ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems

Shasha Wen
College of William and Mary
swen@email.wm.edu

Lucy Cherkasova
ARM Research
ludmila.cherkasova@arm.com

Felix Xiaozhu Lin
Purdue ECE
xzl@purdue.edu

Xu Liu
College of William and Mary
xl10@cs.wm.edu

ABSTRACT

New memory technologies, such as non-volatile memory and stacked memory, have reformed the memory hierarchies in modern and emerging computer architectures. It becomes common to see memories of different types integrated into the same system, as known as heterogeneous memory. Typically, a heterogeneous memory system consists of a small fast component and a large slow component. This encourages new style of data processing and exposes developers with a new problem: given two memory types, how shall we redesign applications to benefit from this memory arrangement and decide on the efficient data placement? Existing methods perform detailed memory access pattern analysis to guide data placement. However, these methods are heavyweight and ignore the interactions between software and hardware.

To address these issues, we develop ProfDP, a lightweight profiler that employs differential data-centric analysis to provide intuitive guidance for data placement in heterogeneous memory. Evaluated with a number of parallel benchmarks running on a state-of-the-art emulator and a real machine with heterogeneous memory, we show that ProfDP is able to guide nearly-optimal data placement to maximize performance with minimum programming efforts.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Runtime environments**; *Application specific development environments*;

KEYWORDS

Sampling, profiling, data placement, heterogeneous memory

ACM Reference Format:
Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. 2018. ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205320

1 INTRODUCTION

In recent years, new memory technologies beyond traditional DRAM have been developed in both industry and academia, such as 3D stack memory [33], non-volatile memory (NVRAM) [25], scratchpad memory [5] and among others. Systems that employ multiple memory technologies are heterogeneous memory systems. Typically, a heterogeneous memory system consists of *a fast memory component* and *a slow memory component*. The fast component has either lower latency or higher bandwidth or both compared to the slow component. However, the capacity of the fast component is far smaller than the slow component. For example, Intel Knights Landing (KNL) has on/off-package memories. The 16GB on-package memory (HBM) has 5× the bandwidth of 384GB off-package memory (DRAM) [40]. Thus, the HBM is abstracted as a fast memory component, while the DRAM is the slow memory component.

Heterogeneous memory usually provides flexible data placement. One can decide data placement via software. For example, KNL can be configured in flat mode so programmers can control the data placement manually using `libnuma` [4] or `memkind` [7]. Moreover, NVML [15] provides seven separate libraries that can benefit memory allocation at different points, such as `libpmemblk`, which may be helpful for applications that keep a cache of fixed-size objects. However, the manual data placement adds extra burdens to programmers. Programmers need to decide which memory segments should be placed in the fast memory with the limited capacity for the best performance. However, given a real program that has complex memory patterns, manual data placement is tedious and difficult to achieve the optimal performance.

To address this issue, tools and technologies are proposed to guide programmers to place data objects. State-of-the-art tools [18, 39] guide data placement according to the data size, number of accesses, or memory access patterns. However, existing data placement profilers have two potential drawbacks. First, these tools collect the behavior of each data object, e.g., memory access patterns in a pure software way. They ignore the hardware details, such as multi-level caches and pre-fetchers, which may significantly impact the behaviors in the memory hierarchy. Second, collecting memory access patterns often requires heavyweight memory instrumentation, which typically incurs more than 100× runtime overhead [47]. This high overhead prevents these tools from applying to production code bases.

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205320

To overcome these limitations, we develop ProfDP, a novel profiler that guides data placement in heterogeneous memory. ProfDP makes the following four contributions.

- Instead of collecting memory access patterns in software, ProfDP utilizes hardware performance counters to measure a binary's real behavior in the memory hierarchy with low overhead.
- ProfDP employs a novel differential analysis to quantify the sensitivity of each data object to heterogeneous memory with different latency and bandwidth.
- ProfDP associates all the analysis with data objects identified by their allocation contexts or names, as known as data-centric analysis and presents the data-centric analysis in a user-friendly interface.
- Guided by ProfDP, we categorize a number of parallel programs and evaluate the data placement in a state-of-the-art emulator and a real architecture with heterogeneous memory. Our experiments show that ProfDP is able to achieve nearly optimal performance by placing a minimum amount of data objects into the limited fast memory.

The rest of this paper is organized as follows. Section 2 introduces some background knowledge and motivates ProfDP with an example. Section 3 describes the methodology ProfDP utilizes to guide data placement. Section 4 elaborates the implementation of ProfDP. Section 5 shows the experiments to evaluate ProfDP. Section 6 describes several case studies. Section 7 discusses some limitations of ProfDP. Section 8 distinguishes ProfDP from existing approaches. Section 9 presents some conclusions.

2 BACKGROUND AND MOTIVATION

Emerging byte-addressable, non-volatile memory (NVRAM) technologies [25] are an alternative to disk for persistence and provide accessing latency with in the order of magnitude of DRAM. In addition, 3D stack memory [33], as a complement to DRAM, provide lower accessing latency or higher bandwidth or both. From the forward-looking, the new architecture suggests to build systems with heterogeneous memory systems, i.e., including both DRAM and NVRAM or including both DRAM and 3D stack memory. This new system design leads to a new problem of the efficient data placement: given the two memory types, how shall we design new applications to benefit from this memory arrangement and decide on the efficient data placement?

It is usually ideal to place all the data in the memory with lower latency or higher bandwidth, as known as *fast memory*. Throughout this paper, we define the optimal performance as the performance obtained when placing all data objects into the fast memory. However, compared to *slow memory*, fast memory has a much smaller capacity. Placing all the data in fast memory is often impossible and undesirable, especially when the machine is time-shared by multiple users. Thus, the ideal case is to put an as small amount of data as possible into the fast memory to achieve the nearly-optimal performance. However, a parallel program typically has many data objects. Different data objects may have different sensitivities to heterogeneous memory.

As an example, we use the code of matrix multiplication in the form of $C(n,n) = A(n,n) \times B(n,n)$. The code sweeps matrix A in the row major and matrix B in the column major. We evaluate this code example on Intel Knights Landing (KNL), which, as mentioned in the previous section, employs high-bandwidth memory (HBM) as fast memory and DRAM as slow memory. By default, all matrices are allocated in DRAM. We configure the execution in four ways: ① placing only matrix A in HBM, ② placing only matrix B in HBM, ③ placing all matrices A, B, and C in HBM, and ④ placing no matrix in HBM. The configuration ④ serves as the baseline of the performance study.

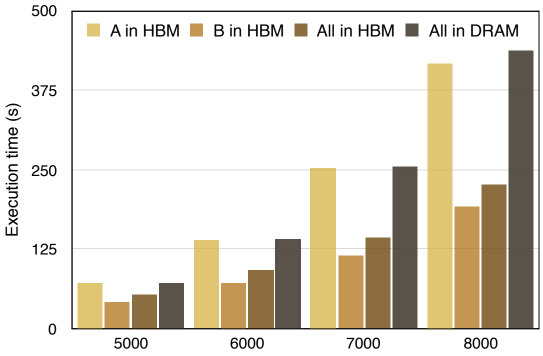


Figure 1: Execution time when putting different arrays of matrix multiplication into HBM. Axis X shows the tuning of array size; axis Y shows the execution time.

Figure 1 compares the execution time among the four execution configurations according to different matrix sizes n. The experiments show that putting A into HBM does not show any speedup. In contrast, placing B into HBM leads to faster execution. It is because matrix A shows better spatial locality over B with a row-major access pattern. The large volume of cache misses incurred by accesses to B can benefit from fast memory. Thus, different data objects show different sensitivities to HBM. Moreover, placing all matrices in HBM show nearly-optimal performance (slightly slower than placing B only). It is because placing all matrices in HBM does not utilize DRAM, which provides 20% bandwidth of HBM. However, it is usually difficult to identify the optimal placement strategy. In this paper, we use the case "placing all data in fast memory" as the experimental optimal data placement strategy to evaluate ProfDP-guided data placement.

From this example, we can see that accessing frequency and data size simply do not provide useful insights for data placement, as both matrices A and B share the same size and number of accesses. It is the runtime behavior in the memory hierarchy that influences the benefit from placing different data objects in fast memory. The state-of-the-art work [18, 36, 39] extracts memory accessing patterns with heavyweight memory instrumentation to assess such behavior. However, this approach incurs high overhead and ignores hardware features, such as pre-fetchers and multi-level caches. In contrast, ProfDP guides data placement based on the measurement of binary execution on real hardware with low overhead.

3 METHODOLOGY

In this section, we introduce the features used to describe each data object, including how to quantify these features and how these features lead to ProfDP's data placement decisions.

3.1 Features to Prioritize Data Placement

In heterogeneous memory system, the fast memory component either shows lower accessing cost or higher bandwidth capacities. Thus, it is important to understand the program under optimization is memory bound or not. ProfDP uses the average latency per instruction metric [30] to exclude applications that are not memory bound in study. In the next step, to decide which data objects should be allocated in fast or slow memory, it is necessary to understand the performance features of each data object.

The reason we place data A in fast instead of slow memory is that A will benefit more from the lower memory latency (or higher bandwidth capacity). The more sensitive one data object is to latency (or bandwidth), the more benefit it will get when putting in fast memory. When both data A and B are sensitive to the latency (or bandwidth), the reason why we put data A instead of B into fast memory is that putting A will speedup the program more. This tells the significance of A and B. If A is more significant than B, allocating A in fast memory gives more performance improvement. Moreover, the size of a data object is another feature we need to consider. Placing smaller-sized data objects gives less pressure to the limited fast memory, especially when multiple programs time share the system and compete the fast memory.

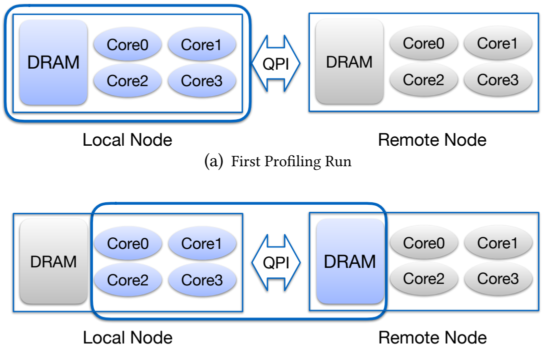
We organize the features we need to know for each data object as follows:

- **Latency Sensitivity** quantifies how sensitive one data object is to memory latency. The data objects that show higher sensitivity to latency are suggested to be put into fast memory that has lower latency than slow memory.
- **Bandwidth Sensitivity** quantifies how sensitive one data object is to memory bandwidth. The data objects with higher bandwidth sensitivity are suggested to be put into fast memory that has higher bandwidth capacity over slow memory.
- **Importance** describes how significant each data object is throughout the whole program execution. Significant data objects should stay in fast memory to speedup their frequent accesses.
- **Size** quantifies whether a data object is resource friendly. Placing small sized data objects leaves space in fast memory for other data objects.

3.2 Metrics to Quantify Features

In this section, we describe the methods and metrics to quantify each data's **latency sensitivity**, **bandwidth sensitivity**, **importance**, and **size**.

To measure latency and bandwidth sensitivities, we use *differential analysis*, which is effective in identifying performance bottlenecks [14, 31, 34]. The basic idea of differential analysis is that it runs a program twice with two different configurations and generates two profiles. For example, one configuration doubles the number of cores for program execution over the other. We, then, have an expectation in differentiating the two profiles. For example, the expectation for linear scaling is that the code execution time should be halved with doubling the cores (strong scaling). Thus, if some code segments do not meet the expectation, e.g., the execution time is longer than half when doubling the cores, we can interpret and quantify the performance problem, such as scaling loss. We apply differential analysis to quantify latency and bandwidth sensitivities for each data object.



(a) First Profiling Run

(b) Second Profiling Run

Figure 2: Cores and memory configuration for the two profiling runs in Differential Analysis .

Quantifying latency sensitivity: Our differential analysis for latency sensitivity is based on the following expectation:

Expectation 1: A data object is latency insensitive, if the average latency of its accesses does not change when the program, with the same input and parallelism, runs on a machine with higher memory access latency.

We run the program twice in a non-uniform memory access (NUMA) system. The first time, we run with all cores and memory from the same socket as shown in Figure 2 (a). The second time, we run with all cores and memory from different sockets as shown in Figure 2 (b). So in the second run, all memory accesses will go to remote memory with longer latency than in the first run. If the average latency to access a data object does not follow Expectation 1, this data object is latency sensitive. We compute the average latency of accessing data object A as follows:

$$C(A) = \frac{\sum_{s=1}^{N} Latency(s, A)}{N} \tag{1}$$

$Latency(s, A)$ denotes the latency in CPU cycles of accessing data A performed by a memory instruction s. In the first profiling run, we get the average access latency of A in $C_{local}(A)$, and in the second profiling run, we get the average access latency of A in $C_{remote}(A)$. We then quantify the **latency sensitivity** of data object A as:

$$LS(A) = \frac{C_{remote}(A) - C_{local}(A)}{C_{local}(A)} \tag{2}$$

Quantify bandwidth sensitivity: To evaluate the bandwidth sensitivity of a data object, we can apply the differential analysis for two profiling runs with various bandwidth capacities allocated for the program and quantify the difference of bandwidth consumption. Intel RDT [16] package and Perf [22] supports software to monitor bandwidth usage on recent Intel processors. However, the granularity of these existing monitoring tools is on the process level, rather than data objects. Moreover, unlike latency, bandwidth is an aggregate metric in a time window, which does not provide a quantitative value for individual memory accesses. Thus, we need to design a software workaround to avoid using bandwidth consumption as the expectation metric.

According to Little's law [29], the available bandwidth can be expressed as Equation 3:

$$bandwidth = transfer\_size \times \frac{MLP}{Latency} \tag{3}$$

MLP is short for Memory-Level Parallelism, which is the average number of concurrent memory requests on the fly. *Latency* is averaged across all memory accesses. $transfer\_size$ is the bytes of each memory request.

Eklov et al. [19] pointed that each core has a local bandwidth. Because parallelizing code on more cores increases MLP, the aggregate local bandwidth is proportional to the number of cores until reaching the global bandwidth of the system. In modern processors, a single thread can saturate local bandwidth but not global bandwidth. Thus, from Little's law, we know that bandwidth is proportional to MLP, unless latency changes. If latency increases, bandwidth cannot achieve the expected value. With this knowledge, we convert the expectation for bandwidth differentiation to latency differentiation.

Expectation 2: A data object is bandwidth insensitive, if the average latency of its memory accesses does not change when the program, with the same input, scales from one (low MLP) to more (high MLP) cores in the same socket.

It is worth noting that we make an assumption that running a parallel program with more cores increases MLP. This is mostly true for data parallel applications. However, this may not be true for some task parallel programs; they are even not memory bound. In such cases, ProfDP will not recommend the optimization with fast memory because the bottlenecks are not in memory.

To apply Expectation 2, we run program twice: the first run uses only one core, which does not saturate the global bandwidth of the system, while the second run uses all the cores in the same socket, which, with highest probability, may saturate global bandwidth. In the first run, we measure the average access latency $C_{one}(A)$ for data object A. In the second time, we measure the average access latency $C_{multi}(A)$ for the same data object. We further quantify the **bandwidth sensitivity** of data object A as follows:

$$BS(A) = \frac{C_{multi}(A) - C_{one}(A)}{C_{one}(A)} \tag{4}$$

Quantify importance and size: **Importance** quantifies the significance of a data object. One can use two metrics to define importance of a data object: `total_num_access`, which is the total number of memory accesses to this data object, and `total_cost`, which is the total access cost in latency to access this data object. As accesses have various latency, `total_num_access` does not reflect the performance of data significance. Thus, we apply the `total_cost` metric.

The **Importance** of data object A is defined as the ratio of aggregate latency incurred to access A to the aggregate latency throughout the entire program:

$$I(A) = \frac{\sum_{s=1}^{N} Latency(s, A)}{\sum_{s=1}^{N} Latency(s, ALL)} \tag{5}$$

$\sum_{s=1}^{N} Latency(s, ALL)$ denotes the aggregate latency of all the accesses in the program and $\sum_{s=1}^{N} Latency(s, A)$ is the aggregate latency of all the accesses to A.

Size quantifies memory occupation of each data object, which can be either an absolute number of bytes allocated in memory or a percentage value over the total memory bytes used in the program. We record both but just use percentage value to calculate moving factor in the next section.

3.3 Moving Factor for Placement Decisions

After quantifying the **latency sensitivity**, **bandwidth sensitivity**, **importance**, and **size** for each data object, the next step is to make data placement decisions based on these metrics. We define *Moving Factor*, which is derived from the four metrics. Data objects with higher *Moving Factor* values are of higher priority to be placed in fast memory.

For each data object A, we calculate the *Moving Factor (MF)* of it using the following formulas:

$$MF(A) = \frac{S(A) \times I(A)}{Size(A)} \tag{6}$$

$$MF(A) = \begin{cases} MF_{LS}(A) : S(A) = LS(A) \\ MF_{BS}(A) : S(A) = BS(A) \end{cases} \tag{7}$$

The sensitivity $S(A)$ hints the benefit we can get for data object A when moving it from slow to fast memory. In a heterogenous memory where the fast memory shows lower latency, $S(A)$ will be assigned with the value of $LS(A)$ while in a system where the fast memory is produced with higher bandwidth, $S(A)$ will be replaced with $BS(A)$. We use term MF_{LS} and MF_{BS} to represent the *Moving Factor* calculated with LS and BS respectively as described in Equation (7).

As the importance $I(A)$ quantifies the significance of accessing a data object throughout the entire program execution, $S(A) \times I(A)$ implies the benefit we can get for the entire execution of placing A in fast memory. The result value is then divided by the size of A, which computes $MF(A)$ as the benefit per byte.

ProfDP computes and ranks MFs for all the monitored data objects and prioritizes them accordingly. The data object with the highest MF value is the top candidate to be placed in fast memory.

4 IMPLEMENTATION DETAILS

In this section, we describe the implementation of ProfDP. We first introduce the performance monitoring units (PMUs) available in modern CPU architectures that provide necessary runtime information for ProfDP. We then describe the implementation of ProfDP's online data-centric analysis that associates metrics with program data objects. Finally, we elaborate on the implementation of ProfDP's offline differential analysis that provides a user-friendly view of exploring the analysis results.

4.1 PMU-supported Address Sampling

Modern x86 CPU architectures employ powerful PMUs to measure program execution. Unlike traditional performance counters, PMUs can periodically select a memory access to monitor and record its execution behavior through the pipeline. With the support of PMUs, one can sample memory accesses to monitor program behaviors in the memory hierarchy at low cost. Such techniques include instruction-based sampling (IBM) [17] available in AMD Opteron processors (family 10h and successors) and precise event-based sampling (PEBS) [1] which is available in Intel processors starting from SandyBridge microarchitecture.

For each sampled memory access, Both IBS and PEBS can capture necessary information for ProfDP, such as (1) the effective address (EA) touched in memory and (2) the cost in terms of data access latency (LAT) in CPU cycles. ProfDP is able to leverage both IBS and PEBS to perform measurement and analysis: ProfDP uses EA for data-centric analysis (Section 4.2) and LAT for differential analysis (Section 4.3).

4.2 Online Data-centric Analysis

ProfDP monitors the allocation and lifetime of static and heap data objects.

Static data. Static data are allocated statically or globally in a load module (executable or dynamic library). In the symbol table of each load module, each static data object has a tuple of name and offset from the beginning of the load module. The memory for static variables is allocated when the enclosing load module is loaded into memory and reclaimed when the load module is unloaded. ProfDP tracks the loading and unloading of each load module. When a load module is loaded in the execution address space, ProfDP reads its symbol table to extract information about the memory ranges for all of its static variables. These memory ranges are inserted into a map for future use. When a load module is unloaded from the execution space, all the data objects associated with this module are removed from the map.

Heap data. Heap data are allocated dynamically by one of the malloc family of functions (malloc, calloc, realloc). To monitor a heap data, ProfDP overloads memory allocation and free functions. At each monitored allocation, ProfDP enters the allocated memory range into a map. At each monitored free, the profiler deletes the reclaimed memory range from the map. Moreover ProfDP determines the call path of the allocation site with a lightweight on-the-fly binary analysis technique [43].

For stack data, which are allocated locally, ProfDP does not directly monitor them. One can convert the stack data into static or heap data to enable ProfDP to monitor them. ProfDP then uses the effective address collected along with each address sample and checks the map to identify which data object encloses this address. If the data object is found, ProfDP attributes the sample to it along with all the associated metrics. To scale the analysis for parallel programs, ProfDP produces a data-centric profile for each thread.

Machines	Intel-SandyBridge	Intel-Broadwell	Intel KNL
Processor	Xeon E5-4650	Xeon E7-4830v4	Xeon Phi 7210
Frequency	2.7GHz	2.0GHz	1.3GHz
SMT×Cores×Sockets	2×8×4	2×14×4	4×64×1
L1/L2/L3 Cache	32KB/256KB/20MB	32KB/256KB/35MB	32K/1024K/–
Memory	256GB DRAM	256GB DRAM	16GB HBM 128GB DRAM

Table 1: Machine configurations.

4.3 Offline Differential Analysis

ProfDP's differential analysis consists of three components: aggregate profiles from all threads of each individual execution, differentiating profiles from different executions, and visualizing the analysis.

Profile aggregation. Aggregating all the profiles and computing average metrics across threads or/and processes of a single execution are important to show the overall program performance, not biased by individual threads/processes. Merging multiple data-centric profiles requires ProfDP aggregating metrics associated with the same data objects. ProfDP defines two data objects are the same if (1) they are both static data with the same name, or (2) they are both heap data with the same allocation context. The execution time of profile aggregation grows linearly with the number of threads and processes used by the monitored program because each thread/process owns one profile. ProfDP leverages an existing reduction tree technique [42] to parallelize the merging process. In our experiments, ProfDP requires less than 10 seconds to produce an aggregate profile.

Profile differentiation. Profile differentiation is similar to profile aggregation. It takes the two aggregated profiles collected from two different executions as input. It first identifies the same data objects using the same method of profile aggregation. It then performs the differentiation on the metrics of the same data objects to derive the metrics described in Section 3. It is worth noting that, ProfDP uses the sampling data to estimate the average latency per access, which is of high statistical accuracy if ProfDP samples more than 20 accesses to a monitored data object [41].

Profile visualization. ProfDP employs a graphic user interface inherited from HPCToolkit [2]. The interface presents differential data-centric profiles and associates the profiles with source codes easily. Figure 5 is an example snapshot of ProfDP's interface, which we describe in detail in Section 6.1.

5 EXPERIMENTS

We evaluate ProfDP with three real machines with the configurations as shown in Table 1. ProfDP collects profiles of programs running on the SandyBridge machine and guides data placement in memory systems with two kinds of heterogeneities.

Latency heterogeneity. We use Quartz [44] to emulate memory system with DRAM and NVRAM. Quartz works on a multi-socket DRAM machine. It runs programs on a single socket and uses the memory attached to another socket to emulate NVRAM by epochal inserting delays. Quartz applies `X86 rdtscp` to read timestamp and spin wait the program until waiting time reaches the delay. Quartz calculates delays based on the difference between the NVRAM latency and the real access cost to the node used to emulate NVRAM. The NVRAM latency is a parameter that can be set by users. We install and run Quartz on a Broadwell machine. In the experiments, the DRAM latency is 120ns and the NVRAM latency varies from 400ns, 600ns, 800ns to 1000ns.

Bandwidth heterogeneity. Intel Knights Landing (KNL) supports high-bandwidth memory (HBM), which has five times bandwidth over DRAM. HBM can be configured in *flat mode*, *cache mode*, or partially flat and partially cache mode. The flat mode uses the entire HBM as addressable memory while the cache mode uses the entire HBM as a cache. In our experiments, we configure HBM in flat mode.

The benchmark suites used in our experiments are described as follows:

- Coral [26], developed by Lawrence Livermore National Laboratory (LLNL) includes scalable science benchmarks, throughput and big data processing benchmarks, which show high parallelism and representative memory patterns.
- Rodinia [11], a benchmark suite for parallel computing on general-purpose CPU architectures with accelerators. We use its OpenMP version in our evaluation.
- NAS Parallel Benchmarks (NPB) [35], which includes a small set of programs derived from computational fluid dynamics applications. These programs are designed with various memory access patterns. We run its C version [38] parallelized with OpenMP.
- PARSEC [6] is a parallel benchmark suite, including programs from many different areas such as computer vision, video encoding, and image processing with various access patterns. We use its pthread version for evaluation.

ProfDP profiles programs with the following configuration: all the benchmarks are compiled with `gcc 4.8.5 -O3` on SandyBridge, Broadwell, and KNL machines. In the rest of this section, we first evaluate the overhead of ProfDP and then use ProfDP to categorize benchmarks with different sensitivities.

5.1 Profiling Overhead

We profile all the benchmarks on the Intel SandyBridge machine. When profiling the latency sensitivity, we run each benchmark with all the eight cores from one socket with SMT disabled. When profiling the bandwidth sensitivity, we first profile with one core and then profile with all eight cores.

For each profiling execution, ProfDP monitors one memory access for every one million memory accesses. Figure 3 shows the runtime profiling overhead of ProfDP when running with eight cores in one socket. The GeoMean overhead of all the benchmarks is 1.1%. Most of the benchmarks show overhead less than 6%. One exception in our experiment is Rodinia `cfd`, which has the highest overhead—18%. This high overhead is due to the frequent memory allocation by c++ **new[]** operator, which ProfDP needs to overload for the data-centric analysis. For all of our experiments, ProfDP incurs <5MB memory overhead per thread.

5.2 Benchmark Classification

One step before optimizing code is to understand whether the program can benefit from using fast memory. ProfDP is able to categorize benchmarks and pinpoint the potential benchmarks that can benefit data placement optimization in heterogeneous memory. Given that the fast memory provides either lower latency or higher bandwidth, to identify whether an application is a good candidate for optimization in heterogeneous memory, we quantify its latency and bandwidth sensitivities for the entire execution. Figure 4 shows the distribution of benchmarks selected from the aforementioned benchmark suites. The horizontal axis denotes the **latency sensitivity** of the entire program execution, while the vertical axis represents the **bandwidth sensitivity** of the entire program execution.

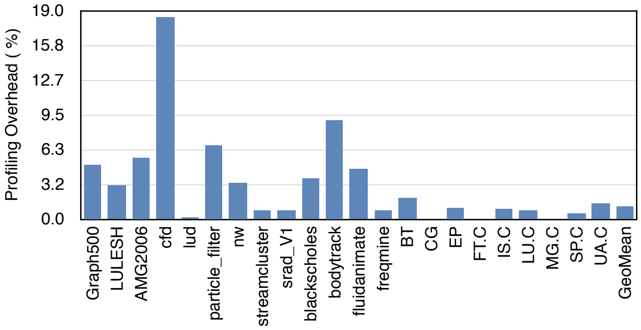


Figure 3: ProfDP's profiling overhead.

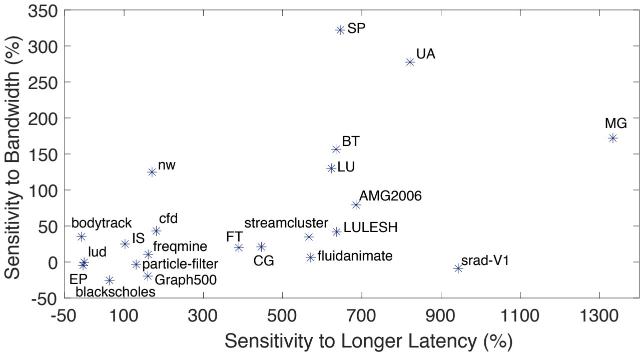


Figure 4: Benchmarks classification based on its sensitivity to latency and bandwidth.

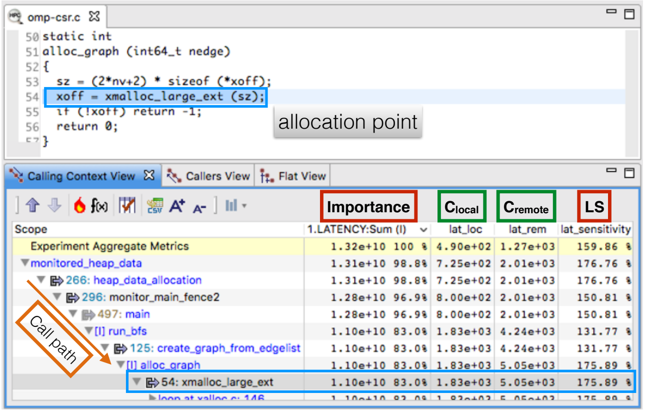
The higher the sensitivity is, the more likely it will benefit from fast memory. Take NPB `MG` benchmark for an example, it's **bandwidth sensitivity** is 172%. This indicates that `MG` can benefit from higher memory bandwidth in fast memory component. `MG`'s **latency sensitivity** is as high as 1333%, which means that `MG` can obtain significant benefit from lower memory latency in fast memory component. Thus, we are able to reason that `MG` is worth of optimization in systems with heterogeneous memory that varies in either latency or bandwidth or both.

6 CASE STUDIES

In this section, we select several benchmarks with either high latency sensitivity or high bandwidth sensitivity or both to further study the program internals with ProfDP. ProfDP is able to rank

Program	Data Objects	Data Allocation Location	Features				Moving Factor	
			%LS	%BS	%Importance	%Size	MF_{LS}	MF_{BS}
Graph500	xoff	omp-csr.c:54	176	0	83	2.1	70	0
	bfs_tree	graph500.c:191	334	0	6.7	1.1	20	0
	edges	make_graph.c:53	269	0	7.1	33.6	0.6	0
LULESH	vnew	LULESH.C:2196	1452	35.5	3.6	1.24	42	1.0
	y	LULESH.C:2159	3040	33.5	1.2	1.24	29	0.3
	x	LULESH.C:2158	1139	0	2.5	1.24	23	0
	nodelist	LULESH.C:2222	639	48.5	17.6	5.2	22	1.6
	delx_xi	LULESH.C:2207	1198	54.7	2.2	1.24	21	1.0
	vdov	LULESH.C:2191	820	56.3	1.5	1.24	9.9	0.7
	f_elem	LULESH.C:2211	636	68.2	39.5	29.6	8.5	0.9
AMG2006	P_diag_j_new	par_interp.c:1241	6332	12.6	1.9	1.1	109	0.2
	diag_j	IJMatrix_parcsr.c:1803	3234	124	3.6	1.9	61	2.4
	diag_data	IJMatrix_parcsr.c:1804	788	789	12.4	3.7	26	26
	RAP_diag_data	par_rap.c:1385	626	77.8	27.3	9.1	19	19
	RAP_diag_j	par_rap.c:1386	637	90.1	12.4	4.5	18	18
	P_diag_data_new	par_interp.c:1242	464	114	2.6	2.3	5.2	5.3
NW	input_itemsets	needle.cpp:114	180	160	69	33.1	3.7	3.3
	referrence	needle.cpp:113	177	161	30.3	33.1	1.6	1.5
StreamCluster	block	streamcluster_omp.cpp:1113	618	23	96.6	87	6.9	0.3
CG	a	cg.c:63	628	38.8	54.5	32	11	0.7
	colidx	cg.c:54	885	0	14.1	16	7.8	0
MG	u	mg.c:82	1971	115	48	33	29	1.7
	r	mg.c:84	730	211	39	33	8.6	2.5

Table 2: Data objects in different benchmarks with high priority for placement in fast memory.



```
static int
alloc_graph (int64_t nedge)
{
  sz = (2*nv+2) * sizeof (*xoff);
  xoff = xmalloc_large_ext (sz);
  if (!xoff) return -1;
  return 0;
}
```

```
Scope                                    1.LATENCY:Sum (I)  lat_loc   lat_rem   lat_sensitivity
Experiment Aggregate Metrics             1.32e+10 100  %    4.90e+02  1.27e+03  159.86 %
monitored_heap_data                      1.31e+10 98.8%     7.25e+02  2.01e+03  176.76 %
 266: heap_data_allocation               1.31e+10 98.8%     7.25e+02  2.01e+03  176.76 %
  296: monitor_main_fence2               1.28e+10 96.9%     8.00e+02  2.01e+03  150.81 %
   497: main                             1.28e+10 96.9%     8.00e+02  2.01e+03  150.81 %
    [I] run_bfs                          1.10e+10 83.0%     1.83e+03  4.24e+03  131.77 %
     125: create_graph_from_edgelist     1.10e+10 83.0%     1.83e+03  4.24e+03  131.77 %
      [I] alloc_graph                    1.10e+10 83.0%     1.83e+03  5.05e+03  175.89 %
       54: xmalloc_large_ext             1.10e+10 83.0%     1.83e+03  5.05e+03  175.89 %
        loop at xalloc.c: 146            1.10e+10 83.0%     1.83e+03  5.05e+03  175.89 %
```

Figure 5: ProfDP's GUI presents the profile of Graph500.

all the data objects according to the Moving Factor metric to guide data placement in its user interface. We then evaluate placing top data objects in fast memory and quantify the placement impact to the entire execution. As mentioned in previous section, we evaluate ProfDP for each program in two heterogeneous memory systems. One system uses Quartz emulator where DRAM is used as fast memory with lower access latency while NVRAM is emulated to be slow memory with higher access latency. We experiment with various latency difference between DRAM and NVRAM to evaluate the effectiveness of ProfDP. The other system is Intel KNL, where HBM is the fast memory with 5× bandwidth of the slow memory, DRAM. ProfDP uses metric MF_{LS} to guide data placement in Quartz while MF_{BS} to guide data placement in KNL. We run all the benchmarks compiled with gcc 4.8.5 -O3 on all available cores in one socket (14 threads in Quartz and 256 threads in KNL).

6.1 Graph500 Breadth First Search

Graph500 [20] is designed for developing comprehensive benchmarks to address application kernels in graph-related areas. We evaluate its OpenMP version with the total graph datasets to be around 12GB. In our experiment, we run one iteration of the Breadth First Search (BFS).

Figure 5 shows the GUI of ProfDP. The top panel shows the source code, the bottom left panel shows the data objects identified by their allocation points with full call paths, and the bottom right panel shows various metrics. In this example snapshot, the highlighted blue box shows the allocation of data object $xoff$ and its associated metrics. Array $xoff$ is used to index the graph. Its Importance metric, as known as the contribution to the total memory access cost, is 83%. Its average local memory access cost $C_{local}(xoff)$ is 1830 cycles, while the average remote memory access cost $C_{remote}(xoff)$ is 5050 cycles, resulting in latency sensitivity — $LS(xoff)$ to be 176%. Its moving factor MF_{LS} is 40, which is ranked as the top candidate for placement. Table 2 shows more metrics for $xoff$ and other significant data objects in BFS.

Latency sensitivity. Figure 6 (a) shows the execution speedups running on Quartz with setting various NVRAM latency. The based line to compute the speedups is placing all data objects in NVRAM. If we set the NVRAM latency to be 600ns, we achieve 1.47× speedup when placing all data in DRAM. If we put array $xoff$ only in DRAM, we are able to get as high as 1.15× speedup. If we place the top 3 data objects into DRAM, we get a 1.34× speedup. The total size of the three data objects is 37% of the total memory usage in bytes. The speedup keeps increasing with a longer NVRAM latency. Thus, ProfDP effectively helps place data in fast memory of low latency.

Bandwidth sensitivity. ProfDP reports Graph500 BFS is not a bandwidth sensitive benchmark due to its small MF_{BS}. We place all the data objects of BFS into the HBM of Intel KNL and find that the speedup is negligible, less than 1.03×. This proves that ProfDP reports correct guidance for data placement in heterogeneous memory with various bandwidths.

6.2 LULESH

LULESH [24] is an arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in three dimensions. We study a highly-tuned LULESH implementation written in C++ and parallelized with OpenMP. The input is a 80×80×80 mesh.

LULESH shows both high sensitivity to latency and bandwidth. In LULESH, we detect two static data and 50 heap allocated data objects. Each heap allocated data contributes only a small portion of the total memory access latency. Table 2 shows seven data objects with the highest MF values.

Latency sensitivity. Array *vnew* has the highest MF_{LS} value—42, which ProfDP recommends to place into the fast DRAM. We use the case that NVRAM has 600ns latency as an example. If we place *vnew* only to DRAM, which accounts for less than 4% of the memory bytes used in the program, we can achieve around 67% of the ideal speedup that is obtained when placing all data in DRAM. When we place all top seven data objects to DRAM, we get around 75% of the idea speedup. The aggregate memory volume of the seven data objects counts for 40% of the total memory usage of the whole program.

Bandwidth sensitivity. LULESH shows 42% sensitivity to the bandwidth. Table 2 lists the MF_{BS} of the aforementioned seven data objects. It is worth noting that one data object x shows high latency sensitivity but low bandwidth sensitivity. When running LULESH on the KNL machine, allocating all data objects in HBM achieves a 1.46× speedup. As array *nodelist* has the highest MF_{BS}, placing array *nodelist* only into the HBM yields a 1.17× speedup. When we place all the data objects highlighted by ProfDP to HBM, the speedup slightly increases to 1.2×. The gap between the speedup of allocating the highlighted ones in HBM and the speedup of placing all data in HBM is due to that LULESH allocates more than 50 arrays and relatively unimportant memory accesses spread across these arrays; However, these unimportant memory accesses aggregate to more than 40% of total memory access latency.

6.3 AMG2006

AMG2006 [32] is a parallel algebraic multigrain solver for linear systems arising from problems on unstructured grids. We run the code written in C and parallelized with OpenMP with the input of a 30×30×30 grid.

AMG2006 execution shows both high sensitivity to latency and bandwidth. ProfDP identifies 60 heap allocated data objects. Table 2 shows the top data structures with the highest MF_{LS} and MF_{BS}. The array *P_diag_j_new*'s sensitivity to latency is more than 6000% and accounts for only 1.1% of the total memory usage. Its MF_{LS} is 109.

The plethoric allocation points and large memory usage result in a relatively high memory overhead of testing AMG2006 on Quartz. We can only run AMG2006 on Quartz with small inputs that may not represent the behavior of real inputs. Thus, we only test the execution with a real input on KNL. The array *diag_data*'s MF_{BS} is 26, the highest among the data listed in Table 2. If we only place this array in HBM, compared to using DRAM only, the solver phase of AMG2006 yields a 1.06× speedup. If we place the top five data objects with the highest MF_{BS} into HBM, we are able to achieve a 1.12× speedup, which is the same compared to placing all data in HBM. The aggregate size of the top five data objects accounts for less than 22% of the total memory usage. This validates that ProfDP helps achieve optimal speedups with minimum data placement efforts in the fast memory.

6.4 Rodinia NW & Streamcluster

Rodinia Needleman-Wunsch (NW) [9] implements a nonlinear global optimization method for DNA sequence alignments parallelized with OpenMP. Rodinia Streamcluster [10] is a variant of the streamcluster benchmark from the PARSEC suite. We run these two benchmarks with the default inputs released with the benchmark suites.

In NW, there are three data objects, $input\_itemsets$, $referrence$, and $output\_itemsets$. These data objects are of the same size. Data $output\_itemsets$ is neither sensitive to latency nor bandwidth, so Table 2 lists the metrics for the other two data structures, where array $input\_itemsets$ shows both higher MF_{LS} and MF_{BS}.

Figure 6 (c) shows the speedup on Quartz emulator. When allocating the two data structures ($input\_itemsets$, $referrence$) in DRAM, the speedup is the same compared to placing all the three data objects in DRAM. Since $input\_itemsets$ and $referrence$ are accessed along with each other alternatively, simply placing one data object in DRAM yields limited speedup compared to placing both of them in DRAM. When running on KNL, placing $input\_itemsets$ in HBM yields a 1.07× speedup. Placing both the two arrays in HBM yields a 1.10× speedup, which is the same as placing all data in HBM.

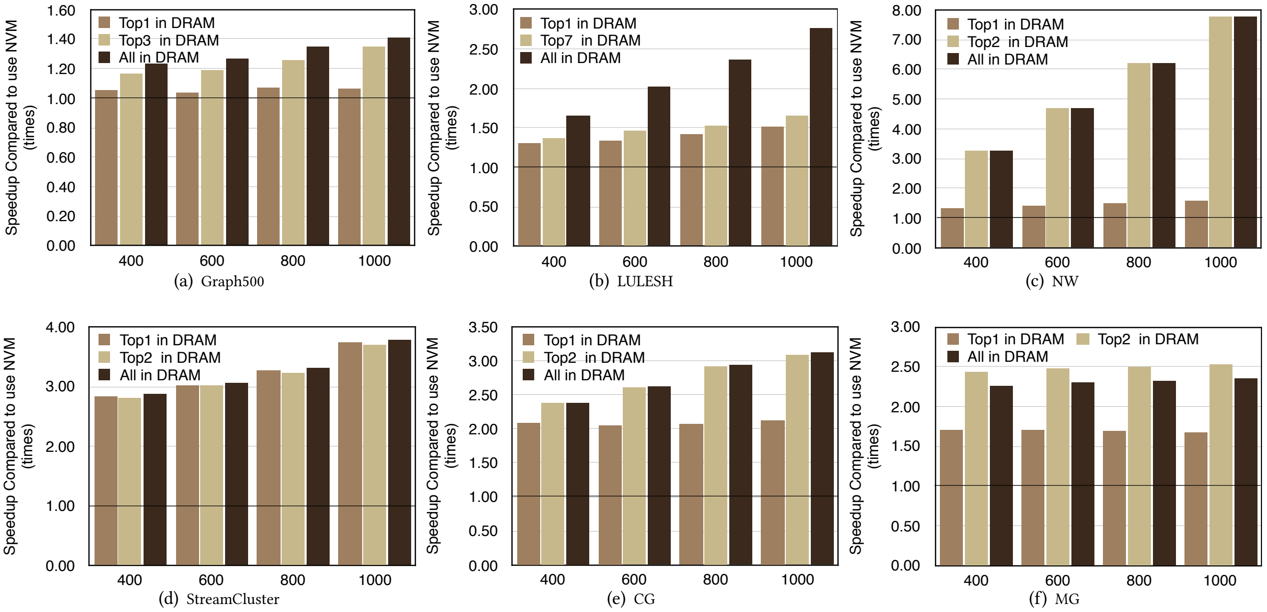
For Streamcluster, ProfDP identifies 13 arrays. Table 2 listed object $block$, which accounts for 97% of the total access latency. Placing `block` in the fast memory in Quartz and KNL yields the same speedup compared to placing all the data in the fast memory. The speedup on KNL is 1.15×.

6.5 NPB CG & MG

CG and MG are two NAS parallel benchmarks, parallelized with OpenMP. We run both of them with input class `C`.

In CG execution, there are over 15 static objects allocated. Among all these data objects, there are two most important ones, a and $colidx$, as shown in Table 2. These two data objects account for 48% of memory volume used in the program. We modified all the static data to be heap allocated, so as to use the NVRAM allocation API in Quartz and `libnuma` to allocate data in HBM. Figure 6 (e) shows that placing these two objects only yields near-optimal speedups compared to placing all the data in the fast memory. CG's sensitivity to bandwidth is not high and the MF_{BS} value for data objects are low, so applying HBM for all the data on KNL gives us limited 1.02× speedup for CG.

MG shows both high latency sensitivity and bandwidth sensitivity in Figure 4. MG has four global static objects allocated: u, v, r and $starts$. Among these data objects, u, v and r each accounts for ~33% of the total memory usage. Furthermore, data objects u



(a) Graph500 (b) LULESH (c) NW

(d) StreamCluster (e) CG (f) MG

Figure 6: Speedups when putting selected data objects in DRAM. Running with Quartz emulator with different NVRAM latency (ns). The based line is putting all data objects in NVRAM.

and *r* play more important roles as shown in Table 2. Their sensitivity and importance are high, resulting in a higher moving factor. Data object *u* shows higher MF_{LS}, while data object *r* shows higher MF_{BS}.

Figure 6(f) lists the speedup we get running with Quartz when placing only *u* in DRAM, both *u*, *r* in DRAM, and all the data in DRAM. When we set the emulated NVRAM latency to be 600ns, the total speedup for placing all the data in DRAM is ~2.3×. Simply placing *u* in DRAM and the rest data in NVRAM yields a 1.7× speedup. Placing both *u* and *r* in DRAM yields the speedup that is even higher than placing all data in DRAM. This is because the bandwidth needs for MG is high. When we partition the data for DRAM and NVRAM, we are able to benefit from the aggregate bandwidth of both DRAM and NVRAM.

For KNL, placing all data in HBM yields a ~2.4× speedup. Placing *u* and *r* in HBM yields a similar speedup—2.3×. Even placing data *u* only in HBM achieves 1.8× speedup, which is not far from the ideal speedup. Thus, ProfDP is effective to identify the most appropriate data to be placed into fast memory to benefit performance most.

7 DISCUSSIONS

In this section, we discuss some limitations of ProfDP. First, ProfDP requires to run a program twice to compute a sensitivity metric. Although in each run, the runtime overhead is negligible, the nature of running a program twice implicitly incurs 2× overhead for the measurement. However, compared to the 40× overhead of the state-of-the-art tool [18], ProfDP’s 2× overhead is still much smaller.

Second, ProfDP does not directly work on KNL because KNL does not provide necessary hardware registers to support capturing memory latency information. Thus, ProfDP needs to run on a Xeon processor-based machine and use the analysis result to guide data placement in KNL. As we show the usefulness of the latency information along with PEBS, we expect Intel will provide this support in the next generation of Xeon Phi.

Third, as to all profilers, ProfDP monitors program execution with specific but not all inputs. To minimize the bias of profiling results, we profile programs with typical inputs (usually released along with the programs) that trigger the most representative behavior in production usage.

8 RELATED WORK

Data placement in heterogeneous memory has been in investigation for years. Prior work mainly utilizes simulators to study the data placement [8, 21, 27, 37, 45, 46, 48]. There are two weaknesses of this approach: first, given the hardware complexity, it is difficult to simulate every feature of heterogeneous memory and its interactions with CPU. Second, due to the high simulation overhead, it is often time consuming or even impossible to evaluate real, long-run parallel programs. Unlike these approaches, ProfDP is implemented and evaluated on real systems, without any hardware extension.

As the most related work, Dulloor et al. [18] proposed data tiering. They categorized all the memory accesses into three patterns: streaming, point-chasing, and random accessing. Different patterns show different potential benefit facing a shorter (or longer) latency. Then they use Pin to instrument every memory access of one application and analyze the access pattern of each data object. The access pattern helps quantify the benefit each data can get when being placed into a faster memory. Shen et al. [39] and Peng et al. [36]

proposed similar Pin-based tools to guide data placement in heterogeneous memory. These tools are based on heavyweight memory instrumentation that incurs high overhead. Moreover, they do not consider the cases when two memory components share different bandwidth capacities instead of latency. Unlike these tools, ProfDP uses a lightweight method to attribute performance metrics to different data objects and quantifies their sensitivity to latency and bandwidth for data placement guidance.

OS-supported data placement [23, 28] for heterogeneous memory is orthogonal to ProfDP. These approaches enhance the OS service for fast data placement or movement. ProfDP complements these approaches by providing user-level guidance for code optimization that can leverage these OS serves for more efficient data placement.

Beyond CPU architectures, heterogeneous memory is widely used in GPUs and other embedded architectures. Chen et al. [12, 13] proposed PORPLE, a framework to guide data placement in the GPU memory hierarchy. They adopted online profiling to assess the benefit of placing data in different memory types and then selected the best placement for the following-on work. Moreover, Agarwal et al. [3] combined program-annotated hints and the access pattern profiles to balance page placement between CPU and GPU. Zhang et al. [49] proposed a non-volatile memory management unit, a new hardware component to connect the solid state disk with GPU. They use this hardware to reduce data movement overhead between CPU and GPU. Unlike CPU code, GPU kernels are often small. A dedicated profiler like ProfDP is necessary to guide performance optimization for CPU codes, which may consist of thousands of lines of code and hundreds of data objects.

9 CONCLUSIONS

This paper presents ProfDP, a novel profiler that guides data placement in heterogeneous memory. ProfDP performs data-centric analysis to associate performance metrics with data objects. Moreover, ProfDP utilizes differential analysis to derive new metrics that prioritize data objects into fast memory with lower latency or/and higher bandwidth. Finally, ProfDP presents its analysis results in a user-friendly way for intuitive data placement guidance. ProfDP leverages performance monitoring units available in modern CPU architectures to perform all the analyses with low runtime overhead, 1.1% on average, and negligible memory overhead. With the evaluation of several parallel programs running on an state-of-the-art DRAM-NVRAM emulator and Intel Knights Landing, we show that ProfDP is able to guide nearly-optimal data placement by placing *minimum* amount of data objects into fast memory.

ACKNOWLEDGMENTS

This project is partially supported by National Science Foundation (NSF) under grant numbers 1618620 and 1619075.

REFERENCES

[1] 2010. Intel® 64 and IA-32 Architectures Software Developer's Manual. (2010).

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* (2010).

[3] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. 2015. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 607–618.

[4] Joseph Antony, Pete P Janes, and Alistair P Rendell. 2006. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *International Conference on High-Performance Computing*. Springer, 338–352.

[5] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, 73–78.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of the 17th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*.

[7] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *Memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).

[8] Niladrish Chatterjee, Manjunath Shevgoor, Rajeev Balasubramonian, Al Davis, Zhen Fang, Ramesh Illikkal, and Ravi Iyer. 2012. Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. 13–24.

[9] Shuai Che. 2009. Rodinia NW Benchmark. https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Needleman-Wunsch. (2009).

[10] Shuai Che. 2009. Rodinia Streamcluster Benchmark. https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Streamcluster. (2009).

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*.

[12] Guoyang Chen and Xipeng Shen. 2016. Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/2925426.2926277

[13] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 88–100. https://doi.org/10.1109/MICRO.2014.20

[14] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. 2007. Scalability Analysis of SPMD Codes Using Expectations. In *Proceedings of the 21st annual international conference on Supercomputing*. ACM, 13–22.

[15] Intel Corp. 2014. NVM Library. http://pmem.io/nvml/. (2014).

[16] Intel Corporation. 2016. Intel Resource Director Technology. https://events.linuxfoundation.org/sites/events/files/slides/cat8.pdf. (2016).

[17] Paul J. Drongowski. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf. (November 2007). Last accessed: Dec. 13, 2013.

[18] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 15.

[19] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. 2013. Bandwidth Bandit: Quantitative Characterization of Memory Contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10.

[20] William Gropp. 2016. Graph500 Benchmark. http://www.graph500.org/. (2016).

[21] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. Software-managed Energy-efficient Hybrid DRAM/NVM Main Memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*. ACM, New York, NY, USA, Article 23, 8 pages.

[22] Intel Corporation. 2010. Linux Performance Tool. http://www.brendangregg.com/linuxperf.html. (2010).

[23] Michael R Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A Doshi. 2013. A Framework for Application Guidance in Virtual Memory Systems. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 155–166.

[24] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.

[25] Martijn HR Lankhorst, Bas WSMM Ketelaars, and RAM Wolters. 2005. Low-cost and Nanoscale Non-volatile Memory Concept for Future Silicon Chips. *Nature materials* 4, 4 (2005), 347–352.

[26] Lawrence Livermore National Laboratory. [n. d.]. LLNL Coral Benchmarks. https://asc.llnl.gov/CORAL-benchmarks. ([n. d.]). Last accessed: Dec. 12, 2013.

[27] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 945–956.

[28] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 369–383.

[29] John DC Little and Stephen C Graves. 2008. Little's Law. In *Building intuition*. Springer, 81–100.

[30] Xu Liu and John Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 183–193. https://doi.org/10.1109/ISPASS.2013.6557169

[31] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 47, 12 pages.

[32] LLNL. 2014. LLNL AMG Benchmark. https://asc.llnl.gov/CORAL-benchmarks. (2014).

[33] Gabriel H Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *ACM SIGARCH computer architecture news*, Vol. 36. IEEE Computer Society, 453–464.

[34] Paul E McKenney. 1995. Differential Profiling. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995. MASCOTS'95., Proceedings of the Third International Workshop on*. IEEE, 237–241.

[35] NASA. 2016. NAS Benchmark. http://www.nas.nasa.gov/publications/npb.html. (2016).

[36] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 82–91. https://doi.org/10.1145/3092255.3092273

[37] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the international conference on Supercomputing*. ACM, 85–95.

[38] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*. IEEE Computer Society, Washington, DC, USA, 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[39] Du Shen, Xu Liu, and Felix Xiaozhu Lin. 2016. Characterizing Emerging Heterogeneous Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 13–23.

[40] Avinash Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–24.

[41] Nathan Russell Tallent. 2010. *Performance analysis for parallel programs from multicore to petascale*. Ph.D. Dissertation. Rice University.

[42] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *SC*.

[43] Nathan R. Tallent, John Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *Proc. of the 2009 ACM PLDI*. ACM, NY, NY, USA, 441–452.

[44] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 37–49.

[45] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 163–173. https://doi.org/10.1109/PACT.2015.10

[46] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 141–152.

[47] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A Higher Order Theory of Locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 343–356. https://doi.org/10.1145/2451116.2451153

[48] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. 2012. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 337–344.

[49] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 13–24. https://doi.org/10.1109/PACT.2015.43