-dependent libraries (e.g. Strata active-message library on the CM5) to trigger data transfers when needed. OCR [26] is a generic runtime system that offers a low-level API to higher-level programming models, including abstractions to manage data by means of a globally accessible shared name space of data objects. OCR relies on the higher-level programming models to map application data to the OCR data objects and to perform the data transfers between address space, if needed.

Charm++ has some preliminary support to manage stacked DRAM memories [12]. There are numerous and important differences between our approach and the one proposed by Charm++. A big difference is that Charm++ requires the programmer to explicitly indicate the tasks that have to map its data to the stacked DRAM, so the runtime system does not decide whether the data of a task should be mapped to the stacked DRAM or not. This is important because, if the stacked DRAM is full, the tasks are blocked until some free space is available. In this paper we propose that the runtime system manages the stacked DRAM without any programmer intervention, and it automatically decides to map data to the stacked DRAM or to allow tasks to access the off-chip memory when the stacked DRAM is full or the data is not reused. Another important difference is that Charm++ relies on prefetching data to the stacked DRAM using helper threads. The main problem of prefetching is that a big part of the stacked DRAM has to be devoted to prefetch buffers, limiting the amount of data that can be accessed by the in-flight tasks. This imposes a severe limitation either to the number of tasks that can be executed in parallel or to the granularity of the tasks. In the evaluation, Charm++ only considers two benchmarks running with 64 threads (counting helper and compute threads) and restricts the working set size of the in-flight tasks to 2 to 8 GB. In such scenario, their results show similar trends to the ones we observe with the parallel copies when using 16 and 32 compute threads (Figure 6a). In this paper we use 9 benchmarks, we explore different task granularities and number of threads, and we observe that the best performance is achieved using 64 to 256 compute threads, and that the working set size of the in-flight tasks occupies most of the capacity of the stacked DRAM. In this scenario, adding helper threads is not so effective because of the resource contention in the SMT cores, and we propose a reuse-aware bypass policy that adapts to the characteristics of the applications and always achieves the best performance.

7 CONCLUSIONS

This paper proposes a runtime approach to transparently manage stacked DRAM memories of heterogeneous memory systems without affecting programmability. The proposed techniques leverage the runtime system of task-based programming models to dynamically copy to the stacked DRAM the data that is going to be accessed by the tasks. The proposal includes two optimizations to overcome the cost of copying data between the stacked DRAM and the off-chip memory. The first optimization parallelizes the data copies across idle threads or additional helper threads. In the second optimization the runtime system exploits the data reuse pattern information to avoid copying unworthy data to the stacked DRAM. Results on the Intel Knights Landing processor show that the performance of state-of-the-art approaches heavily depends on the data reuse pattern of the applications. The proposed runtime approach automatically adapts to the characteristics of each application and always provides the best performance, achieving average speedups of 14% and 29% against a stacked DRAM managed as part of memory and as a hardware cache, respectively.

ACKNOWLEDGEMENT

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by the European Union's Horizon 2020 research and innovation programme (grant agreement 779877). M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104.

REFERENCES

[1] OpenMP Application Program Interface. Version 4.0. 2013.

[2] Lluc Alvarez, Miquel Moreto, Marc Casas, Emilio Castillo, Xavier Martorell, Jesus Labarta, Eduard Ayguade, and Mateo Valero. 2015. Runtime-Guided Management of Scratchpad Memories in Multicore Architectures. In *International Conference on Parallel Architectures and Compilation (PACT '15)*. 379–391.

[3] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *International Conference on Parallel and Distributed Computing (Euro-Par '11)*. 187–198.

[4] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'Brien, and Kathryn M. O'Brien. 2004. Nanos Mercurium: a Research Compiler for OpenMP. In *European Workshop on OpenMP (EWOMP '04)*. 103–109.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. 66:1–66:11.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*. 207–216.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *IEEE Computing in Science and Engineering* 15, 6 (2013), 36–45.

[8] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. 2013. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *International Conference on Supercomputing (ICS '13)*. 359–368.

[9] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. 2012. Productive Programming of GPU Clusters with OmpSs. In *International Parallel and Distributed Processing Symposium (IPDPS '12)*. 557–568.

[10] Marc Casas, Miquel Moreto, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2015. Runtime-Aware Architectures. In *International Conference on Parallel and Distributed Computing (Euro-Par '15)*. 16–27.

[11] Emilio Castillo, Miquel Moreto, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kalia Chronaki, Rosa M. Badia, Jose L. Bosque, Ramon Beivide, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2016. CATA: Criticality Aware Task Acceleration for Multicore Processors. In *International Parallel and Distributed Processing Symposium (IPDPS '16)*. 413–422.

[12] Kavitha Chandrasekar, Xiang Ni, and Laxmikant V. Kale. 2017. A Memory Heterogeneity-Aware Runtime System for Bandwidth-Sensitive HPC Applications. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW '17)*. 1293–1300.

[13] Alejandro Duran, Eduard Ayguade, Rosa M Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters* 21, 2 (2011), 173–193.

[14] Max Grossman, Alina Simion Sbirlea, Zoran Budimlic, and Vivek Sarkar. 2010. CnC-CUDA: Declarative Programming for GPUs. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC '10)*. 230–245.

[15] Habanero Multicore Software Project. http://habanero.rice.edu. 2018.

[16] Memkind. http://memkind.github.io/memkind. 2018.

[17] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *International Conference on Parallel Architectures and Compilation (PACT '14)*. 51–60.

[18] JEDEC. 2013. High Bandwidth Memory (HBM) DRAM. JESD235.

[19] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *International Symposium on Microarchitecture (MICRO '14)*. 25–37.

[20] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *International Symposium on Computer Architecture (ISCA '13)*. 404–415.

[21] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. 2010. CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms. In *International Symposium on High-Performance Computer Architecture (HPCA '10)*. 1–12.

[22] Dounia Khaldi and Barbara Chapman. 2016. Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing. In *Workshop on LLVM Compiler Infrastructure in HPC (LLVM-HPC '16)*. 12–20.

[23] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. 2007. Compilation for Explicitly Managed Memory Hierarchies. In *Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. 226–236.

[24] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *International Symposium on Microarchitecture (MICRO '11)*. 454–464.

[25] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericas, and Per Stenstrom. 2016. RADAR: Runtime-Assisted Dead Region Management for Last-level Caches. In *International Symposium on High Performance Computer Architecture (HPCA '16)*. 644–656.

[26] Timothy G. Mattson, Romain Cledat, Vincent Cave, Vivek Sarkar, Zoran Budimlic, Sanjay Chatterjee, Josh Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, Benoit Meister, Brian Nickerson, Nick Pepperling, Bala Seshasayee, Sagnak Tasirlar, Justin Teller, and Nick Vrvilo. 2016. The Open Community Runtime: A Runtime System for Extreme Scale Computing. In *High Performance Extreme Computing Conference (HPEC '16)*. 1–7.

[27] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995), 19–25.

[28] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories. In *International Symposium on High Performance Computer Architecture (HPCA '15)*. 126–136.

[29] Dragomir Milojevic, Sachin Idgunji, Djordje Jevdjic, Emre Ozer, Pejman Lotfi-Kamran, Andreas Panteli, Andreas Prodromou, Chrysostomos Nicopoulos, Damien Hardy, Babak Falsari, and Yiannakis Sazeides. 2012. Thermal Characterization of Cloud Workloads on a Power-efficient Server-on-chip. In *International Conference on Computer Design (ICCD '12)*. 175–182.

[30] Abhisek Pan and Vijay S. Pai. 2015. Runtime-driven Shared Last-level Cache Management for Task-parallel Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. 11:1–11:12.

[31] Thomas Pawlowski. 2011. Hybrid Memory Cube (HMC). In *Hot Chips Symposium (HCS '11)*. 1–24.

[32] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *International Symposium on Memory Management (ISMM '17)*. 82–91.

[33] Judit Planas, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. 2013. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *International Parallel and Distributed Processing Symposium (IPDPS '13)*. 138–149.

[34] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *International Symposium on Microarchitecture (MICRO '12)*. 235–246.

[35] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *International Parallel and Distributed Processing Symposium (IPDPS '17)*. 297–306.

[36] Alina Sbirlea, Yi Zou, Zoran Budimlic, Jason Cong, and Vivek Sarkar. 2012. Mapping a Data-flow Programming Model Onto Heterogeneous Platforms. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES '12)*. 61–70.

[37] Harald Servat, Antonio J. Peña, German Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesus Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *International Conference on Cluster Computing (CLUSTER '17)*. 126–136.

[38] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM As Part of Memory. In *International Symposium on Microarchitecture (MICRO '14)*. 13–24.

[39] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *International Symposium on Microarchitecture (MICRO '12)*. 247–257.

[40] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (2016), 34–46.

[41] Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguade, and Jesus Labarta. 2014. Runtime-Aware Architectures: A First Approach. *Journal on Supercomputing Frontiers and Innovations* 1, 1 (2014), 29–44.