Automating the Application Data Placement in Hybrid Memory Systems

Harald Servat*, Antonio J. Peña†, Germán Llort†,
Estanislao Mercadal†, Hans-Christian Hoppe* and Jesús Labarta†‡
*Intel Corporation
†Barcelona Supercomputing Center (BSC)
‡Universitat Politècnica de Catalunya (UPC)

***Abstract*—Multi-tiered memory systems, such as those based on Intel® Xeon Phi™ processors, are equipped with several memory tiers with different characteristics including, among others, capacity, access latency, bandwidth, energy consumption, and volatility. The proper distribution of the application data objects into the available memory layers is key to shorten the time–to–solution, but the way developers and end-users determine the most appropriate memory tier to place the application data objects has not been properly addressed to date.**

In this paper we present a novel methodology to build an extensible framework to automatically identify and place the application's most relevant memory objects into the Intel Xeon Phi fast on-package memory. Our proposal works on top of in-production binaries by first exploring the application behavior and then substituting the dynamic memory allocations. This makes this proposal valuable even for end-users who do not have the possibility of modifying the application source code. We demonstrate the value of a framework based in our methodology for several relevant HPC applications using different allocation strategies to help end-users improve performance with minimal intervention. The results of our evaluation reveal that our proposal is able to identify the key objects to be promoted into fast on-package memory in order to optimize performance, leading to even surpassing hardware-based solutions.

***Index Terms*—heterogeneous memory, hybrid memory, high-bandwidth memory, performance analysis, PEBS, sampling, instrumentation**

I. Introduction

Hybrid memory systems (HM)[1] accommodate memories featuring different characteristics such as capacity, bandwidth, latency, energy consumption, or volatility. A recent example of an HM-processor is the Intel® Xeon Phi™, containing two memory systems: DDR and on-package Multi-Channel DRAM (MCDRAM) [1]. While HM systems present opportunities in different fields, the efficient usage of these systems requires prior application knowledge because developers need to determine which data objects to place in which of the available memory tiers. A common objective is to shorten the application time–to–solution and this translates into placing the appropriate data objects on the fastest memory. However, fast memory is a scarce resource and the application working set may not fit. Consequently, it is important to characterize the application behavior to identify the (critical) data that benefits the most from being hosted into fast memory and, if not sufficient, keep the non-critical data away in slower memory.

[1]Also known as *heterogeneous* or *multi-tiered* memory systems.

Tools like EVOP [2], ADAMANT [3], MACPO [4] and Intel® Advisor [5] solely rely on instruction-level instrumentation to monitor the data object allocation and their respective accesses to advise the user about the most accessed variables and even their access patterns. These metrics are valuable to understand the application behavior, but the imposed overhead limit their relevance because they alter the application performance and generate a daunting amount of data for in-production application runs, leading to long analysis times.

To overcome these limitations, processor manufacturers have augmented their performance monitoring unit with sampling mechanisms to provide rich information, including the referenced memory address. Precise-Event Based Sampling (PEBS) is the implementation of such a feature in recent Intel processors [6]. Performance analysis tools such as HPCToolkit [7], MemAxes [8], Extrae [9], and Intel® Vtune™ Amplifier [10] use a hybrid approach combining instrumentation to track data allocation and PEBS to monitor the application data references. This approach enables exploring in-production executions with a reduced overhead at the cost of providing statistical approximations, even though approximations for long runs resemble the actual results.

These tools typically identify the data structures that are associated to metrics like high-latency loads or high number of cache misses but leave the tedious work of substituting the memory allocation calls to the application developer. This paper advances the current state of the art by introducing a novel framework design to help end-users, application developers and processor architects understand the usage of HM systems, and to automatically promote the critical data to the appropriate memory layer. The framework design consists of four stages: (1) low-overhead data collection using hardware-based sampling mechanisms; (2) attribution of a cost based on Last-Level Cache (LLC) misses to each data object; (3) distribution of data objects for a given memory configuration; and (4) re-execution of the application binary automatically promoting the different data objects to the proper memory tier. The two first stages rely on an unmodified open-source tools, while the third stage is a derivative from an already existing tool and the fourth stage relies newly developed interposition library.

The contributions of this paper include:

1) the design of an extensible framework to automatically distribute the data objects of in-production binaries in

 DOI: 10.1109/CLUSTER.2017.50

HM systems targeting performance, and implementing it for Intel Xeon Phi processors;
2) an exploration of several strategies to help determine which application variables to place on which memory tier in HM systems;
3) the evaluation of the proposed distribution approaches on a set of well-known benchmarks using the presented framework methodology, including a comparison with already existing hardware and software solutions; and
4) the proposal of a novel metric to report the *efficient use* of the fast on-package memory by applications.

This paper follows contextualizing the work we present with already existing state–of–the–art tools and methodologies in Section II. Section III describes in detail the framework and its components. In Section IV we put the framework in use through several benchmarks and applications while analyzing the obtained results. Finally, Section V draws conclusions and discusses possible future research directions.

II. Related Work

There exist several alternatives for taking advantage of the MCDRAM on Intel Xeon Phi processors. The user can benefit from the fast on-package memory transparently by using it as a direct-mapped LLC. However, if MCDRAM is configured in flat mode (*i.e.* sits on a different part of the address space), then the easiest alternative for the user is to rely on the `numactl` command to place as much application data as possible into the fast memory. Another alternative is to use the `autohbw` library provided by the *memkind* package [11]. This library is injected into the application before process execution and it forwards dynamic allocations into MCDRAM if the requested memory is within a user-given size range (as long as it fits). The most tedious situation requires the developer to learn (somehow) about the application behavior with respect to main memory accesses and manually change the memory allocations so that they reside on MCDRAM using *memkind*. Even though using MCDRAM in cache mode leads to good performance results, it is not as efficient as consciously exploiting it in flat mode (see Figure 1), especially for those workloads where the lack of associativity is a problem. Using the `numactl` approach, irrelevant data objects may be placed on MCDRAM and prevent critical objects from fitting, while using `autohbw` or changing the application code requires detailed application knowledge.

We next describe earlier approaches from a variety of performance tools that have focused on the analysis of data structures to bring this knowledge to the user. We divide this research into two groups depending on the mechanism used to capture the addresses referenced by the load/store instructions. Then, we describe how these approaches fit within the framework design we propose.

a) Instrumenting-based Solutions: The first group of tools includes those that instrument the application instructions to obtain the referenced addresses. MemSpy [12] is a prototype tool for profiling applications on a system simulator that introduces the notion of data-oriented—in addition to code-oriented—performance tuning. This tool instruments every memory reference from an application run and leverages the references to a memory simulator that calculates statistics such as cache hits and misses for a given cache organization. SLO [13] suggests locality optimizations by analyzing the application reuse paths to find the root causes of poor data locality. This tool extends the GCC compiler[2] to capture the application's memory accesses, function calls and loops to track data reuses, and then it analyzes the reused paths to suggest code loop transformations. MACPO captures memory traces and computes metrics for the memory access behavior of source-level data structures. The tool uses PerfExpert [14] to identify code regions with memory-related inefficiencies, then it employs the LLVM compiler[3] to instrument the memory references, and finally it calculates several reuse factors and the number of data streams in a loop nest. Intel Advisor is a performance analysis tool that focuses on the thread and vector performance, and it also explores the memory locality characteristics of a user-given code. The tool relies on PIN [15] to instrument binaries at instruction-level allowing correlation of the instructions and the memory access patterns. Tareador [16] is a tool that estimates the amount of parallelism that can be extracted from a serial application using a task-based data-flow programming model. The tool employs dynamic instrumentation to monitor the memory accesses of delimited regions of code to determine whether they can simultaneously run without data race conditions, and then it simulates the application execution based on this outcome. EVOP is an emulator-based data-oriented profiling tool to analyze actual program executions in a system equipped only with a DRAM-based memory [17]. EVOP uses dynamic instrumentation to monitor the memory references in order to detect which memory structures are the most referenced and then estimate the CPU stall cycles incurred by the different memory objects to decide their optimal object placement in a heterogeneous memory system by means of the *dmem_advisor* tool [2]. ADAMANT uses the PEBIL instrumentation package [18] and includes tools to characterize application data objects, to provide reports helping on algorithm design and tuning by devising optimal data placement, and to manage data movement improving locality.

b) Hardware-based Solutions: The second group consists of tools that benefit from hardware mechanisms to sample addresses referenced when processor counter overflows occur and that estimate the access cost from the samples. The Oracle Developer Studio (formerly known as Sun ONE Studio) incorporates a tool to explore memory system behavior in the context of the application's data space [19]. This extension provides the analyst with independent and uncorrelated views that rank program counters and data objects according to hardware counter metrics and it shows metrics for each element in data object structures. HPCToolkit was extended to support data-centric profiling of parallel programs using hardware sampling capabilities to expose the long latency memory operations. Similarly, Intel Vtune Amplifier shows application data objects that induce more cache misses. These two tools provide their respective graphical user interface that

[2]http://gcc.gnu.org
[3]http://www.llvm.org

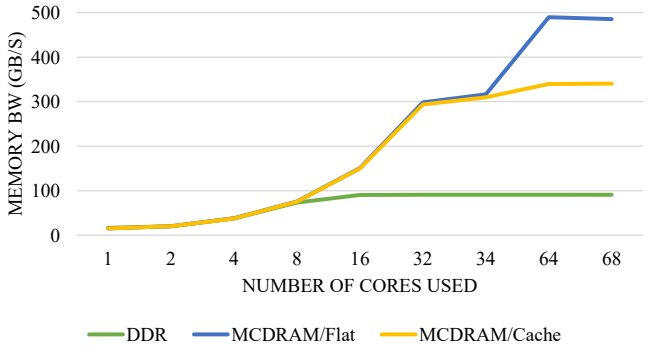


Fig. 1: Bandwidth observed on the Triad kernel of the Stream benchmark when executed with a single thread per core on an Intel Xeon Phi processor 7250 running at 1.40 GHz and placing the data in DDR or MCDRAM.

presents data- and code-centric metrics, easing the correlation among the two. MemAxes uses PEBS to monitor long-latency load instructions that access addresses within memory regions delimited by user-specified data objects. The novelty of its approach is that it associates the memory behavior with semantic attributes, including the application context which is shown through a visualization tool. BSC tools have been extended to sample memory references and then show detailed access patterns on the application address space, and correlate them with the application code and other performance counters through the Folding technique [20].

The work described in this paper combines aforementioned mechanisms to identify the data objects and report which would benefit the most from moving to a faster memory. The report is then analyzed by a novel mechanism that automatically substitutes dynamic allocations referring to critical data at run-time, letting developers and end-users apply the method easily even for production binaries. This approach leverages finer granularity than that of `autohbw`. Although we have used the BSC tools and a *dmem_advisor* derivative to leverage the data analysis and object selection stages of our proposed methodology, it is possible to swap them with analogous tools.

III. Design and Proposed Implementation

We present an overview of the framework and its main components in this section. The components of the framework are executed sequentially as illustrated in Figure 2 leading to a *profile-guided execution*. The framework starts by collecting metrics of the memory objects into a trace-file by using Extrae. Then, Paramedir [21] identifies those objects that have missed the most in the LLC (and likely to be the most bandwidth-demanding) and their respective sizes. Third, *hmem_advisor* reports which memory objects are best to place in fast memory according to a given memory specification. Finally, *auto-hbwmalloc* automatically substitutes the regular allocation memory calls to MCDRAM memory honoring the previous report in a final application execution. The following subsections provide further details on the components of our framework proposal.

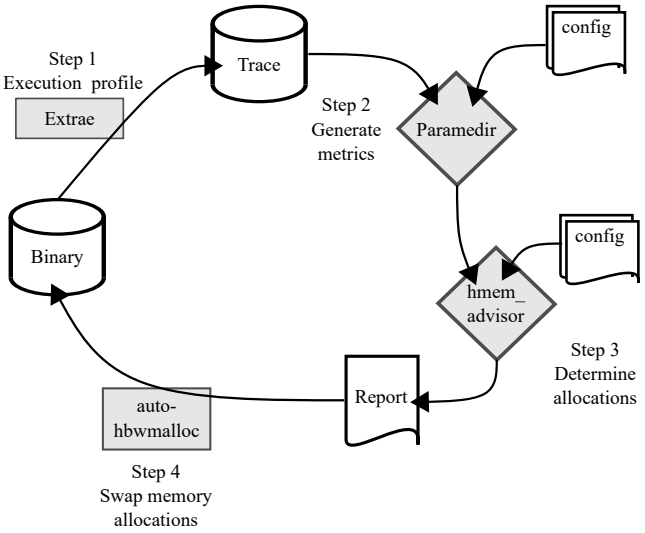


Fig. 2: The framework design and its components.

Step 1: Extrae

Extrae is an open-source tracing package developed at BSC that generates Paraver trace-files. This package automatically instruments applications using the `LD_PRELOAD` mechanism to capture information from parallel programming models such as MPI, OpenMP, POSIX threads, OpenCL, CUDA and combinations of them. Extrae complements the trace-files with sampling mechanisms, making sure that performance analysts get performance details even for long uninstrumented regions.

While Extrae has traditionally focused on capturing the activity of parallel runtimes, it has been recently extended to instrument memory allocations, and to sample load and store instructions from the application using the PEBS mechanism, to include further information regarding data objects and their accesses. This information includes the time-stamp, performance counters, and the parameters and results of the call, (*i.e.* requested size, input and output pointers), and the call-stack. Extrae uses *binutils* [22] to obtain human-readable source code references for the memory accesses. Dynamically-allocated variables are identified by their allocation call-stack[4] while static variables are referenced by their given name.

Although Extrae is able to collect data from many sources of information, to perform this analysis the framework only needs dynamic-memory allocations and deallocations and sampled memory references for the LLC misses. For the former, Extrae registers the allocated address range through the returned pointer and the size of the allocation. For the latter, Extrae registers the address of the particular load or store instruction that missed in LLC, and it correlates with its corresponding object by matching the accessed address against the previously allocated object's address ranges. The association of memory references to automatic (stack) variables is not supported at the time of writing this document.

Regarding the PEBS hardware infrastructure, the metrics associated to the memory samples depend on the processor fam-

[4]The call-stack is captured using the `backtrace()` call from *glibc*.

ily as well as on the performance counter used. For instance, the PEBS mechanism in the Intel Xeon Phi processors tracks L2 (LLC) cache load references (either hits or misses) and provides information regarding the address being referenced. The information provided for Intel® Xeon ® processors is richer: it additionally provides the access cost (in cycles) and which part of the memory hierarchy provided the data for load instructions, and whether the access did hit or missed in the L1 cache for store instructions.

Step 2: Paramedir

The result of an instrumented run with Extrae is a Paraver trace-file, a sequence of time-stamped events reflecting the actual application execution. Paraver is the visualization tool of the BSC tool-suite, which enables users to conduct a global qualitative analysis of the main performance issues in the execution by visual inspection, and then focus on the detailed quantitative analysis of the detected bottlenecks. These analyses can be stored in the so-called *configuration files* that can be applied to any trace-file as long as it contains the necessary data. Paramedir, on the other hand, is the non-graphical version of Paraver which allows to automatize the analysis through scripts and configuration files, reporting metric values in a comma-separated-value (CSV) file.

In this stage Paramedir is applied to compute two statistics from the trace for each application data object: (1) the cost of the memory accesses, and (2) the size of the object. We approximate the access cost by the number of LLC misses, but this could be easily extended on Intel Xeon processors thanks to their richer PEBS infrastructure. Regarding the object's data size, it is worth mentioning that dynamically-allocated objects are identified by their call-stack. If an application loops over a data allocation, the call-stack will be the same for each iteration, and hence it can not unequivocally distinguish among the different allocations. In these cases we report the maximum requested size observed for each repeated allocation site.

Step 3: hmem_advisor

hmem_advisor is a tool based on EVOP's *dmem_advisor*. It parses Paramedir's output containing the object-differentiated memory access information and computes an optimized object distribution among the available memory layers. Like *dmem_advisor*, *hmem_advisor* is based on a relaxation of the 0/1 multiple knapsack problem (solving separate knapsacks in descending order of memory performance at memory page granularity), where the memory subsystems represent the knapsacks and the memory objects correspond to the items to be packed. Each memory subsystem is defined by a given size and a relative performance in a configuration file, ensuring that we can extend this mechanism in the future for different memory architectures.

Ideally, we want to minimize the number of stalled cycles by the CPU due to main memory accesses. We achieve this by maximizing the potential CPU stall cycles due to memory accesses that each memory tier avoids with respect to the slowest of them. We approach this as the number of per-object accesses (*i.e.*, LLC misses), as proposed in [2]. We also devise a future additional refinement enabled by our approach based on the PEBS metrics provided in Intel Xeon processors benefiting from object-differentiated information on miss latency.

Computing a pure 0/1 knapsack (with pseudo-polynomial computational cost) involving potentially hundreds of memory objects and large memory levels has proven to be impractical in our experiments. We approach this problem by implementing in *hmem_advisor* two independent and greedy relaxations of the problem. The first alternative is an approach that selects the data objects based on the number of LLC misses and an optionally user-provided percentage threshold. The threshold allows preventing that rarely referenced objects (but that still fit in the knapsack) are promoted to fast-memory. The second alternative is a relaxation based on profit density, *i.e.* promoting those variables with higher $memory\ access/data\ object\ size$ ratio. Either approach has a linear computational cost. No matter the approach leveraged, the current *hmem_advisor* implementation considers that the application address space is static. While this assumption does not hold true for all applications, it may be reasonable for many applications that allocate data from the start and keep it until they finalize. Since the generated trace-file in the first stage of the framework contains a time-varying representation of the application address space, *hmem_advisor* could use this information to further tune the suggested allocations.

The output of the tool is a list of selected data objects that should be promoted to fast memory. This list is written in a human-readable format for two reasons. First, statically allocated objects cannot be migrated to a memory layer different from the default without modifying the application code. Second, application developers may prefer to have full control of the memory placement and modify the code themselves to migrate the selected data objects into a different memory tier.

Step 4: auto-hbwmalloc

The *auto-hbwmalloc* component consists of a shared library that substitutes several dynamic-memory allocation and deallocation calls[5] through the `LD_PRELOAD` mechanism and forwards them to an alternate memory allocator. Currently, the *auto-hbwmalloc* forwards memory allocations to routines from the *memkind* library, but the *auto-hbwmalloc* component has been developed so that it can be easily extended to other allocation mechanisms. At the moment the library only supports dynamically-linked binaries but we foresee the possibility of substituting the memory-related calls in statically-linked binaries using instrumentation frameworks such as PIN or DynInst [23].

The library contains wrappers to substitute all the memory-related calls and use the information provided by *hmem_advisor* to replace the selected dynamic allocations. Algorithm 1 shows an example of this interposition for the `malloc` call with details explained through this section. Each time the application invokes a `malloc`, the library intercepts the call and then checks whether the invocation call-stack matches with any of those identified in the report from step

[5]Including `malloc`, `realloc`, `posix_memalign`, `free`, `kmp_malloc`, `kmp_aligned_malloc`, `kmp_free` and `kmp_realloc`.

Algorithm 1 Pseudo-code for a substituted `malloc`.

```
1: function MALLOC(size)
2:     allocated ← false
3:     if lb_size ≤ size ≤ ub_size then
4:         callstack ← BACKTRACE( )
5:         <found, in, alloc> ← ALLOC_CACHE_SEARCH(callstack)
6:         if ¬found then
7:             tx_callstack ← CS_TRANSLATE(callstack)
8:             <in, alloc> ← MATCH(tx_callstack, sel_callstacks)
9:             ALLOC_CACHE_ANNOTATE(callstack, in, alloc)
10:        end if
11:        if in then
12:            if alloc → FITS(size) then
13:                ptr ← (alloc → MALLOC(size))
14:                ALTERNATE_REGION_ANNOTATE(ptr, size, alloc)
15:                alloc → STATS_ADD(size)
16:                allocated ← true
17:            end if
18:        end if
19:    end if
20:    if ¬allocated then
21:        ptr ← (posix → MALLOC(size))
22:        posix → STATS_ADD(size)
23:    end if
24:    return ptr
25: end function
```

3 (line 8). In case of a positive match, it returns a pointer to the appropriate allocator object (`alloc`) that forwards the allocation to the selected memory allocation call (in this case *memkind*) and is used to allocate the data object (line 13). Due to the inclusion of the ASLR (Address Layout Space Randomization) security features that randomize the position of library symbols in the application address space, it is necessary not only to unwind (line 4) the call-stack but also to translate it at run-time (using the *binutils* package [line 7]).

The library itself needs to perform some book-keeping including the following items: (1) allocated regions per allocator, (2) memory used per allocator and (3) execution statistics. First, memory allocations and deallocations need to be handled by their specific memory allocation package and cannot be mixed with others. This makes it necessary to keep a relation of which allocations have been done by the alternate allocators in order to use the appropriate calls (line 14). Second, in Step 2 we mentioned that the framework currently reports the highest allocation values for dynamically-allocated objects in loops. This means that *hmem_advisor* may not be aware of the exact amount of memory used by an application *a priori*. We have implemented *auto-hbwmalloc* so that it keeps the total amount of alternate space used by the process (line 15) and it will not request from the alternate allocator more memory than that specified by the advisor (line 12). This approach also covers the case where applications allocate memory from inlined routines. In this case, different allocation sites sharing the same call-stack may exist and thus mislead the library to substitute allocations when it should not. Third, and finally, the *auto-hbwmalloc* component also captures several application metrics upon user request that may be valuable for analysis and debug purposes. These metrics include, among others, the number of allocations, the average allocation size, the observed High-Water Mark (HWM) and whether any variable did not fit into memory due to user size limitations given to *hmem_advisor*.

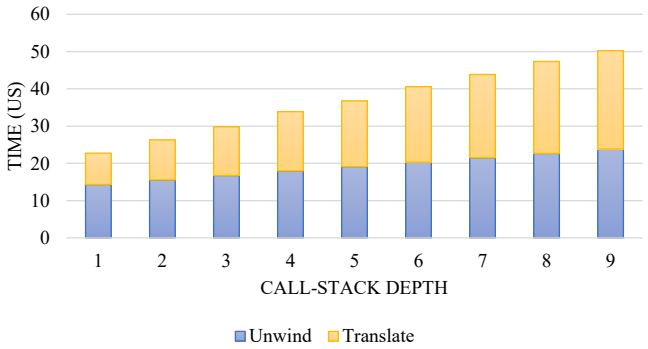


Fig. 3: Overhead breakdown for call-stack unwinding and call-stack translation on an Intel Xeon Phi 7250 processor running at 1.40 GHz using *glibc* 2.17 and *binutils* 2.23.

Since applications may face large number of memory call invocations during execution, we have also explored the overhead that these may suffer using this library in order to evaluate whether the overhead could hide the gains by promoting the data objects to MCDRAM. Figure 3 shows a breakdown of the unwind and translation cost (Y-axis in μseconds) when varying the call-stack depth (X-axis). The results show that the cost of unwinding a short call-stack is larger compared to the cost of translating its frames, but the translation cost increases faster than the unwind cost when increasing the call-stack depth. In this particular case, the translate cost surpasses the unwind cost eventually and for the machine tested this occurs when processing a call-stack deeper than 6 levels. We address this overhead with two approaches. First, we include a small cache indexed by the unwound addresses that keep whether an allocation invoked in that position shall or shall not be allocated using the alternate allocator (lines 5 and 9). Second, the *hmem_advisor* tool provides the lowest and highest allocation sizes (`lb_size` and `ub_size`) to filter the allocations to be checked by their size (line 3), although this can be disabled upon user request.

IV. Experimental Evaluation

To demonstrate the value of the proposed framework we evaluate the following applications and provide some of their characteristics in Table I. The applications include:

- High Performance Conjugate Gradient (HPCG) [24] - a code that benchmarks computer systems based on a simple additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient.
- Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (Lulesh) proxy application [25] - a representative of simplified 3D Lagrangian hydrodynamics on an unstructured mesh.
- Block-Tridiagonal (BT) benchmark - part of the NAS parallel benchmarks [26] that mimics the computation and data movement in CFD applications.
- MiniFE - a proxy application for unstructured implicit finite element codes from the Mantevo and CORAL benchmark collections [27].

TABLE I: Explored applications and their characteristics.

	HPCG 3.0mod [24]	**Lulesh** 2.0 [25]	**NAS BT** 3.3.1 [26]	**miniFE** 2.0rc3 [27]
Lines of code	5,718	7,240	6,415	4,609
Language	C++	C++	Fortran	C++
Parallelism	MPI+OpenMP	MPI+OpenMP	OpenMP	MPI+OpenMP
Execution geometry	64 ranks, 4 threads/rank	64 ranks, 4 threads/rank	272 threads	64 ranks, 4 threads/rank
Problem size	104^3, 400s	96^3, 50 its	D 408^3, 250 its	$520\times512\times512$, 200 its
Compilation flags	-g -O3 -xMIC-AVX512 -qopenmp	-g -O3 -xMIC-AVX512 -qopenmp -fno-inline	-g -O3 -xMIC-AVX512 -qopenmp -mcmodel=medium	-g -O3 -xMIC-AVX512 -qopenmp
Figure of Merit (FOM)	GFLOPS	z/s	Mop/s	MFLOPS
Allocation statements[6]	0/0/0/33/17/0/0	1/0/1/35/23/0/0	0/0/0/0/0/15/15	0/0/0/5/1/0
Number of allocations/process/second	3,263	29.48	0.49	1,006.55
Memory used-HWM[7] (MB/process [total])	928 [59,399]	859 [54,992]	11,136 [11,136]	1,022 [65,439]
Monitoring overhead[8]	0.42%	0.29%	0.32%	4.10%
Number of samples/process	13,629	3,201	38,215	3,194
Number of samples/process/second	30.46	9.08	12.59	12.25

	CGPOP 1.0 [28]	**SNAP** 1.0.7 [29]	**MAXW-DGTD** [30]	**GTC-P** 160328 [31]
Lines of code	4,612	8,583	20,835	8,362
Language	Fortran	Fortran	Fortran	C
Parallelism	MPI	MPI+OpenMP	MPI+OpenMP	MPI+OpenMP
Execution geometry	64 ranks	64 ranks, 4 threads/rank	64 ranks, 4 threads/rank	64 ranks, 4 threads/rank
Problem size	180×120, 200 trials	$32\times64\times64$, 20 its	4th order	mi=3,861,390, 50 its
Compilation flags	-g -O3 -xMIC-AVX512	-g -O3 -xMIC-AVX512 -qno-opt-dynamic-align -fno-fnalias -qopenmp	-g -O3 -xMIC-AVX512 -qopenmp -align dcommons	-g -O3 -xMIC-AVX512 -qopenmp
Figure of Merit (FOM)	Trials / s	Iterations / s	Iterations / s	Iterations / s
Allocation statements[6]	0/0/0/0/0/29/6	0/0/0/5/1/0/0	0/0/0/0/0/75/71	156/0/156/0/0/0/0/0
Number of allocations/process/second	18.17	1,006.55	15,853.98	20.57
Memory used-HWM[7] (MB/process [total])	158 [10,173]	1,022 [65,439]	285 [18,276]	1,329 [85,074]
Monitoring overhead[8]	0.88%	0.15%	0.65%	0.78%
Number of samples/process	8,258	3,194	2,072	17,254
Number of samples/process/second	17.44	12.25	4.13	28.56

- CGPOP [28] miniapp - the conjugate gradient solver from LANL POP (Parallel Ocean Program) 2.0, which is the performance bottleneck for the full POP application.
- SNAP [29] - a proxy application to model the performance of a modern discrete ordinates neutral particle transport application solving the linear Boltzmann transport equation in multi-dimensional phase space.
- MAXW-DGTD [30] - an adoption of a Discontinuous Galerkin Time-Domain solver for computational bioelectromagnetics which use 4$^{\text{th}}$ order Lagrange basis functions on tetrahedra for the simulation of human exposure to electromagnetic waves [32].
- Princeton Gyrokinetic Toroidal Code (GTC-P) [31] - a simulator for plasma turbulence within Tokamak fusion devices generating a magnetic field that confines a plasma within a toroidal cavity and accelerates the plasma particles around the torus.

We have used most of the applications out–of–the–box from sources, changing only the compilation process as stated in Section IV-A and Table I. The only modifications were performed in BT, CGPOP and HPCG. Regarding BT and CGPOP, the first analyses with the framework indicate that all the variables that should go into MCDRAM are static variables. However, since our interposition library cannot promote static and automatic variables into fast memory, we modified the most observed variables in BT and CGPOP to be dynamically allocated so that they can be intercepted. Regarding HPCG, we have slightly modified the reference code using some well-known modifications (provided by the official website) that improve the application performance (see [24] for details).

A. System Setup

We have used a system with one Intel Xeon Phi 7250 processor running at 1.40 GHz. The system has 96 Gbytes and 16 Gbytes of DDR and MCDRAM memory, respectively, meaning that the majority of the working sets do not fit in fast memory. All the experiments have been executed in flat mode except for the experiments labeled accordingly. The processor tiles are interconnected using the quadrant cluster mode.

Regarding the software, the machine runs CentOS Linux 7 with a Linux kernel 3.10.0 and the XPPSL package version 1.3.3. All the explored applications have been compiled using Intel® C/C++ and Fortran compilers version 2017 update 2 with aggressive optimization flags and generating debug information. We note that for Lulesh we have disabled inlining because the application uses it aggressively and this confuses *auto-hbwmalloc* due to the same call-stack being reported from many different allocation sites. MPI applications use Intel® MPI library 2017 update 2 and we have not utilized the MCDRAM memory for the internal MPI buffers to avoid

[6]Direct allocation statements in format: *m/r/f/n/d/a/D*, where *m* stands for `malloc`, *r* for `realloc()`, *f* for `free()`, *n* for `new`, *d* for `delete`, *a* for `allocate` and *D* for `deallocate`. Note that container allocations (such as C++ STL allocations) are not reported and that `allocate` and `deallocate` may operate on multiple data objects in a single invocation.

[7]As reported by each process by the Virtual Memory High-Water Mark (VmHWM) in `/proc/self/status` before the process termination.

[8]The overhead is calculated using the reported FOM.

interfering with our framework. We use Extrae version 3.4.3 to monitor memory allocations larger than 4 Kbytes and to sample one out of every 37,589 L2 cache misses. We have chosen this allocation size to avoid small (and possibly frequent) allocations such as those related to I/O that are unlikely to benefit from MCDRAM. The period is a relatively large number to keep the impact on application execution small (typically below 1%) although the number of samples depends on the instruction mix (see details in Table I). Finally, the auto-hbwmalloc library employs *memkind* version 1.5.0 to allocate selected objects in fast memory.

B. Application Analyses

Following our proposed framework, we have monitored the applications to obtain trace-files containing their memory allocations and sampled references. Then we applied the *hmem_advisor* tool with a range of memory sizes and several allocation strategies. In OpenMP-only applications (*i.e.* NAS BT) the exploration size ranges from 32 Mbytes to 16 Gbytes. MPI (and hybrid MPI+OpenMP) applications have been run with 64 MPI ranks; we explore the performance obtained when limiting the used MCDRAM memory in a range from 32 to 256 Mbytes per rank. With respect to allocation strategies, we have explored the two independent strategies provided in *hmem_advisor*: based on LLC misses (with 0%, 1% and 5% thresholds) and the density-based. Each parameter combination generates an object distribution and then we run again the application with the *auto-hbwmalloc* library to check for improvements by placing the selected objects in MCDRAM.

For comparison purposes with the results obtained using our framework, we have also executed the applications in four execution conditions and we report the results in Figure 4. First, and as a reference, we measure the application performance when the workload is located in DDR memory. The second execution refers to placing as much data as possible into MCDRAM following a FCFS strategy until it is exhausted and then fall-backs to DDR (*i.e.* using `numactl -p 1`). Third, we have used the *libautohbw* library to automatically place dynamically-allocated data objects larger than 1 Mbyte in fast memory. Finally, we have also conducted the experiments when using the MCDRAM in cache mode allowing the user to completely ignore data object placement.

C. Analysis of the Results

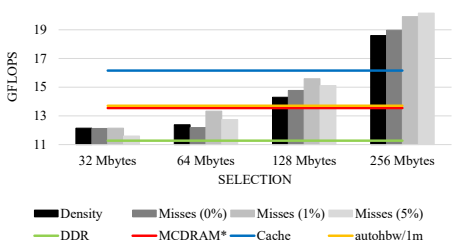
Figure 4 shows the results of the different experiments executed per application (row). There are three figures per application (from left to right: absolute performance, MCDRAM utilization and good use of the MCDRAM). The absolute performance refers to the application Figure of Merit (FOM—the higher the better) if the application reports it and for the applications that do not report FOM (*i.e.*, SNAP, CGPOP, MAXW-DGTD and GTC-P) the Figure shows the $iterations/time$ metric. The values represented in columns show the performance achieved when executed with our proposed framework and the given configuration based on memory size limit and allocation strategy. The horizontal lines help to compare the performance with other approaches, namely: when the application is executed in DDR (green), when placing as much as data as possible in fast memory through `numactl -p 1` (red), when using `autohbw` with 1 Mbyte threshold (yellow) and when setting MCDRAM in cache mode (blue). The plots in the middle report the highest MCDRAM memory utilization (HWM) as reported by the *auto-hbwmalloc* library. This information allows us to understand the application allocation necessities and explore how much of the assigned memory has been actually used. Finally, the plot on the right column depicts the $\Delta FOM/mbyte$ metric. This metric represents the good use of fast memory's provided size and helps identifying sweet-spots for dimensioning different memory tiers on HM machines. A naïve description of this metric refers to the performance increase achieved when using a given amount of fast memory. The metric is calculated as:

$$\Delta FOM/mbyte_x(y) = (FOM_x(y) - FOM_{ddr}(y))/MEM_x \quad (1)$$

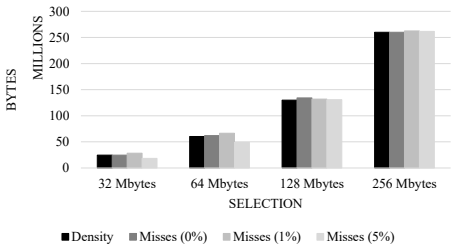
where x refers to an experiment, y refers to an application, FOM_{ddr} represents the FOM achieved when running the application in the reference (DDR) memory, FOM_x and MEM_x represent the FOM obtained and MCDRAM-size when using the selection in experiment x, respectively. Since we do not have the HWM value for cache and `numactl -p 1` experiments we have decided their MEM_x value to be 16 Gbytes as this is the provided memory. The `autohbw/1m` experiment has been left out of this analysis because we do not know the exact amount of data promoted to MCDRAM.

a) Performance-wise remarks: First, and focusing on the columns of the performance plots (plots on the left), we observe the expected behavior where the more data placed in fast memory, the higher the performance (see HPCG [4a], Lulesh [4d], BT [4g], miniFE [4j], MAXW-DGTD [4s] and GTC-P [4v]). The exceptions to this behavior are SNAP (4p) and CGPOP (4m). As stated earlier, CGPOP has been modified to change most of its static allocations into dynamic allocations, and the latter already fit in the smaller case (32 Mbytes per process), so adding more memory does not provide any benefit. Regarding the knapsack alternatives, we observe that performance is typically on par with few examples were `density` behaves better (Lulesh [4d] and GTC-P [4v]) and one (HPCG [4a]) where `Misses(5%)` is better.

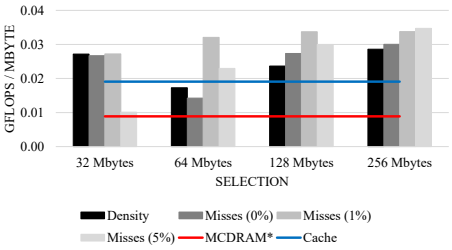
When comparing the absolute performance from the various placement alternatives (*i.e.* columns *vs* horizontal lines) we notice the following. Our framework provides best results for HPCG (4a), miniFE (4j) and GTC-P (4v). The best case of HPCG shows a 78.88% performance increase when compared to the DDR execution and 24.82% performance increase when compared to the second best alternative (cache). The cache mode is superior for Lulesh (4d) and slightly superior for MAXW-DGTD (4s). For instance, the best case of Lulesh in cache mode is 46.98% faster than executing in DDR and 12.68% faster than the second best alternative (using our proposed framework for 256 Mbytes per process with the density approach). The usage of `numactl -p 1` command outperforms marginally the cache and framework approaches on BT (4g), CGPOP (4m) and SNAP (4p). We have also explored the reason for the difference in Lulesh and SNAP when compared to our framework. With respect to Lulesh, it allocates and deallocates many objects during the application



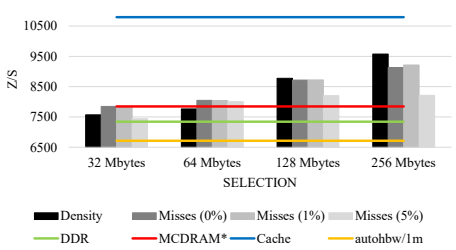
(a) HPCG - FOM



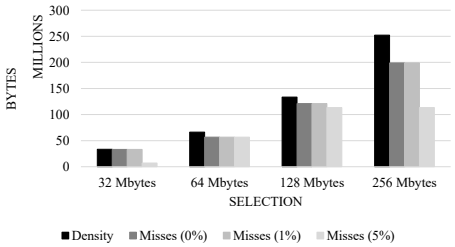
(b) HPCG - HWM



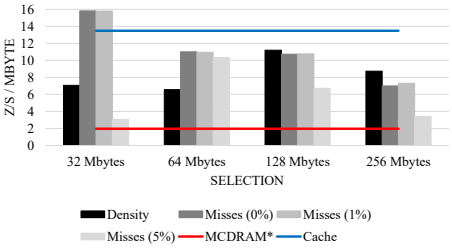
(c) HPCG - $\Delta FOM/mbyte$



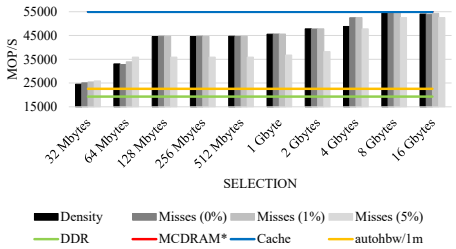
(d) Lulesh - FOM



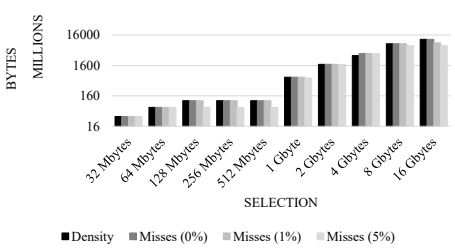
(e) Lulesh - HWM



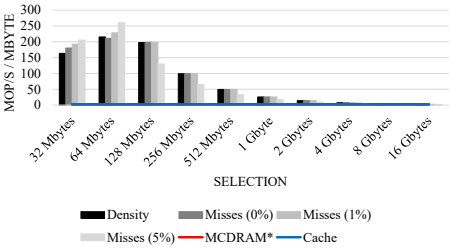
(f) Lulesh - $\Delta FOM/mbyte$



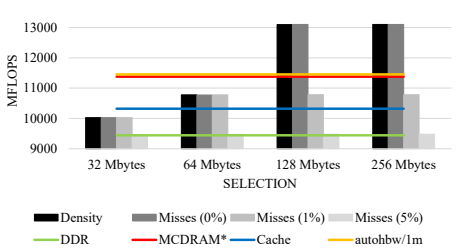
(g) BT - FOM



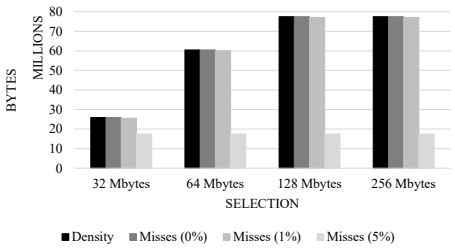
(h) BT - HWM



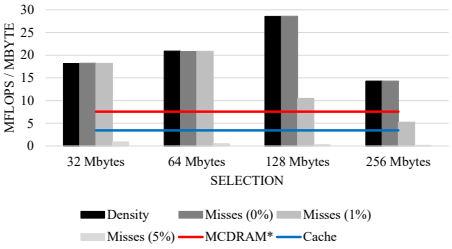
(i) BT - $\Delta FOM/mbyte$



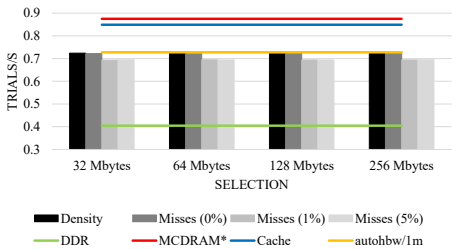
(j) miniFE - FOM



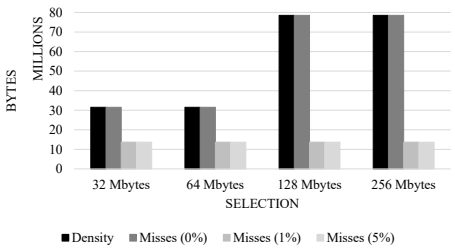
(k) miniFE - HWM



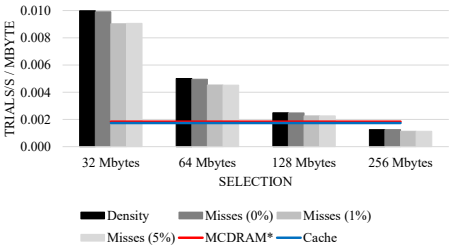
(l) miniFE - $\Delta FOM/mbyte$



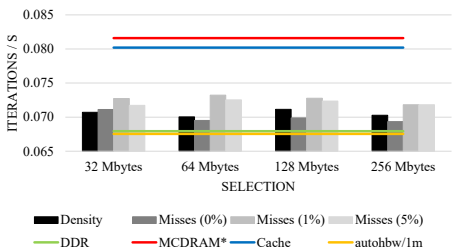
(m) CGPOP - FOM



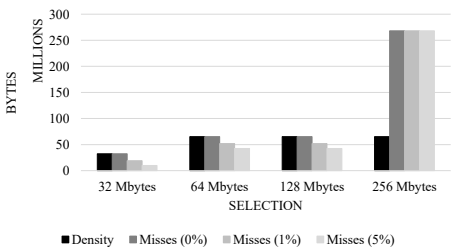
(n) CGPOP - HWM



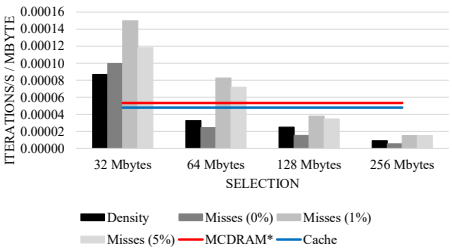
(o) CGPOP - $\Delta FOM/mbyte$



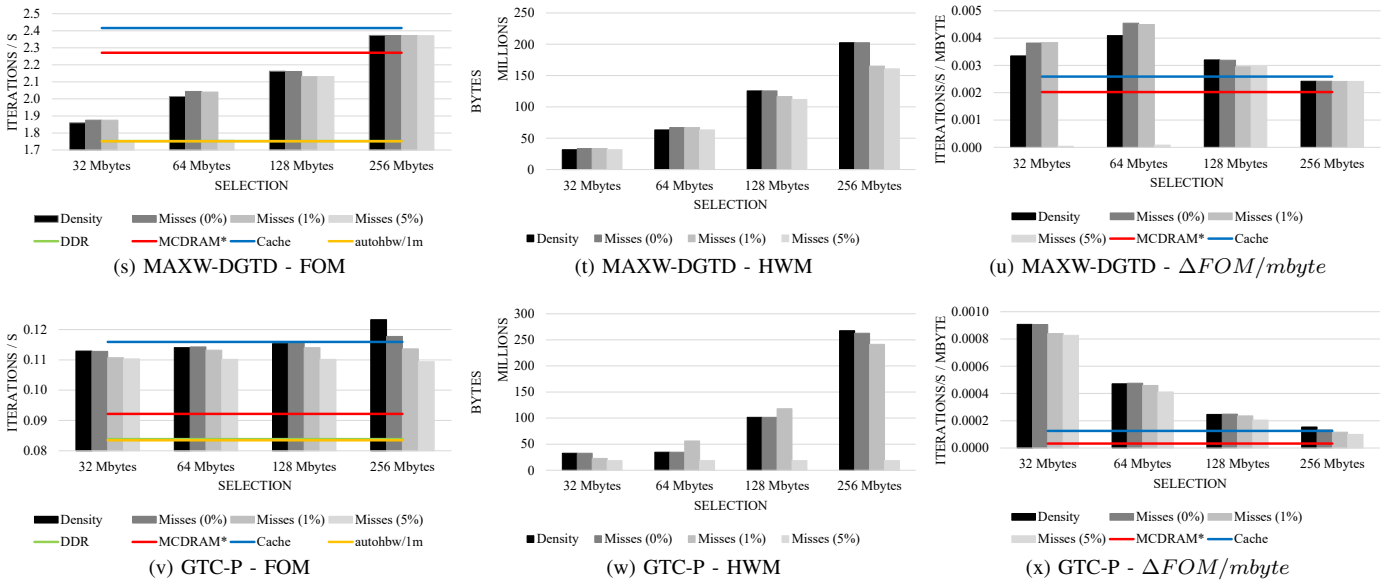
(p) SNAP - FOM



(q) SNAP - HWM



(r) SNAP - $\Delta FOM/mbyte$



(s) MAXW-DGTD - FOM (t) MAXW-DGTD - HWM (u) MAXW-DGTD - $\Delta FOM/mbyte$

(v) GTC-P - FOM (w) GTC-P - HWM (x) GTC-P - $\Delta FOM/mbyte$

Fig. 4: Application experiment results. Density refers to use the density strategy while Misses (0%, 1%, 5%) refer to use the strategy based on LLC misses with 0%, 1% and 5% thresholds in *hmem_advisor*. DDR refers to place everything in regular memory. MCDRAM* refers to allocate everything in fast memory and use DDR as a fall-back when MCDRAM is exhausted. Cache refers on configuring MCDRAM as Cache mode. autohbw/1m refers on using autohbw library with 1 Mbyte threshold.

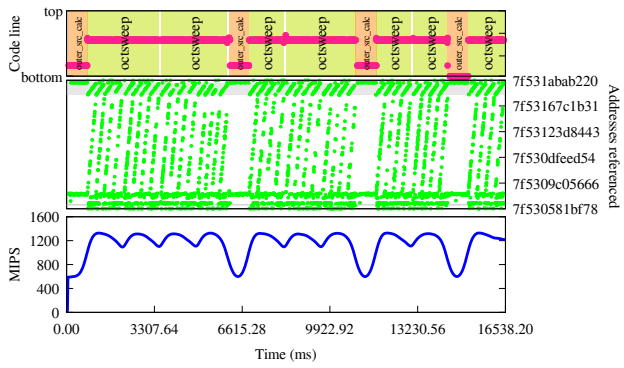


Fig. 5: Performance evolution for the main iteration of SNAP. The plots from top to bottom: source code (function) executed, the address space referenced and the performance achieved (in MIPS). The X-axis spans for the duration of the main iteration.

run and this misleads the framework because *hmem_advisor* considers data objects alive for the whole execution. To overcome this limitation, we have forced *hmem_advisor* to consider it has 512 Mbytes of MCDRAM per process but still limit *auto-hbwmalloc* to 256 Mbytes per process. With this approach, which simulates additional address for selecting data objects the difference shortens to 5.33% (still in favor of cache mode). Regarding SNAP, we have used the Folding technique to compare the behavior of the application when using `numactl -p 1` and the framework and we show the results for the latter in Figure 5. Notice that when the application executes the `outer_src_calc` routine (in orange, shown in the top plot), then the MIPS rate (in blue, shown in the bottom plot) drops but this does not happen when using `numactl -p 1` (not shown). The assembly code for this routine shows register spilling due to register pressure, and as register copies are stored in stack the execution benefits from running with `numactl -p 1` but not with the framework. There is no case where the *autohbw* library outperforms the rest but still improves the performance in several cases (HPCG, BT, CGPOP and miniFE) but also decreases the performance on Lulesh by 8%. This performance drop is explained by two facts. First, *autohbw* promotes non-critical data objects into fast memory which limits its impact. Second, we observe that allocations ranging from 1 to 2 Mbytes through memkind are more expensive than regular allocations (this issue is under investigation at the moment of writing this document). This second point is important because Lulesh allocates and deallocates memory during the main computation while the rest of the applications allocate memory during the initialization.

b) Memory-usage remarks: Regarding the memory used, we notice that all workloads tend to use more MCDRAM when they are allowed to, except for CGPOP and miniFE that only use 80 Mbytes per process (*circa* 5 Gbytes in total). This indicates that CGPOP and miniFE working sets could be larger and still fit in MCDRAM, and specifically for CGPOP, that additional performance could be achieved if some static variables were migrated into fast memory. We also notice some allocation differences when using the two relaxations of the 0/1 multiple knapsack problem. The usage of a threshold (`Misses(1%)` and `Misses(5%)` cases) reduces the amount of data promoted to fast memory, especially in Lulesh (4e), miniFE (4k), CGPOP (4n) and GTC-P (4w). Interestingly, the behavior in SNAP (4q) is the opposite, *i.e.* the `density`

approach allocates far less memory (64 Mbytes) in the 128 and 256 Mbyte cases. This occurs because the application allocates few small chunks of memory and one large (256 Mbytes) buffer, and the selection mechanism favors the placement of the small chunks in MCDRAM but then the large buffer does not fit.

c) MCDRAM-efficiency remarks: Finally, with respect to the $\Delta FOM/mbyte$ metric we identify different sweet-spots where the applications get the highest performance metric. On the one hand, Lulesh (4f), CGPOP (4o), SNAP (4r) and GTC-P (4x) maximize the use of the fast memory when using 32 Mbytes per process. On the other hand, miniFE (4l) and HPCG (4c) raise the sweet-spot to 128 and 256 Mbytes per process, respectively. In either case, when using more memory than the sweet-spot, the value of the metric decreases. This effect means that *hmem_advisor* selects the data objects by criticality and that moving non-critical objects into fast memory does not provide any benefit. Even though HPCG does not show this effect, one could foresee this to happen if additional MCDRAM is available.

D. General Discussion

Before concluding this section, we highlight some general conclusions extracted from these experiments. First, if the workload fits in MCDRAM, then it is worth using the `numactl -p 1` command because it places all (static, automatic and dynamic) data objects on MCDRAM because the framework can only place dynamic variables in MCDRAM and cache mode is not as performant as flat mode. If the workload does not fit, then it is crucial to ensure that critical data objects are stored in MCDRAM and this can be achieved through the framework or setting the MCDRAM in cache mode. Interestingly, cache mode and the framework complement each other regarding the applications they benefit most (miniFE, HPCG and GTC-P using the framework and Lulesh, SNAP, MAXW-DGTD using cache mode). We believe that this divergence is good to cover wider ranges of applications in terms of maximizing performance and to promote future research on this topic, but for now requires experimental testing to determine which approach is better. An additional conclusion is that most of the selected workloads do not require large amounts of fast memory to increase the performance, although there may be cases (in our experiments, HPCG) that will benefit from having more MCDRAM. Furthermore, our framework may help processor architects to dimension memory tiers on forthcoming processors.

Also, we remark that all this valuable information has been generated using relatively coarse-grain sampled information. Revisiting Table I, the reader will notice that the number of samples captured per application is relatively low (up to 38 K samples). This leads us to believe that a hybrid approach of minimal instrumentation and hardware-based sampling mechanisms is considerably helpful for exploring in-production executions as opposed to instruction-level instrumentation.

Finally, we want to address productivity in two directions besides the automatism provided by *auto-hbwmalloc*. First, we highlight that *hmem_advisor* identify a few allocation sites that greatly benefits from MCDRAM avoiding the user to need to explore every data allocation. For instance, the fastest cases of HPCG and miniFE reach their maximum performance by placing 2 and 3 data objects into fast memory, respectively, which would save coding time if users prefer to change the source code. Second, modern languages (such as C++) and runtimes (such as OpenMP) can hide data-object allocations (using templates/STL and private constructs, respectively) from a manual exploration. These allocations are captured by the tools used in our proposed framework, making sure that they are not ignored during the analysis.

V. Conclusions and Future Work

We have presented and evaluated a framework that promotes critical application data objects into the appropriate memory tier to shorten the time–to–solution automatically. The framework is applied to in-production binaries allowing end-users and developers to take advantage of HM systems without having to access the application source code. The usage of the framework increases the performance of many explored applications on an Intel Xeon Phi processor, surpassing the performance of using the MCDRAM as a LLC. The results we have shown may well be valuable for processor architects that need to dimension future HM systems based on current and future application demands.

As future work, it would be interesting to explore ways on predicting the application performance gains when moving some data objects into fast memory and one possible approach could be to replay the trace-file containing all the memory samples using a simulator. We also envision benefiting from the detailed memory access patterns obtained through the Folding technique when intelligently combining coarse-grain samples. First, Folding allows correlating the code regions and access patterns with other performance counters such as stalled cycles due to insufficient load and store buffers. This information might be useful to determine which objects prevent the processor from running at full speed due to unavailable hardware resources and promote them into the fastest memory layer to avoid the processor stalling. Second, it also leads us to identify regions of code with regular and irregular access patterns. This analysis would help placing irregularly accessed variables into the memory with shorter latency. Finally, the current framework places a whole data object in fast memory but it is possible that it does not fit or that not all the object is accessed uniformly, so it could be wise to place in fast memory only the critical portion. In this direction, we could take advantage of research that focus on data object partitioning [33], [34].

Acknowledgments

This work has been performed in the Intel-BSC Exascale Lab. Antonio J. Peña is cofinanced by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266. We would like to thank the Intel's DCG HEAT team for allowing us to access their computational resources. We also want to acknowledge this team, especially Larry Meadows and Jason Sewall, as well as Pardo Keppel for the productive discussions. We thank Raphaël Léger for allowing us to access the MAXW-DGTD application and its input.

REFERENCES

[1] A. Sodani, "Knights landing (KNL): 2nd generation Intel® Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–24.

[2] A. J. Peña and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 123–131.

[3] P. Cicotti and L. Carrington, "ADAMANT: tools to capture, analyze, and manage data movement," in *International Conference on Computational Science (ICCS)*, 2016, pp. 450–460.

[4] A. Rane and J. C. Browne, "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2012, pp. 147–156.

[5] "Intel Advisor XE," last accessed May 2017. [Online]. Available: https://software.intel.com/en-us/intel-advisor-xe

[6] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2016, vol. Volume 3B: System Programming Guide, Part 2, ch. 18.4.4.

[7] X. Liu and J. M. Mellor-Crummey, "A data-centric profiler for parallel programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, 2013, pp. 28:1–28:12.

[8] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P. Bremer, and B. Hamann, "Dissecting on-node memory access performance: A semantic approach," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 166–176.

[9] BSC, *Extrae user guide*, Barcelona Supercomputing Center, 2016, last accessed May 2017. [Online]. Available: https://tools.bsc.es/sites/default/files/documentation/extrae-3.2.1-user-guide.pdf

[10] "Intel VTune Amplifier," last accessed May 2017. [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe

[11] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, *memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies*, Mar 2015. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1245908

[12] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 1992, pp. 1–12.

[13] K. Beyls and E. D'Hollander, "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62–71, 2009.

[14] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "PerfExpert: An easy-to-use performance diagnosis tool for HPC applications," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 190–200.

[16] V. Subotic, R. Ferrer, J. Sancho, J. Labarta, and M. Valero, "Quantifying the potential task-based dataflow parallelism in MPI applications," *Euro-Par, Parallel Processing*, pp. 39–51, 2011.

[17] A. J. Peña and P. Balaji, "A framework for tracking memory accesses in scientific applications," in *43rd International Conference on Parallel Processing Workshops (ICCPW)*. IEEE, 2014, pp. 235–244.

[18] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 175–183.

[19] M. Itzkowitz *et al.*, "Memory profiling using hardware counters," in *ACM/IEEE conference on Supercomputing (SC)*, 2003, p. 17.

[20] H. Servat, G. Llort, J. González, J. Giménez, and J. Labarta, "Low-overhead detection of memory access patterns and their time evolution," in *European Conference on Parallel Processing*. Springer, 2015, pp. 57–69.

[21] BSC, "Paraver web-site," Barcelona Supercomputing Center, 2016, last accessed May 2017. [Online]. Available: http://tools.bsc.es/paraver

[22] Free Software Foundation, "GNU Binutils," http://www.gnu.org/software/binutils Last accessed May, 2017.

[23] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.

[24] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *IJHPCA*, vol. 30, no. 1, pp. 3–10, 2016, URL: http://www.hpcg-benchmark.org/downloads/hpcg-3.0.tar.gz, SHA1/10: 39e1b7e45e, modifications cover slides 4-7 from http://www.hpcg-benchmark.org/downloads/sc14/HPCG_on_the_K_computer.pdf.

[25] "Hydrodynamics challenge problem," Lawerence Livermore National Laboratory, Tech. Rep., last accessed May 2107. URL: https://codesign.llnl.gov/lulesh/lulesh2.0.3.tgz, SHA1/10: 1ff51421bf. [Online]. Available: https://codesign.llnl.gov/pdfs/LULESH2.0_Changes.pdf

[26] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, 1991, pp. 158–165, URL: https://www.nas.nasa.gov/publications/npb.html, SHA1/10: 70e727ff39.

[27] "Mantevo benchmark suite," last accessed May 2017. URL: https://asc.llnl.gov/CORAL-benchmarks/Throughput/MiniFE_ref_2.0-rc3.tar.gz, SHA1/10: 64e79502d9. [Online]. Available: https://mantevo.org/download

[28] A. Stone, J. Dennis, and M. M. Strout, "The CGPOP miniapp, version 1.0," Colorado State University, Tech. Rep. Technical Report CS-11-103, 2011, URL: https://github.com/xiehuc/cgpop, SHA1/10: 5e7deea26a.

[29] "SNAP: SN (discrete ordinates) application proxy," last accessed May 2017. URL: https://github.com/losalamos/SNAP.git, SHA1/10: b25fd4197c. [Online]. Available: https://github.com/lanl/SNAP

[30] R. Léger, D. Alvarez Mallon, A. Duran, and S. Lanteri, "Adapting a finite-element type solver for bioelectromagnetics to the DEEP-ER platform," vol. Parallel Computing: On the Road to Exascale, no. 27. Edinburgh, United Kingdom: IOS Press, 2016, p. 850, URL: N/A - provided by user directly, SHA1/10: 1ff51421bf. [Online]. Available: https://hal.inria.fr/hal-01243708

[31] "Gyrokinetic Toroidal Code - Princeton," last accessed May 2017. URL: http://www.nersc.gov/research-and-development/apex/apex-benchmarks/gtc-p, SHA1/10: 7b28264821.

[32] C. Durochat, S. Lanteri, and R. Léger, "A non-conforming multi-element DGTD method for the simulation of human exposure to electromagnetic waves," *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 27, no. 3, pp. 614–625, 2014. [Online]. Available: http://dx.doi.org/10.1002/jnm.1943

[33] A. J. Peña and P. Balaji, "A data-oriented profiler to assist in data partitioning and distribution for heterogeneous memory in HPC," *Parallel Computing*, vol. 51, pp. 46–55, 2016.

[34] P. Roy and X. Liu, "StructSlim: a lightweight profiler to guide structure splitting," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO*, 2016, pp. 36–46.