Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems

Antonio J. Peña
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: apenya@anl.gov

Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: balaji@anl.gov

***Abstract*—The increasing number of memory technologies offering different features such as optimized access patterns or capacity/speed ratios lead us to advocate for future HPC compute nodes equipped with heterogeneous memory subsystems. The aim is to alleviate further the ever-increasing gap between computation and memory access speeds, by taking advantage of the benefits these memory technologies provide. Compute nodes equipped with memory technologies such as scratchpad memory, on-chip 3D-stacked memory, or NVRAM-based memory are already a reality. Careful use of the different memory subsystems is mandatory in order to exploit the potential of such supercomputers. While most multiple-memory models concentrate on extending the depth of the memory hierarchy by incorporating more levels of hardware-managed memories, we advocate for compute nodes equipped with heterogeneous software-managed memory subsystems. Although the exact approach to efficiently exploit them is still uncertain, a software ecosystem clearly is required in order to assist in an efficient data distribution. We address this problem at the memory object granularity. In this paper we use an object-differentiated profiling tool we have developed on top of the Valgrind instrumentation framework, in order to assess the most suitable memory subsystem for the different memory objects of two miniapplications from the Mantevo codesign project. Our results considering two different memory configurations as use cases reveal the potential benefits of carefully placing the different memory objects of an application among the different memory subsystems.**

I. Introduction

Computers have been incorporating deep memory hierarchies to alleviate the ever-increasing gap between computing and memory access performance. So far, this strategy has translated into different levels of cache, which are handled automatically by hardware heuristics taking advantage of space and time locality. The requirement for more and faster memory from high-performance computing (HPC) consumers, along with advances in memory technology, is leading to the incorporation of on-chip 3D-stacked memory [1], which will bring relatively large amounts of memory inside the processor die. This trend, if successful, is likely to end up generalizing heterogeneous memory systems, bringing more memory subsystems to compute nodes featuring different characteristics, such as size, performance, energy consumption, resilience, or specialized memory for different access patterns. Examples are scratchpad memory featuring cachelike speeds but small sizes; vector-specialized memory, such as GDDR; low-power memory, bringing an increased energy/speed ratio; ECC-enabled memory, offering fault tolerance with some speed and size overhead; and NVRAM, featuring large memory spaces at reduced speeds and energy consumption. A theoretical example of such a system, equipped with a large variety of memory subsystems, is depicted in Figure 1. Although because of technical and practical constraints we do not necessarily foresee such largely equipped systems, we clearly advocate for systems equipped with several types of memory depending on the cluster computing requirements.

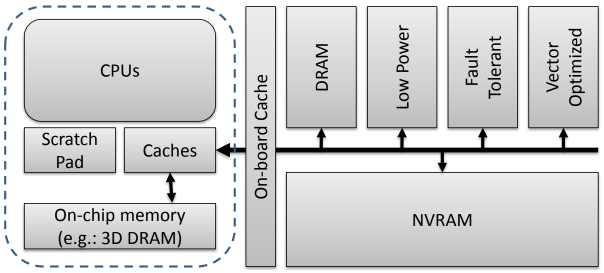
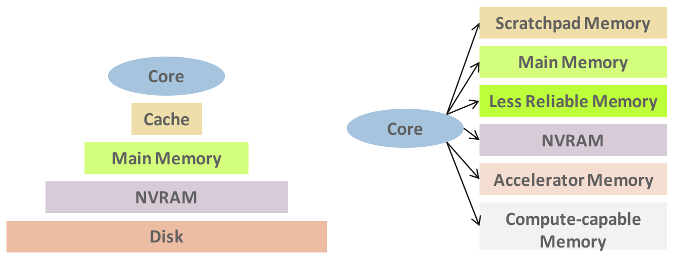


Fig. 1. Simplified view of a heterogeneous memory node sample.



(a) Hierarchical memory view. (b) Explicitly managed memory.

Fig. 2. Hierarchical memory view versus explicitly managed memories.

In order to efficiently exploit such heterogeneity in memory subsystems, these must be brought in as first-class citizens rather than being disposed in a traditional hierarchical view (see sample cases in Figure 2). Otherwise, it will not be possible to exploit the largely different features of the memory subsystems by efficiently distributing the applications' data among them to optimize memory access latencies.

Decisions about how to distribute an application's data among these memory subsystems and what method to leverage are still unclear. The operating system in these upcoming computers could use a heuristic-based model, on-the-fly analysis (possibly hardware-assisted), historic data, or user hints

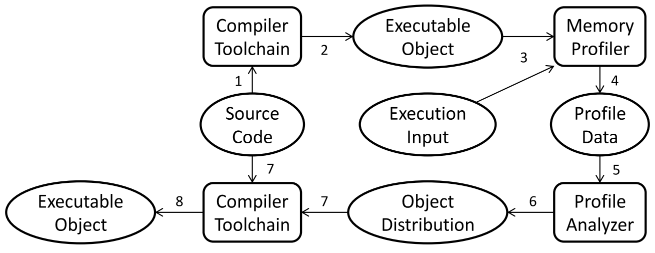


Fig. 3. Abstraction of the proposed compiling flow. Rectangular objects represent software agents, whereas elliptic objects represent the input/output of these agents. The different stages are numbered.

to decide what data to place in each memory subsystem. A good understanding of the way that data are accessed will allow one to avoid relying on heuristics, hence attaining an optimal exploitation of the available memory resources. So far, however, no suitable tools exist to assist with this decision. An ecosystem aware of explicitly addressable multiple memory subsystems offering different features such as profilers, libraries, and runtime systems will be required in order to make the different memory subsystems first-class citizens, thus enabling an efficient exploitation of the benefits these provide.

Current profiling solutions, such as Callgrind [2], Gprof [3], perf [4], and OProfile [5], are code oriented: they focus on those parts of the code featuring the highest levels of a given metric (often execution time, cache misses, or CPU stalled cycles). Although these help developers identify bottlenecks in their software and help maximize the exploitation of the underlying hardware, they do not provide data-oriented profiling information. In this paper we use emulator-based data-oriented profiling to analyze actual program executions in an emulated system featuring a conventional memory architecture equipped with a DRAM-based main memory subsystem only. This setup permits us to associate the occurred (simulated) cache misses with the different memory objects of the executed application. We use this data to estimate the CPU stall cycles incurred by the different memory objects, in order to decide their optimal placement at object granularity in a proposed heterogeneous memory system. The profiling information obtained from our tool may be used in two ways:

1) It can be used by programmers to provide hints to compilers or runtime systems regarding memory object distribution or to explicitly select the location of the different memory objects. This approach requires either a programming model enabled for heterogeneous memory or explicit management from the programmer.
2) The profiling data can be fed back to a second compiling stage, where the compiler automatically distributes the memory objects accordingly, in what can be considered a feedback-directed optimization technique [6]. This approach requires a compiler toolchain capable of explicitly managing data allocation and movement among heterogeneous memory subsystems. This proposal, depicted in Figure 3, is independent of the programming model/language used, thus addressing one of the main concerns regarding heterogeneous memory management in HPC.

Note that although the profiling-based approach assumes that all the considered space in the different memory subsystems is available for the application, this reflects real-case scenarios in most HPC environments where dedicated nodes are leveraged in order to avoid competition for computational resources, and hence the amount of memory available for user-space applications is known beforehand.

To accomplish our goal, we have developed a profiling tool based on the generic Valgrind instrumentation framework [7] capable of differentiating the cache misses per memory object [8], plus an algorithm to optimally distribute the different objects among the proposed memory subsystems based on the 0/1 knapsack problem [9]. We use this tool to analyze a pair of miniapplications on two proposed heterogeneous memory configurations. Our experiments reveal the usability and potential benefits of employing our methodology to efficiently leverage heterogeneous memory systems. The main contributions of this paper are as follows:

1) We propose a profiling-based method to determine the optimal distribution of application data on heterogeneous memory systems at memory object granularity.
2) We provide experimentation results based on miniapplications for a pair of realistic heterogeneous memory configuration proposals.
3) We demonstrate the feasibility of using our methodology to assess the optimal distribution of application data at object granularity on heterogeneous memory systems.

To the best of our knowledge, this is the first work discussing a method based on data-oriented profiling to assess the optimal distribution of memory objects across multiple memory subsystems in heterogeneous-memory compute nodes. We also consider this to be the first work applying data distribution among different heterogeneous memory subsystems to code representatives of widely used HPC applications. Although multiple open questions remain, such as the best programming model to adopt or the tradeoff between the potentially high performance of profiling-based memory placement and the convenience of operating system placement based on heuristics, this paper constitutes one of the first steps toward making the heterogeneous memory system as we foresee it a reality.

The rest of this paper is structured as follows. Section II provides background information about the different memory technologies covered in this paper. Section III describes the methodology we propose to address the optimal distribution of application data on heterogeneous memory systems at object-level granularity. Section IV introduces our modeled systems and the test case applications we use. Section V presents our experimental results. Section VI reviews previous related work. Section VII provides concluding remarks and suggestions for future work.

II. Background

In this section we review the different memory technologies that we consider for our target system proposals, namely cache, scratchpad, 3D-stacked, conventional DRAM, and NVRAM.

A. Cache

Usually arranged in multiple levels of increasing size and latency, CPU cache memories are small hardware-managed, low-latency memories sitting close to the processors. These are employed automatically by the hardware to keep track of the most recently used data in order to have it quickly available if requested again by the CPU.

Usually, CPU stall cycles caused by accesses to cache memories are nonexistent for first-level caches (L1), are around 10 CPU cycles on accesses to the second level of cache (L2), and keep increasing with the distance to the processor [10]. Since our cache simulator can simulate only two levels of cache, we consider a relatively high number of CPU stall cycles of 20 for second-level cache hits, in order to compensate for the absence of a commonly employed third cache level. On the other hand, L1 sizes on the order of kilobytes and upper-level sizes on the order of megabytes are commonly found in today's systems.

B. Scratchpad

Scratchpad memories are a type of cache that is explicitly managed by software. These are commonly used in embedded processors and GPUs but are uncommon in current compute nodes. Providing the same benefits as cache memory, they offer the advantage of a high level of control, which prevents performance issues caused by the heuristic-based automatic management policies such as competition from different objects leading to undesirable early cache evictions. However, the inconvenience of the explicit management at such small granularity has prevented their adoption by general-purpose processors so far.

In our heterogeneous memory proposal we consider a scratchpad memory region placed along with the L2 cache featuring the same size and latency as the latter.

C. On-Chip 3D-Stacked Memory

The recently introduced on-chip 3D-stacked memories, like traditional main memory systems, are based on the widely known DRAM (dynamic random access memory) technology. What differentiates them is that they are physically stacked in multiple layers within the microprocessor die, hence featuring lower latencies and higher bandwidths than external DRAM memories. Currently energy dissipation is one of the limiting factors of this technology.

Today, 8 GB to 16 GB of 3D-stacked memory per chip are usually considered reasonable [1], [11], [12], along with 30% reduction of access latency from that of their off-chip counterpart [11], [12].

D. Main DRAM

DRAM is currently the technology on which the main memory of computers is based. What makes it *dynamic* is the fact that it needs to be refreshed (read and written) periodically, thus introducing a delay in accesses during the refresh process and causing a nonnegligible energy consumption. The most-used class of DRAM is DDR SDRAM (double data rate synchronous DRAM).

Usually, 32 GB, 64 GB, and even larger amounts of DRAM memory are found in current compute nodes. On the other hand, average latencies of around 200 CPU cycles are commonly used as rough estimations of the data access times to this kind of memory.

E. NVRAM

NVRAM (non-volatile RAM) is a memory technology that does not require energy to maintain the data. On the negative side, the number of write-erase cycles is limited, and write speeds are considerably lower than those attained by reads. In addition, current NVRAM-based memories may feature limits on the access patterns, since these memories are not byte addressable at low level, enforcing block accesses. Nevertheless, this limitation can be overcome employing high-level libraries such as NVMalloc [13], which offer a byte-addressable view of this memory space by internally handling the block access operations.[1] The NVRAM technology has recently been widely adopted for use in I/O-based storage. Although its usage as main memory is uncommon, we adopt it as part of our heterogeneous memory proposals—assuming a byte-addressable approach—as a means of incorporating a low-power high-capacity slow memory subsystem.

The fastest reported latencies to date start at 10μs [14]. Accordingly, we round the average data latency of our NVRAM memory subsystem down to 100 times that of our DRAM estimations (that is, 20,000 CPU cycles).

III. Methodology

Our methodology consists of profiling the execution of a given application in order to determine the last-level cache misses that occurred during its execution, grouped by the different memory objects[2] that the application leverages. Next, we assess their optimal distribution among the different memory subsystems of a proposed heterogeneous memory system, in order to minimize the processor stall cycles those accesses will cause. We plan to publicly release our developed tool shortly, after code cleanup and integration efforts.

A. Profiling

To be able to associate the last-level cache misses of an application's execution with its memory objects, we perform executions in an emulated system. Specifically, we used the Valgrind instrumentation framework [7], which provides the execution platform, and its Callgrind tool [2], a call-graph cache profiler, which provides the cache simulation features. These include branch prediction and hardware prefetching, based on the features of a third Valgrind tool: Cachegrind [15].

For our profiling analysis, we incorporated per-object differentiation capabilities into Valgrind and adapted Callgrind to make use of our object-differentiation engine and provide last-level cache miss data per memory object [8].

The way we associate last-level cache misses with memory objects is determined by the origin of the object. The information about statically allocated objects, including those residing on a stack frame, is located in the debug information of the code, embedded at compile time by the compiler (for example, by specifying the `-g` parameter in most current compilers). The data include information that permits determining the address range on where objects are placed, their scope (whether they

[1]Accesses not featuring space and time locality may suffer from relatively high overheads.

[2]By "memory object" we refer to every data entity of an application, that is, both statically and dynamically allocated variables and buffers.

Listing 1. Main cache miss recording algorithm.

```
found = false
if trace_dynamic:
  found = record_mem_access(addr, size, meta);
if trace_static and not found:
  found = record_var_access(addr, size, meta);
```

Listing 2. Pseudocode for the profiling procedure.

```
simulate(max_iter):  /* max_iter = 2 */
  CALLGRIND_START_INSTRUMENTATION
  for i in 1..max_iter:
    /* Simulation original code */
    if i = 1:
      CALLGRIND_ZERO_STATS
  CALLGRIND_STOP_INSTRUMENTATION
```

are defined given an instruction pointer), their name, and the place on the source code where they are declared. We extended Valgrind's core debug information module with functionality to locate the object containing a memory address and to store the associated last-level cache misses within its data.

On the other hand, the approach we follow for dynamically allocated memory is to deploy wrappers to intercept the system memory management function calls. Upon an intercepted allocation call, we store the corresponding address range and stack trace. This information is arranged in a sorted structure in order to provide efficient (logarithmic) searches. We use Valgrind's specific capabilities to intercept these calls, and we provide a versatile module that different tools may incorporate with minimal changes. For instance, in [8] we demonstrate its usability from the Lackey [16] tool to obtain object-differentiated memory access traces along executions. Further details on our extensions to Valgrind and its tools with object-differentiated analysis capabilities, including design and implementation details, are presented in [8].

During the execution of a profiled application, every data address accessed by the application causing a last-level cache miss is checked against both the existing dynamically and statically allocated objects, as depicted in pseudocode in Listing 1. Although in spite of our optimizations this approach poses a considerable execution overhead in top of that of the Callgrind tool, which is already far from negligible, we use Callgrind's capabilities to start code instrumentation and reset counters upon the client's request, in order to limit the profiling overhead to our region of interest and hence be able to profile production-sized executions. We run two iterations/ timesteps of our test-case applications, profiling the second of them after using the first to warm up the caches and avoid accounting for cold misses. Specifically, we start the instrumentation just before the first timestep—used to warm up the cache—clearing the profiling counters at the end of it and finalizing the instrumentation at the end of the second iteration. This process is illustrated in the pseudocode in Listing 2.

The output of our profiling stage is the per-object cache misses that occurred during the profiled portion of the execution. Although we collect data relating cache misses with the execution timeline, this information is not used in our subsequent analysis; it is intended for future work targeting memory migrations.

Listing 3. Object distribution algorithm.

```
memories.sort(key=latency, order=ascending)
for memory in memories:
  packed = knapsack(objects, memory.size)
  objects = objects - packed
```

B. Analysis

We next analyze the obtained profiling data, in order to distribute the different memory objects among the available memory subsystems and thus to minimize the number of stalls caused by memory accesses. This turns to be a multiple knapsack problem, where the knapsacks are the different memory subsystems and the knapsack capacity is represented by the memory size; the items to pack are the memory objects, their weight is their size, and their value is the number of load cache misses these feature. However, our case differs from a textbook multiple knapsack problem in that the different knapsacks modify the value of their items, since these multiply the cache misses by a different factor (the average read latency, in CPU cycles, for a data word to reach the CPU) in order to obtain our actual metric: the CPU stall cycles caused by each object given a particular memory subsystem.

To tackle this situation, we follow a greedy approach by solving separate 0/1 knapsack problems [9]. We target the different memories in ascending order of average access cycles, as shown in the pseudocode in Listing 3, thus prioritizing the placement of the most "valuable" objects in the faster memory subsystems. Although this algorithm is not guaranteed to attain an optimal global solution, because we consider the different knapsacks separately, we do not foresee a large divergence from optimal distributions in practice. Moreover, the algorithm relaxes the problem, largely removing computational complexity. Since the 0/1 knapsack problem is *weakly NP-hard* with time complexity $O(nW)$, where n is the number of items and W is the knapsack capacity, if m is the number of memory subsystems, the complexity of our algorithm is $O(mnW)$. Note that we express W by the number of 4 KB pages, mimicking most real-use cases.

The output of our analysis stage is the proposed distribution of the different memory objects of the analyzed application among the different memory subsystems being considered, along with an estimation of the stall cycles that would be caused by that configuration. On the other hand, we limit our study to the memory objects allocated within the user binary object, discarding those used by external libraries or system calls.

C. Assumptions and Current Known Limitations

Derived from the fact that our method relies on executions on emulated hardware, our timeline is the number of executed instructions instead of execution time. On the other hand, Callgrind limits the cache simulation to a two-level shared cache system: a first level of separate instruction and data caches and a second level of unified cache. We also consider write misses to cause no stall cycles, assuming a *buffered write-through* cache policy with unlimited buffer size (in practice, stall cycles caused by read misses are much larger than those caused by write misses, and hence this relaxation is not expected to notably impact the accuracy of our estimations).

TABLE I. CACHE CONFIGURATION IN OUR EXPERIMENTS

Description	Total Size	Associativity	Line Size
L1 Instruction	32 KB	8	64 B
L1 Data	32 KB	8	64 B
L2 Unified	8 MB	16	64 B

TABLE II. MEMORY CONFIGURATION IN OUR EXPERIMENTS

Memory		Scenario		
Description	Latency	Baseline	Scenario 1	Scenario 2
L1	0 c	32 KB + 32 KB		
L2	20 c	8 MB		
SP	20 c	0 B	8 MB	8 MB
3D	135 c	0 B	8 GB	1 GB
Main	200 c	32 GB	32 GB	4 GB
NVRAM	20,000 c	0 B	0 B	32 GB

Another limitation of our methodology of analysis lies in the fact that we are not simulating the different memory subsystems we consider; instead, we use average latency estimations. Although these reduce the accuracy of our analysis, we choose this approach because of factors such as the unavailability of detailed low-level specifications from commercial memory technologies and the large amount of time that those simulations would involve.

Currently, we consider neither memory migrations nor reuse of freed space (other than by the same memory object[3]): our algorithm pursues a static placement of the different objects assuming that these will be allocated during the whole lifetime of the application. Memory migration would be interesting in those cases where the access patterns of the objects considerably change along the lifetime of the execution. We note that our study is limited to profiling a core part of an application in order to determine the best object distribution during that execution part, and hence this limitation is not expected to affect the validity of our analysis. Studying memory subsystem migration is left for future work.

IV. TEST CASES

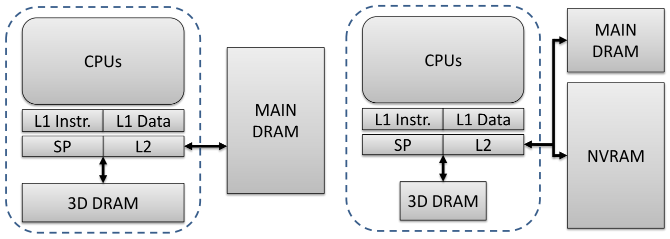
In this section we first introduce our target system. We then present the software we use as test cases for our study.

A. System Setup

For our experiments we consider an 8-core processor featuring a set-associative cache with the configuration specified in Table I.

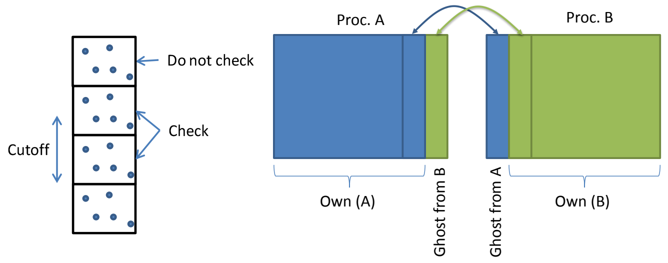
Our baseline system is equipped with a traditional DRAM-based main memory space. We target two different heterogeneous memory configuration proposals, as depicted in Figure 4. Scenario 1 (Figure 4a) features a scratchpad (SP) memory region at level 2 cache, an on-chip 3D-stacked DRAM subsystem, and the corresponding main memory system. The common components (caches and main memory) are sized as in the baseline compute node. Scenario 2 (Figure 4b) reflects a more power-friendly system, featuring 3D-DRAM and main memories of reduced size, and incorporating an NVRAM-based memory subsystem, while the rest of the components are left as in scenario 1. The sizes of the different configurations are shown in Table II.

[3]We consider different memory objects to be the same as long as these are allocated or declared from the same execution stack [8].



(a) Scenario 1. (b) Scenario 2.

Fig. 4. Target system memory architectures.



(a) Cutoff distance. (b) Stencil communications.

Fig. 5. Some MiniMD features.

In our estimations, we consider that our processor leverages one instruction per cycle and that no stall cycles are caused by hazards. The average latencies that we use for the different memory subsystems, expressed in CPU cycles, are specified in Table II.

B. Applications

In our study we used a pair of miniapplications from the Mantevo codesign project [17]: MiniMD and HPCCG. These are intended as benchmarks for assessing the performance of certain production applications. Hence, they are representative of the behavior of more complex applications.

1) MiniMD: MiniMD is a reduced version of the LAMMPS molecular dynamics simulator [18], [19], [20]. It is defined as "a simple proxy for the force computations in a typical molecular dynamics application." It simulates the movement of atoms in a box area following Newton's laws. Two methods are available in MiniMD: Lennard-Jones (`LJ`) and the embedded atom model (`EAM`). As with LAMMPS, MiniMD simulations are compute bound. MiniMD implements all the major optimizations of its big brother, such as the concept of *cutoff distance* to avoid computations of interactions with distant atoms and *atom binning* to reduce the computational load (see Figure 5a). Other characteristics of MiniMD include spatial decomposition by means of MPI processes and stencil communications among them in order to exchange ghost atoms, as illustrated in Figure 5b.

Users can set a variety of simulation parameters such as the simulated size, atom density, temperature, timestep size, number of timesteps, cutoff distance, or reneighboring frequency.

The simulation is run iteratively computing the forces, positions, and velocities of atoms on successive timesteps.

Every n number of timesteps, a reneighboring is performed, in order to recompute each atom's neighbors. Since the atoms' mobility is limited, the reneighboring stage is not needed in every iteration. The two most time-consuming tasks are the compute of the forces, performed every timestep, and the reneighboring (usually 4 to 5 times longer than the compute forces stage), in a user-specified frequency.

An interesting characteristic of this miniapplication for our study is that it leverages multiple relatively large memory objects gathering largely different number of cache misses. Thus, it enables us to explore the performance of different object distributions among the available memory subsystems and hence demonstrate the potential benefits of a careful object distribution on heterogeneous memory nodes.

We use the reference implementation of miniMD (version 1.2). Our simulations leverage LJ interactions among $2.9 \cdot 10^6$ atoms in a multithreaded process running 8 threads, using 26 GB of memory. Our profiling reveals that 23% of the estimated execution cycles in our baseline hardware setup are produced by cache misses.

2) HPCCG: HPCCG is defined as "a simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors." This miniapplication focuses on mimicking the performance behavior of applications executing a conjugate gradient solver as their main computational kernel to address an unstructured grid problem, reproducing its finite-element generation, assembly, and solution stages. The physical domain consists of a three-dimensional box decomposed by using recursive coordinate bisection on hexahedric elements. This is a linear problem resulting in a sparse symmetric matrix. Interprocess communications are minimal, involved only in the exchange of nearest-neighbor boundary information and scalar computations in the conjugate gradient algorithm. The code is designed to be highly scalable (in a weak sense).

The access pattern of this miniapplication has been demonstrated to be highly memory demanding and sensitive to different memory architectures [17]. Hence we expect a high sensitivity to memory placement on this miniapplication.

We use HPCCG reference version 1.0 in our experiments to run a $400 \times 400 \times 400$ node problem on an 8-threaded process, requiring 24 GB of memory. According to our profiling data, this leads to 48% of the estimated execution cycles in our baseline architecture being produced by cache misses.

V. Experimental Results

In this section we discuss our experimental results. We first profile the execution of our test case miniapplications as described in Section III-A. From the profiling data, we compute an optimized distribution of the memory objects as discussed in Section III-B. To compare our optimized proposal with an unoptimized case, while still presenting a realistic distribution proposal for comparison, we compute a worse-case scenario as follows:

- We invert the "value" of those objects presenting nonzero cache misses, so that those featuring fewer such misses are preferably allocated in the fastest memory subsystems.
- We discard those memory objects not presenting cache misses during our profiled portion of the execution before feeding the profiling data to our knapsack-based memory distribution algorithm. Doing so makes our unoptimized case not be the worst possible case, as it would be if we were to place in the fastest memories those objects presenting zero cache misses.

TABLE III. Object distribution for MiniMD – Scenario 1

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	21	53%	-4 M	0.0%
3D	9	65%	-743 M	-7.6%
DRAM	0	0%		
Total			-747 M	-7.6%

Our results include an estimation of the execution cycles (for the profiled portion of interest) from the modeling data we introduced in Section IV-A for both the optimized and unoptimized distribution proposals in the three hardware configurations we consider: baseline, scenario 1, and scenario 2.

In the remainder of this section we first introduce the MiniMD experiments, followed by those obtained for the HPCCG miniapplication. We conclude this section with a discussion of the obtained results.

A. MiniMD

Our profiling for MiniMD revealed that over 99% of the 4,716 memory objects this miniapplication leverages caused no last-level cache read misses in our simulated cache during the execution of our profiled region of interest. These occupy almost 21 GB of memory. These objects are not considered in our subsequent analysis, since their distribution in the different memory subsystems does not affect our profiled region of interest.

1) Scenario 1: Our results for MiniMD in scenario 1 are summarized in Table III. In this case 21 objects are smaller than the SP size, while 9 are larger than that size and smaller than the 3D memory size. In addition all 21 small objects fit together in the SP memory, and all 9 large objects fit in the 3D memory. Hence, the distribution is trivial, and we do not distinguish optimized and unoptimized distributions. Our results show an improvement of 7.6% in execution time with respect to the baseline memory architecture, contributed mostly by the on-chip 3D memory performance benefits.

2) Scenario 2: In scenario 2 the memory size restrictions make a choice available at the DRAM/NVRAM level, where four of our objects of interest are small enough to be allocated on the DRAM, although there is sufficient space for only three of them. Our optimized distribution of the objects, summarized in Table IV, attains a limited loss of 13.7% with respect to the baseline configuration, led by the limited size of memories featuring low latencies. On the other hand, the unoptimized object distribution (see Table V) shows a slowdown of over 1,000% with respect to the baseline case, caused by the placement of the memory object featuring the largest number of cache misses in the NVRAM memory, thus illustrating the relevance of an optimized object distribution.

TABLE IV. OPTIMIZED DISTRIBUTION FOR MINIMD – SCENARIO 2

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	21	53%	-4 M	0.0%
3D	5	21%	-18 M	-0.2%
DRAM	3	93%		
NVRAM	1	4%	+1 G	+13.9%
Total			+1 G	+13.7%

TABLE V. UNOPTIMIZED DISTRIBUTION FOR MINIMD – SCENARIO 2

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	21	53%	-4 M	0.0%
3D	5	21%	-18 M	-0.2%
DRAM	3	94%		
NVRAM	1	4%	+130 G	+1,330.5%
Total			+130 G	+1,330.3%

TABLE VIII. OPTIMIZED DISTRIBUTION FOR HPCCG – SCENARIO 2

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	9	0%	-2 K	0.0%
3D	2	95%	-224 M	-3.8%
DRAM	5	54%		
NVRAM	2	60%	+193 G	+3,236.7%
Total			+193 G	+3,232.9%

TABLE IX. UNOPTIMIZED DISTRIBUTION FOR HPCCG – SCENARIO 2

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	9	0%	-2 K	0.0%
3D	2	95%	-6 M	-0.1%
DRAM	4	83%		
NVRAM	2	60%	+193 G	+3,236.7%
Total			+193 G	+3,236.6%

B. HPCCG

In the case of HPCCG, 742 of the 760 memory objects this miniapplication leverages do not incur in any last-level read cache miss during our profiled portion of the execution, which occupy less than 2 GB out of the 24 GB of memory that our execution requires. On the other hand, only 9 small memory objects fit into the SP memory, saving a negligible amount of CPU cycles. The remainder 9 objects, however, can be distributed in different ways, as discussed next for the different scenarios.

1) Scenario 1: In scenario 1 only one memory object is larger than the 3D memory size and hence forced to be placed in the DRAM memory space. The remaining eight objects can be allocated in either the DRAM or the 3D memory, although the space constraints prevent placing all of them in the 3D memory. Our algorithm for an optimized distribution obtains an 8.3% improvement in iteration execution time, as shown in Table VI, whereas the unoptimized distribution reveals only a 4.9% improvement (see Table VII).

2) Scenario 2: The restrictions on the memory sizes in scenario 2 mean that two of the memory objects of our execution fit only in NVRAM memory, leaving seven objects to be distributed between the 3D and DRAM memory subsystems. The result is a 3.8% performance improvement in the number of accesses to the objects allocated in the 3D memory with respect to our baseline with our optimized distribution, as shown in Table VIII, and merely 0.1% when the unoptimized placement is leveraged (see Table IX). Nevertheless, the large penalty imposed by the objects allocated with no choice into the NVRAM makes the overall benefit from the optimized placement negligible in this case.

TABLE VI. OPTIMIZED DISTRIBUTION FOR HPCCG – SCENARIO 1

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	9	0%	-2 K	0.0%
3D	4	98%	-497 M	-8.3%
DRAM	5	45%		
Total			-497 M	-8.3%

TABLE VII. UNOPTIMIZED DISTRIBUTION FOR HPCCG - SCENARIO 1

Memory	Objects	Occupancy	Cycles Difference	Execution
SP	9	0%	-2 K	0.0%
3D	7	39%	-294 M	-4.9%
DRAM	2	60%		
Total			-294 M	-4.9%

C. Discussion

In all cases the SP memory presented low occupancy rates along with low contributions to the overall performance difference with the baseline case. The reason is that most of the small objects of our studied pieces of code do not present last-level cache misses. This result is expected in highly tuned executions involving large memory objects, which are more likely to present a large number of last-level cache misses, while cache competition effects among memory objects are minimized.

On the other hand, our results, summarized in Table X, reveal nonnegligible performance benefits of an optimized memory object distribution on two of our four test cases, while the remaining two showed no noticeable difference. Large performance benefits—more than 10 times that of the unoptimized case—were obtained for MiniMD in scenario 2 by avoiding placing a memory object causing a large number of last-level cache read misses in the performance-expensive NVRAM memory subsystem. The other favorable case, HPCCG in scenario 1, showed almost 4% improvement for the optimized distribution over its unoptimized counterpart thanks to the allocation of those memory objects presenting a larger number of cache misses closer to the CPUs.

In summary, although different constraints—such as memory latency differences, memory sizes, and object sizes—determine the gains of employing an optimized memory object distribution among the different memory subsystems of a heterogeneous memory system with respect to an unoptimized placement, the potential gains are sufficiently large to merit consideration.

VI. RELATED WORK

In this section we review previous work related to our research. We first discuss data-oriented profiling. Next we present prior research related to multiple-memory systems. We also discuss prior work related to data placement in NUMA systems. To the best of our knowledge, our work is the first

TABLE X. OVERALL PERFORMANCE IMPROVEMENT WITH RESPECT TO UNOPTIMIZED DISTRIBUTION

Hardware	Test Case	
	MiniMD	HPCCG
Scenario 1	0.0%	3.7%
Scenario 2	1,158.0%	0.1%

proposing a method (along with a tool) to distribute data among multiple heterogeneous memory subsystems based on data-oriented profiling and to demonstrate its potential benefits employing code representative of applications from the HPC arena.

A. Data-Oriented Profiling

Data/object-oriented profiling has been of interest as a means of helping developers optimize their applications or optimize object arrangement for improved cache behavior within a traditional memory hierarchy.

MemSpy [21], [22] was described as a “prototype tool” for profiling applications based on the Tango [23] system simulator. Apart from interesting code-oriented performance analysis assistance, it provided object-differentiated cache misses similar to the way our profiling tool does. This was aimed at offering a complementary profiling view to explore application performance. To the best of our knowledge, however, both the Tango and MemSpy projects were discontinued long ago, and MemSpy was not made publicly available.

Gleipnir [24], [25], [26] is a Valgrind tool providing raw object-differentiated data access tracing. It is combined with the Gl_cSim cache simulator (based on DineroIV [27]) to study the cache behavior of the different data structures leveraged by applications. This differs from our work in that we focus on analyzing those data accesses overpassing the last-level cache, since we are not interested in cache performance issues. In addition, we directly analyze memory accesses within the cache simulator, hence avoiding unnecessarily generating large trace files and the time overhead that process implies.

Itzkowitz et al. [28] have studied hardware-assisted data-oriented profiling for the UltraSPARC-III family of processors. Their work extended the Sun ONE Studio compilers and performance tools to provide object-differentiated performance data based on hardware counters. While their technique provides low-overhead profiling, it heavily relies on the UltraSPARC-III hardware capabilities to attain a reasonable accuracy.

Several studies have also focused on object-based profiling techniques to leverage object placement optimizations in traditional memory hierarchies with the objective of improving cache behavior [29], [30], [31] or differentiating per-object access patterns to attain a reduction in the profile size [32]. We also find research in high-level programming languages such as exploring object-alignment strategies based on code pattern analysis for Java [33] or using hardware-based statistics from the garbage collector for the same purpose in the Jikes RVM [34].

B. Heterogeneous/Hybrid Memory Systems

Systems equipped with heterogeneous memories have been studied in the past from different approaches. In this section we review several relevant contributions in this field.

The architectural and hardware challenges of leveraging heterogeneous memory systems were studied in [35]. Exploring the use of secondary memory subsystems for different purposes has also been explored in the past [36], [37].

Phadke ad Narayanasamy [38] proposed a system equipped with three different DRAM-based memories optimized for latency, bandwidth, and power, respectively. Similar to our work, the system involves an offline profiling methodology using the last-level cache misses as the metric in order to determine the most appropriate memory subsystem for each *application*. However, application granularity does not exploit the fact that different memory objects may feature largely diverse access patterns.

[39] proposes a similar profiling-based mechanism targeting embedded systems. The fact that these architectures lack hardware-managed (cache-like) memories, makes them follow a different criteria. [40] approached the same problem for cache-equipped embedded systems, but used heuristics to estimate the probability of conflicts in the data cache, which they used as their metric to partition data between on-chip and off-chip memories.

C. Data Distribution in NUMA Systems

A relatively large amount of research has been done on data distribution in NUMA systems. However, the constraints differ from those we face. With NUMA systems, the problem to solve is to distribute the application’s data so that it is as close as possible to the processors that use it the most. In this case, a given memory object is likely to be accessed by processors located in different NUMA nodes, since multithreaded applications commonly leverage algorithms in which the different threads work collaboratively in different parts of large memory objects. For this reason, the NUMA data distribution research focuses on distribution at page-level granularity. Examples of this research include profiling-based distribution such as [41], [42]. Conversely, in our proposal we consider architectures with uniform access to the different memory subsystems. With the uniform access from the different processes to the data belonging to a given memory object, we follow the more natural approach of distributing data with memory object granularity. In future work we will explore the possibility of splitting objects among different memory subsystems if these present different access patterns on different parts of them.

VII. Conclusions and Future Work

In this work we have proposed a methodology to assess the optimal distribution of application data among the different memory subsystems of upcoming compute nodes equipped with heterogeneous memory systems at memory object granularity. We describe a tool that, based on data-oriented profiling, provides an optimized distribution of the objects leveraged by the profiled application. Our results, from two miniapplications as test cases targeting two different configurations of heterogeneous memory systems, reveal the importance of carefully distributing the different memory objects, justifying our proposed analysis methodology and tool.

We plan to extend the capabilities of our tool to analyze the feasibility of object migration between memory subsystems.

Acknowledgments

This material is based upon support by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

[1] S. Anthony, “Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing,” http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing, Nov. 2013.

[2] J. Weidendorfer, M. Kowarschik, and C. Trinitis, “A tool suite for simulation based analysis of memory access behavior,” in *Computational Science-ICCS 2004*. Springer, 2004, pp. 440–447.

[3] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler,” *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[4] A. C. de Melo, “The new Linux ‘perf’ tools,” in *Linux Kongress*, 2010.

[5] W. E. Cohen, “Tuning programs with OProfile,” *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[6] M. D. Smith, “Overcoming the challenges to feedback-directed optimization,” vol. 35, no. 7. ACM, 2000, pp. 1–11.

[7] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[8] A. J. Peña and P. Balaji, “A framework for tracking memory accesses in scientific applications,” in *43nd International Conference on Parallel Processing Workshops (ICPP Workshops)*, Minneapolis, MN, 2014.

[9] G. Mathews, “On the partition of numbers,” *Proceedings of the London Mathematical Society*, vol. 1, no. 1, pp. 486–490, 1896.

[10] D. Levinthal, *Performance Analysis Guide for Intel® Core^{TM} i7 Processor and Intel® Xeon^{TM} 5500 processors*, Intel Corporation, 2009, http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[11] G. H. Loh, “3D-stacked memory architectures for multi-core processors,” in *35th International Symposium on Computer Architecture (ISCA)*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 453–464.

[12] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. Lee, “An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth,” in *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Jan. 2010.

[13] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, “NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines,” in *26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 957–968.

[14] Marvell, “Dragonfly NVRAM,” http://www.marvell.com/storage/dragonfly/nvram, 2014.

[15] N. Nethercote, “Dynamic binary analysis and instrumentation,” Ph.D. dissertation, University of Cambridge, 2004.

[16] Valgrind^{TM} Developers, “Lackey: An example tool,” http://valgrind.org/docs/manual/lk-manual.html, 2013.

[17] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, “Improving performance via mini-applications,” Sandia National Laboratories, Tech. Rep., 2009, http://www.sandia.gov/~maherou/docs/MantevoOverview.pdf.

[18] S. Plimpton, “Fast parallel algorithms for short-range molecular dynamics,” *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[19] S. Plimpton, R. Pollock, and M. Stevens, “Particle-mesh Ewald and rRESPA for parallel molecular dynamics simulations,” in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[20] Sandia National Laboratories, “LAMMPS molecular dynamics simulator,” http://lammps.sandia.gov, 2014.

[21] M. Martonosi, A. Gupta, and T. Anderson, “MemSpy: Analyzing memory system bottlenecks in programs,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, pp. 1–12, 1992.

[22] M. Martonosi, A. Gupta, and T. E. Anderson, “Tuning memory performance of sequential and parallel programs,” *Computer*, vol. 28, no. 4, pp. 32–40, 1995.

[23] H. Davis, S. R. Goldschmidt, and J. L. Hennessy, “Tango: a multiprocessor simulation and tracing system,” in *Proceedings of the International Conference on Parallel Processing*, Aug. 1991, pp. 99–107.

[24] T. Janjusic and K. Kavi, “Gleipnir: A memory profiling and tracing tool,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 4, pp. 8–12, 2013.

[25] T. Janjusic, K. Kavi, and B. Potter, “Gleipnir: A memory analysis tool,” in *International Conference on Computational Science (ICCS)*, 2011.

[26] T. Janjusic, K. M. Kavi, and C. Kartsaklis, “Trace driven data structure transformations,” in *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*. IEEE, 2012, pp. 456–464.

[27] J. Edler and M. D. Hill, “Dinero IV trace-driven uniprocessor cache simulator,” http://pages.cs.wisc.edu/ markhill/DineroIV, 1998.

[28] M. Itzkowitz, B. J. Wylie, C. Aoki, and N. Kosche, “Memory profiling using hardware counters,” in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 1–13.

[29] S. Rubin, R. Bodík, and T. Chilimbi, “An efficient profile-analysis framework for data-layout optimizations,” in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 140–153.

[30] T. M. Chilimbi, “Efficient representations and abstractions for quantifying and exploiting data reference locality,” in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 191–202.

[31] B. Calder, C. Krintz, S. John, and T. Austin, “Cache-conscious data placement,” in *ACM SIGPLAN Notices*, vol. 33, no. 11. ACM, 1998, pp. 139–149.

[32] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August, “Exposing memory access regularities using object-relative memory profiling,” in *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 315–323.

[33] H. Inoue and T. Nakatani, “Identifying the sources of cache misses in Java programs without relying on hardware counters,” in *ACM SIGPLAN Notices*, vol. 47, no. 11. ACM, 2012, pp. 133–142.

[34] F. T. Schneider, M. Payer, and T. R. Gross, “Online optimizations driven by hardware performance monitoring,” in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 373–382.

[35] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao, “Architectural design for next generation heterogeneous memory systems,” in *IEEE International Memory Workshop (IMW)*, May 2010.

[36] T. Kgil, D. Roberts, and T. Mudge, “Improving NAND flash based disk caches,” in *35th International Symposium on Computer Architecture (ISCA’08)*. IEEE, 2008, pp. 327–338.

[37] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA’09)*. New York, NY, USA: ACM, 2009, pp. 24–33.

[38] S. Phadke and S. Narayanasamy, “MLP aware heterogeneous memory system,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2011, pp. 1–6.

[39] O. Avissar, R. Barua, and D. Stewart, “Heterogeneous memory management for embedded systems,” in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’01)*. New York, NY, USA: ACM, 2001, pp. 34–43.

[40] P. R. Panda, N. D. Dutt, and A. Nicolau, “On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 682–704, Jul. 2000.

[41] J. Marathe, V. Thakkar, and F. Mueller, “Feedback-directed page placement for ccNUMA via hardware-generated memory traces,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, pp. 1204–1219, 2010.

[42] Z. Majo and T. R. Gross, “Matching memory access patterns and data placement for NUMA systems,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 230–241.