Characterizing Emerging Heterogeneous Memory

Du Shen, Xu Liu
Department of Computer Science
College of William and Mary, U.S.
{dshen, xl10}@cs.wm.edu

Felix Xiaozhu Lin
Department of Electrical and Computer Enginnering
Purdue Univesity, U.S.
xzl@purdue.edu

Abstract

Heterogeneous memory (HM, also known as hybrid memory) has become popular in emerging parallel architectures due to its programming flexibility and energy efficiency. Unlike the traditional memory subsystem, HM consists of fast and slow components. Usually, the fast memory lacks hardware support, which puts extra burdens on programmers and compilers for explicit data placement. Thus, HM provides both opportunities and challenges with programming parallel codes. It is important to understand how to utilize HM and set expectations on the benefits of HM. Prior work principally uses simulators to study HM, which lacks the analysis on a real hardware.

To address this issue, this paper experiments with a real system— the TI KeyStone II— to study HM. We make three contributions. First, we develop a set of parallel benchmarks to characterize the performance and power efficiency of HM. It is the first benchmark suite with OpenMP 4.0 features that is functional on real HM architectures. Second, we build a profiling tool to provide guidance for placing data in HM. Our tool analyzes memory access patterns and provides high-level feedback at the source-code level for optimization. Third, we apply the data placement optimization to our benchmarks and evaluate the effectiveness of HM in boosting performance and saving energy.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes; D.2.8 [*Metrics*]: Performance measures.

General Terms Measurement, Performance

Keywords Heterogeneous memory, data placement, heterogeneous benchmarks, performance characterization

1. Introduction

In modern computer systems, the speed gap between processors and memory has become huge. As a result, accessing main memory incurs not only high latency but also excessive energy. To bridge such a speed gap, CPUs employ mulitple levels of caches. Cache hits reduce memory access latency. Caches are precious resources due to their limited space. For a traditional memory subsystem, hardware manages caches at the granularity of cache lines. Hardware also employs built-in algorithms, e.g., least recently used (LRU), to determine which lines of data to evict. Moreover, multicore systems employ sophisticated protocols (e.g., MESI) to guarantee the data consistency in private and shared caches, associated with different cores and sockets.

Caches are transparent to compilers and programmers. One cannot explicitly control the data placement and replacement in caches. Software usually cannot explicitly control the data locality to exploit caches. One policy does not fit all usage patterns. Hence, existing hardware-managed caches do not provide a straightforward way to achieve the optimal performance. Without the entire program profile, hardware is handicapped in making best data movement decisions. The situation is aggravated in the context of parallel architectures, where cores compete for shared cache. For example, the cached data may be evicted without being fully utilized due to an eviction caused by another core. Contention can significantly degrade program performance. Because of the existing hardware-managed cache system, one can only use some workarounds to explicitly but indirectly interact with caches, such as non-temporal instructions [32], cache partitioning based on page coloring [22], and memory footprint reduction via loop tiling [6]. Though effective, these workarounds rely on special support from instruction set architectures (ISAs), special hardware, customized operating systems and complex source code transformations.

As an alternative, emerging parallel architectures offer heterogeneous memory (HM, also known as hybrid memory) to complement hardware-managed caches. A typical HM system consists of a fast memory component and a slow memory component. The fast component, unlike traditional caches, needs explicit software operations to hoist or evict data in or out. Memory techniques include 3D stack memory [24] and non-volatile memory [4], which together with traditional DRAM form the emerging HM systems. For example, the latest generation of Intel Xeon Phi, Knights Landing (KNL), has on/off-package memories. The on-package memory has 5$\times$ the bandwidth of off-package memory [17]. The KNL's memory hierarchy is a kind of HM. Moreover, scratchpad memory [5] is widely used in accelerators, such as GPUs and digital signal processors (DSPs), and has a higher bandwidth and lower latency than DRAM. *In this paper, we refer to a system with fast and slow memory as HM.*

HM offers flexibility in managing data. For example, programmers can partition the fast memory across different threads to avoid contention. Another advantage of HM is its power efficiency. HM does not require power hungry hardware mechanisms for cache management [36]. In the foreseeable future, HM will become more popular to complement hardware caches for its programming flexibility.

However, software-based data movement can incur much higher overhead than a hardware-based approach. Thus, inappropriate data placement and frequent data movement in HM can significantly degrade memory performance, negating its benefit. Therefore, it is important to characterize the performance and energy consumption

of HM to achieve beneficial data placement. Prior work mainly utilizes simulators to study the data placement [7, 16, 20, 31, 37]. There are two weaknesses to this approach: first, given the complex architectures, it is difficult to simulate every feature of HM and its interactions with the CPUs. Second, due to the high overhead of simulation, it is time consuming to evaluate real, long-run parallel programs. To address these two issues, we study HM in real hardware in this paper.

Given the real hardware evaluation, we have the following questions: (1) For a real parallel program, how should we place its data in HM to achieve high performance? (2) How much can a real HM affect a program's performance and energy? In pursuit of answers to these questions, we make three contributions in this paper:

- We develop a benchmark suite, HMBench. HMBench is coded in OpenMP and runs on HM-based machines. To the best of our knowledge, it is the first benchmark suite based on OpenMP 4.0 standard for studying the performance and energy impacts of HM.
- We design a performance tool, DataPlacer, to guide data placement in HM. This tool can help programmers when they try to port their code to a system with HM. DataPlacer provides rich information sorted by key metrics to intuitively present the analysis results.
- We optimize HMBench guided by DataPlacer. We utilize the original and optimized benchmarks to characterize the capabilities of HM-based architectures and understand the importance of HM in both performance and energy.

We use KeyStone II [36], a server-class ARM+DSP heterogeneous architecture from Texas Instruments, in our studies. KeyStone II's HM consists of an off-chip DRAM and a relatively large on-chip SRAM (scratchpad memory). We choose DRAM+scratchpad memory to study HM because stacked memory and non-volatile memory are not commercially available. Moreover, the scratchpad memory on KeyStone II is much larger than the ones in existing embedded systems, which can be effectively used to emulate the emerging fast memory. In the rest of this paper, we refer to DRAM and scratchpad memory as HM on KeyStone II.

Our experiments show that HM demands extensive attentions to obtain high performance and low energy benefits. Our DataPlacer provides rich information with reasonable overhead to successfully guide data placement in HM. This paper also provides insights and practical evaluation of the KeyStone II.

This paper is organized as follows. Section 2 reviews state-of-the-art work and distinguishes our approach from prior work. Section 3 describes KeyStone II, the testbed we use to evaluate HM. Section 4 describes the design and implementation of HMBench, which is used to characterize the performance and energy effects of HM. Section 5 describers DataPlacer, a tool to provide high-level guidance for optimization with HM. Section 6 studies HMBench in KeyStone II, including performance characterization, power measurement, and HM-based optimization. Section 7 discusses some limitations in studying HM with KeyStone II. Section 8 offers our conclusions and previews future work.

2. Related Work

HM has been recognized as a key alternative or complement to caches and main memory [5]. Due to its simpler hardware design, it can provide high performance, predictability, and energy efficiency. Prior research demonstrates that HM-based systems are able to achieve higher performance than cache-based systems when the program data is carefully placed [1]. In modern accelerators, such as Keystone II[35], HM is regarded as a key enabler for high Gflops/Watt value [18], as has been demonstrated in hand-optimized programs [15]. However, using HM faces key challenges of programmability in obtaining good performance.

Recognizing the significance of HM, much work has been done to ease its management. Unfortunately, due to the difficulty in accessing real hardware platforms and applications, prior evaluation methodologies were limited to two categories: most of them are simulation without actual hardware [11, 28, 30]; some of them replay memory trace on hardware platforms, for which the memory trace is often recorded by using a simulator or software instrumentation [27]. Our efforts bridge the gap with a realistic benchmark suite targeting physical platforms.

The lack of an HM benchmark suite has been an important drawback. In much of the prior work, micro benchmarks have been used [1, 14, 33]. However, it is often difficult to map the micro benchmark outcome to that of real applications. Some work employs macro benchmarks in studying HM, however, with somewhat ad-hoc choices. The macro benchmarks employed include SPEC2006 [11], NAS [10], LULESH [28], or hand-selected apps [30]. It is unclear to what extent these benchmarks can exercise HM and therefore these benchmarks cannot quantify the benefits of HM.

Since HM was often employed in embedded and network processors, embedded systems benchmarks, e.g., MiBench, are often selected in evaluating HM proposals targeting these processors [3, 27]. However, embedded system benchmarks often feature very small working sets, from tens to hundreds of KBs. This is also the case for GPU. Since HM is a critical feature of GPU, well-known GPU benchmarks, e.g. SHOC[12], are often exploiting HM for performance and are used in evaluating software that manages GPU HM [9, 19]. However, SHOC benchmarks are explicitly tuned towards the small HM (a few hundreds of KB) on GPU. Moreover, the CUDA programming model cannot represent the general multi-threading models in mainstream CPU processors. Thus, the resulting programs can hardly exercise the large HM that are emerging in new architectures, such as KeyStone II.

Beyond an effective benchmark suite, there exists no tool to guide the data placement in heterogenous memory systems. Our previous work on memif [21] provides an efficient way to support data movement between fast and slow memory in KeyStone II. However, memif is an OS service for data movement, providing no guidance for the data placement to an application developer. A recent work [13] describes a profiler to analyze memory access patterns to guide data placement. However, the profiler does not provide performance insights such as memory footprint metrics in the full calling contexts, which are important to understand the data structure allocations and program phases.

Unlike existing approaches, we developed HMBench, which has three unique advantages. First, HMBench is developed based on a widely used benchmark suite, Rodinia, which represents the program behaviors of different domains. Second, HMBench leverages the latest OpenMP 4.0 standard. Furthermore, we developed DataPlacer and released it along with HMBench. DataPlacer, to the best of our knowledge, is the first practical tool that provides high-level optimization guidance, such as a variety of metrics within the full calling contexts, when programmers port source code to a HM-based machine.

3. Testbed Description and Motivation

Texas Instruments (TI) Corporation developed KeyStone II, a heterogeneous chip that employs CPUs and DSPs. It aims to achieve high performance with low energy costs. A KeyStone II chip integrates a quad-core ARM Cortex-A15 processor as the host CPU and eight TMS320C66x DSPs as accelerators. Each ARM core has 1.4 GHz clock frequency, while each DSP has 1.2 GHz. The theoretical peak performance of the overall chip is 63 GFLOPS

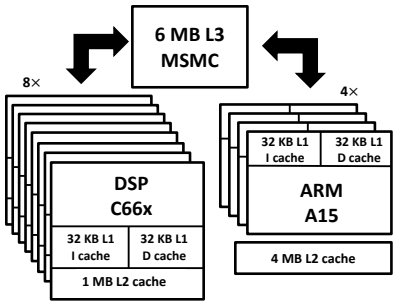

Figure 1: The architecture and memory hierarchy of the KeyStone II. Part of L2 cache in DSP and the whole MSMC shared by DSP and ARM are configured as HM by default.

Table 1: Bandwidth comparison of MSMC and DDR with a single thread.

HM	copy	scale	add	triad
MSMC	5.1 GB/s	4.8 GB/s	5.4 GB/s	5.3 GB/s
DDR	2.7 GB/s	2.8 GB/s	2.6 GB/s	2.8 GB/s

of double precision and 198 GFLOPS of single precision [35]. The ARM cores and DSPs are coupled with security and packet processing and Ethernet switching, which is designed for lower energy consumption, compared to multi-chip solutions. The programming models that KeyStone II supports are OpenMP [29] and OpenCL [34]. The design of KeyStone II is for embedded infrastructure applications, such as media processing, high-performance computing, transcoding, security, gaming, analytics, and virtual desktop.

Figure 1 shows the architecture of KeyStone II and its memory hierarchy. On the host side, each ARM core has a 32 KB L1 instruction cache and 32 KB data cache. All four ARM cores share a 4 MB L2 cache. Both L1 and L2 caches on the host ARM cores are managed by hardware. On the accelerator side, each DSP has a 64 KB L1 cache and a 1 MB L2 cache. By default, the L2 cache is configured as a 256 KB hardware-managed cache and a 768 KB scratchpad memory. The host and accelerators share a Multicore Shared Memory Controller (MSMC), a scratchpad memory of 6 MB. Beyond MSMC, both CPU and DSP have memory controllers to access main memory (DDR). In KeyStone II, both the scratchpad memory in L2 and MSMC are fast HM layers, which requires software to control the data placement and manage data consistency.

With the default configuration, the KeyStone II has three HM layers: L2 scratchpad memory, L3 MSMC, and DDR. Unlike other embedded systems, the scratchpad memories (L2 and L3) in the KeyStone II are large enough to emulate future HM in the mainstream CPU architectures. To explicitly place the data in each layer, TI provides APIs as shown in Listing 1. These APIs, like standard data allocation functions (`malloc` and `free`) in `libc`, can allocate and free memory in different HM layers. The KeyStone II maps the space of different HM layers to disjoint memory segments. One can simply issue a `memcpy` to copy data from one memory segment to another.

Performance characterization of the KeyStone II's HM

To evaluate the impacts of L2, MSMC, and DDR in program performance, we developed two micro benchmarks to characterize KeyStone II: `Stream` and `Lat`. The description of the two micro benchmarks is as follows:

```
/* to allocate in L2 */
void __heap_init_l2 (void *ptr, int size);
/* to manage a heap on L2 */
void *__malloc_l2 (size_t size);

/* to allocate in msmc */
void __heap_init_msmc (void *ptr, int size);
/* to manage a heap on MSMC */
void *__malloc_msmc (size_t size);
void *__calloc_msmc (size_t num,size_t size);
void *__realloc_msmc  (void *ptr,size_t size);
void  __free_msmc (void *ptr);
void *__memalign_msmc (size_t alignment, size_t
     size);

/* to allocate in ddr */
void __heap_init_ddr(void *ptr, int size);
/* to manage a heap on DDR */
void *__malloc_ddr  (size_t size);
void *__calloc_ddr  (size_t num, size_t size);
void *__realloc_ddr (void *ptr, size_t size);
void  __free_ddr  (void *ptr);
void *__memalign_ddr  (size_t alignment, size_t
     size);
```

Listing 1: APIs for managing KeyStone II's L2 cache, MSMC, and DDR.

- Stream, a well-known benchmark [26], is used to quantify the memory bandwidth. It issues memory accesses with streaming patterns, such as array copy, scale, add, and triad. We adapt Stream to measure the bandwidth of both MSMC and DDR.
- Lat is developed for evaluating the performance impact of different data placement policies in HM-based memory hierarchies. The kernel of Lat is a sequence of random accesses to an array placed in a specific HM layer. It also moves data between different HM layers to evaluate the data movement latency.

Table 1 shows the experimental results of bandwidth tests in MSMC and DDR. We can see that MSMC's bandwidth is around 1.7-2× DDR's bandwidth, with respect to different access patterns. Moreover, Lat shows that placing data in MSMC obtains an 8× acceleration compared to placing data in DDR. The L2 scratchpad memory shows similar latency as MSMC. Additionally, Lat evaluates aggressive data movement policy, which always loads each word one by one from DDR to MSMC before using it. This policy significantly degrades the performance because of the nontrivial overhead incurred by the software-based memory movement. It causes a 10× slowdown compared to the original code with all accesses to DDR, and an 80× slowdown compared to the optimal code with all accesses to MSMC.

The experiments on these two micro benchmarks demonstrate the importance of data placement and data movement policies when porting code to HM-based systems. It is necessary to have a set of benchmarks and tools to characterize the performance impact of HM in the real world. The following sections describe the design and implementation of the benchmarks and tools with this purpose. *It is worth noting that the benchmarks we propose in this paper are general for HM systems beyond the KeyStone II*. It is publicly available at `https://bitbucket.org/hmbench/hmbench.git`.

4. Design and Implementation of HMBench

We develop HMBench, a benchmark suite to characterize the performance impact of HM in real hardware. HMBench meets the following four criteria.

1. HMBench needs to work on heterogeneous architectures, i.e., CPU+accelerators, because modern HM (e.g., scratchpad memory + DRAM) is widely used in accelerators, rather than mainstream CPUs.

Table 2: Benchmark descriptions.

Application/Kernel	Domain	Description
`mtrans`	linear algebra	Matrix Transposition.
`mmulti`	linear algebra	Matrix Multiplication.
`bfs`	graph algorithm	Breadth-First Search one a graph.
`cfd`	fluid dynamics	Computational Fluid Dynamics solves 3-D Euler equations for compression fluid flow.
`hotspot`	physics simulation	Hotspot is a thermal simulation benchmark that assesses processor temperatures.
`kmeans`	data mining	K-means clusters points into user specified number of categories based on the distance to other points.
`lavaMD`	molecular dynamics	LavaMD computes particle potentials and relocation forces. It divides a 3D space into cubes for computation.
`lud`	linear algebra	Lud performs matrix LU decomposition.
`nn`	data mining	Nearest Neighbor is a benchmark that finds the first k nearest neighbors for a specified location.
`nw`	bioinformatics	Needleman-Wunsch is a benchmark that performs DNA sequence alignment optimization.
`particlefilter`	medical imaging	Particle Filter assesses the location of a target object with noisy measurements of the target's location.
`pathfinder`	grid traversal	Pathfinder searches a path with the lowest aggregate weights in a 2-D grid.
`srad`	image processing	Srad, with the full name Speckle Reducing Anisotropic Diffusion, is a diffusion algorithm that removes speckles from an image.

2. HMBench should run in parallel, as accelerators typically employ multiple threads for high thread-level parallelism, which leads to significantly different behaviors in HM from sequential execution.
3. HMBench should leverage the interfaces provided by the HM to control the data placement and movement for evaluating different strategies.
4. HMBench should cover different kinds of applications. As the performance of HM is tightly related to memory access patterns, which differ significantly in different kinds of applications, ranging from data analytics to scientific computing. Thus, a high coverage of applications can evaluate HM thoroughly.

HMBench leverages `omp target` to support heterogeneous workloads. Thus, HMBench, with minimal adaptation, works on existing and emerging HM architectures in accelerators or co-processors, such as DSP, GPU and Xeon Phi. Moreover, it provides APIs to encapsulate memory management interfaces provided in HM-based architectures. The initial benchmark suite consists of 13 applications from different areas, including two scientific benchmarks for matrix computation and 11 benchmarks derived from Rodinia 2.2 [8]. The reason we choose to adapt Rodinia benchmarks is that Rodinia has a good coverage of application domains. It has an OpenMP implementation but no HM-aware design, which provides us an opportunity to extend its benchmarks with the OpenMP 4.0 standard and HM-friendly design. HMBench is open to enclose more benchmarks in the future. In the rest of this section, we describe different benchmarks in detail and show our design and implementation specific for HM-based accelerators.

4.1 Benchmark Description

Table 2 shows the descriptions of HMBench benchmarks. They are highly representative in their own fields according to Rodinia's specification [8]. Together with two matrix-based scientific benchmarks, HMBench has good coverage of different domains of real-world parallel applications.

4.2 Simple Benchmark Implementation

Porting these benchmarks to an HM-based system, such as the KeyStone II, is nontrivial. The challenges come from its uncommon architecture and system support, which is different from general-purpose CPUs. Specifically, we need to handle work decomposition between CPU and accelerators, limited compiler support in accelerators, and lack of I/O capability in accelerators. In the rest of this section, we use the KeyStone II to illustrate the challenges and our solutions.

Work decomposition To fully leverage the computing resources in the KeyStone II, we need to split the work into the CPU part and the DSP part. We apply a simple work decomposition strategy in HMBench: we offload all OpenMP parallel regions to the DSP device and leave all non-OpenMP regions running on the CPU. As most computation is done in the OpenMP regions, this strategy can expose as much as computation for evaluating the HM on the accelerator (DSP) side.

To allow different vendor-provided compilers work on different code regions, the offload code region must be encapsulated in a subroutine and placed in a separate target source file. Thus, the CPU compiler can produce CPU binary and also cross compile the DSP binary for execution. To encapsulate each OpenMP region, we identify its inputs and outputs, which are all passed as arguments by reference to the new subroutine.

Code adaptation for the DSP compiler The DSP compiler only supports C-like syntax. Thus, the code running on the DSP cannot use C++ features, such as classes, memory allocations, and type conversions. Moreover, moving data between CPU and DSP requires the support from DSP's OpenMP compiler, which explicitly accepts array names and sizes. However, the compiler does not support multi-dimensional arrays well. In order to move multi-dimensional arrays between CPU and DSP, we need to map them

to a continuous 1-D memory chunk for processing on DSP, and then map them back to the original layout for processing on CPU. Benchmarks, such as `pathfinder`, need such code transformations.

System I/O Some benchmarks, such as `bfs` and `nn`, require input files. Since the DSP software stack in the KeyStone II does not support system I/O, we need to modify the codes so that the host CPU reads input files and map the data to the DSP for computation. DSPs process the data and move it back to the host for writing to the file system. This simple scheme works for most of our benchmarks. However, there is one exception. Benchmark `nn` repeatedly reads in 10 entries of a database for processing, until completing the whole database. To avoid frequent data movement between CPU and DSP, we perform the I/O once to read in all the entries in the input database and offload them altogether to DSP for processing.

Summary: HMBench vs. Rodinia Although HMBench shares some common programs with the Rodinia benchmark suite, it overhauls their implementations: (1) HMBench uses OpenMP 4.0 `omp target` to offload parallel regions to accelerators, where we can use the fast scratchpad memory. We identify the input and output data for `omp target` pragma to ensure the correctness of the code. Moreover, as aforementioned, we need to transpose the array layout for the data transfer between the host and accelerator. (2) HMBench fuses OpenMP regions to minimize the data transfer overhead between the host CPU and the target accelerator. (3) HMBench is extended to manage allocations of heterogeneous memory.

4.3 Limitation of HMBench Implementation

Our implementation of HMBench is straightforward, without taking the HM into consideration. By default, the compiler places all data in the slow memory, e.g., DDR of the KeyStone II. Benchmarks with this implementation do not achieve good performance. We need to take advantage of different HM layers to cache data for efficient accesses. However, determining which data to place in the fast HM layers is difficult, so we need a profiling tool to help make decisions. The next section describes DataPlacer for this purpose.

5. Design and Implementation of DataPlacer

It is challenging for programmers to port code to an HM-based system. One needs high-level guidance to place data objects in the fast HM layers to obtain high performance. In this section, we describe the design and implementation of DataPlacer, a profiler that identifies optimization opportunities of data placement in HM. Figure 2 shows the workflow of DataPlacer. DataPlacer works on an x86 host machine. It monitors program execution and provides optimization guidance for porting this code to an HM-based target machine. DataPlacer has the following three features to make it an effective tool.

- DataPlacer provides software metrics only. Because the host and target machines have different architectures, using the host's hardware metrics is inappropriate to guide the optimization in the target HM-based machine.
- DataPlacer provides high-level optimization guidance for programmers. The guidance can be easily used for source code transformation.
- DataPlacer can monitor parallel program execution with reasonable overhead. For private and shared HM layers between multi-cores, DataPlacer provides different optimization guidances.

In the rest of this section, we describe the basic methodology of DataPlacer and several refinements to make it practical.

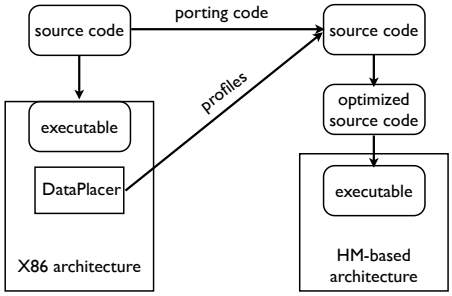

Figure 2: The functionality of DataPlacer. DataPlacer monitors program execution on x86 and generates pure software-based profiles to guide program optimization when porting the code to an HM-based architecture.

5.1 Basic Methodology of DataPlacer

DataPlacer leverages Intel Pin [25] to instrument binary and collect memory traces. All the analyses are based on the memory traces without using any information from architecture-specific hardware performance counters. To provide high-level optimization guidance, DataPlacer performs array-centric analysis. It identifies arrays with a significant amount of accesses, which, if put into fast HM layers, can improve performance. With this guidance, programmers can easily transform the source code for optimization. To achieve array-centric analysis, DataPlacer needs to monitor array allocations, associate memory accesses with arrays, and derive metrics for analysis.

Tracking array allocations DataPlacer leverages Pin to analyze a binary executable and monitor its execution to extract array allocations. DataPlacer monitors both static and heap arrays. On one hand, DataPlacer reads the symbol table of the binary to identify the names and memory ranges of static arrays. On the other hand, DataPlacer instruments array allocation functions, such as `malloc`, `calloc`, and `realloc`, to capture the allocated memory ranges as well as the allocation location mapped to the source code with the help of compiler debugging information. DataPlacer logs these memory ranges and IDs (names for static arrays and allocation sites in source code for heap arrays) into a map for further use.

Collecting and attributing memory traces DataPlacer utilizes Pin to instrument both memory loads and stores for their effective addresses. Upon a memory access, DataPlacer checks the map to identify the memory interval that includes the effective address of this memory access and associates it with the array. DataPlacer counts the number of accesses attributed to each array. For multithreaded programs, accesses from multiple threads can be attributed to the same array at the same time, so DataPlacer needs to use atomic operations to ensure the correctness of accumulating the counter.

Deriving metrics From the array-centric analysis, DataPlacer obtains the number of accesses to each array. Arrays with significant accesses are candidates for being placed into the fast HM layers. To weigh the significance of arrays, we use Equation 1 to derive a metric $\hat{F}$ for each array, which is the average access frequency per byte. In the equation, C is the total number of memory accesses to an array. S is the number of memory bytes allocated to the array.

$$\hat{F} = \frac{C}{S} \tag{1}$$

DataPlacer sorts all arrays according to $\hat{F}$. With a greedy algorithm, DataPlacer recommends placing arrays with high $\hat{F}$ until the space of HM runs out.

5.2 Refined Methodology of DataPlacer

The basic design of DataPlacer is inadequate to be used in practice. There are five major issues.

1. Metrics $\hat{F}$ and C alone are insufficient in providing effective guidance. We need more insightful access pattern analysis to extract more features of an array, such locality, beyond the simple access quantity.
2. If the fast HM layers have limited space and the arrays used in the programs are too large to fit in, DataPlacer cannot place such large arrays. Moreover, not all elements of an array have the same number of accesses to receive the equal treatment.
3. HM may have layers that are private or shared between cores. Applying the same data placing strategy to different kinds of HM layers may hurt performance. For example, inappropriate placement can cause high overhead due to maintaining data consistency.
4. DataPlacer produces static data placement guidance. Once the data is loaded into HM, it never gets replaced. In practice, static placement preclude optimal performance because program execution can have different phases with different memory access patterns.
5. A system that integrates both traditional hardware caches and HM is difficult to optimize. DataPlacer needs to take this into consideration for HM-based data placement.

Thus, we refine DataPlacer to address all these issues.

Data locality An array with a large stride or a random access pattern does not exploit the reuse in caches. We call such array one of poor locality. An array of poor locality can significantly degrade program performance because accesses to this array are more likely to suffer from cache misses and high exposed memory latency. Therefore, DataPlacer prioritizes the placement of arrays with bad locality into fast HM. DataPlacer adopts our previous approach [23] to collect the reuse distance of memory accesses and associates them with arrays. The technology is to instrument all memory accesses and record the trace of effective addresses in a hash map for the computation of reuse distance. We report the instructions and arrays associated with long reuse distances as with poor locality. We evaluate the necessity for placing arrays of poor locality in Section 6.

Large arrays DataPlacer decomposes the memory intervals allocated for large arrays into small chunks with the sizes not larger than N. N is tunable by programmers; by default, we set it as one tenth of the HM size. DataPlacer treats each chunk as a separate array and performs original array-centric analysis. With the offsets computed for chunks in the array, programmers can easily place part of the array in the HM. Besides handling large arrays that do not fit into the HM, DataPlacer's array decomposition is more appropriate for handling irregular access patterns. With irregular access patterns, elements in an array may have different access frequencies. The array decomposition provides more details in the array internals for data placement.

Private vs. shared HM HM can be private or shared in a multicore system, e.g., the KeyStone II. For example, each DSP in the KeyStone II has a private fast layer—L2 cache— and all eight DSPs share a fast layer—L3 MSMC. Optimizations on these two kinds of fast HM layers are different. On one hand, DataPlacer recommends thread-local arrays rather than shared arrays to be placed in private HM because handling shared arrays needs to maintain data consistency. For example, if an element of a shared array is updated by one thread in the private HM, the update should be written back to the main memory. Moreover, all of the copies of this element in different private HMs have to be invalidated and reloaded from the main memory. In a traditional cache system, this data consistency is guaranteed by the hardware, which is efficient. However, HM requires software to keep the data consistency, which is expensive. Thus, DataPlacer avoids recommending shared arrays to be placed in private HM layers.

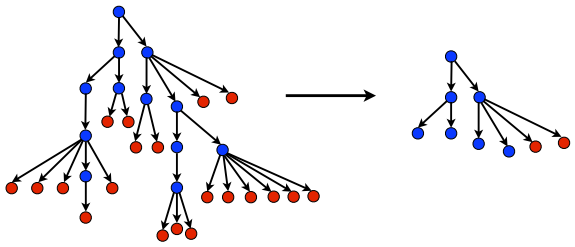

Figure 3: Creating a CCT for a program and pruning it by discarding nodes with small numbers of memory accesses. The blue nodes are internal functions, while the red nodes are leaf functions.

On the other hand, DataPlacer prioritizes shared arrays to be placed in shared HM layers. If there is space, DataPlacer places local arrays in the shared HM. One strength of this strategy is that no software-based data consistency is needed. Moreover, shared arrays are used by multiple threads, so loading them into shared HM can benefit many threads. In contrast, loading local arrays into shared HM only benefits a subset of threads, rather than all of them.

To provide appropriate optimization guidance, DataPlacer identifies whether an array is local or shared and adapts the array-centric analysis accordingly. When it attributes memory accesses to arrays, it also associates the IDs of threads that perform the accesses with the array. If an array is accessed by more than one thread, DataPlacer recognizes it as a shared array. Otherwise, it is a local array.

Static vs. dynamic placement The basic implementation of DataPlacer produces the strategies of array placement in a static way: once an array is placed in the HM, it is never evicted throughout the entire execution. However, a typical program has phases. Loading an array into the fast HM layer without using it in some phases can waste the precious HM resources. Therefore, we improve DataPlacer to provide guidance for placing arrays dynamically. The main challenge of generating guidance for dynamic placement is to identify the phase changes to apply dynamic adaptation of data placement. Moreover, DataPlacer needs to provide high-level guidance that can be used by programmers to refactor their source code.

To address the challenge, DataPlacer makes an assumption: phase changes occur at function boundaries. In other words, DataPlacer applies the same data placement strategy inside a function. When switching to a different function, DataPlacer adapts the data placement strategy, if needed, based on the memory accesses in the new function only. However, frequent changes of data placement are costly because of the heavyweight software-based data movements. To reduce the overhead, DataPlacer adapts the data placement when coming into a new function that invokes a significant number of memory accesses. To provide high-level guidance, DataPlacer associates memory accesses with functions in their full calling contexts.

DataPlacer uses Pin [25] to instrument every function call and return instruction. It maintains a shadow stack to track function frames in the system execution stack. When calling a function, DataPlacer pushes the function frame, identified by the starting address of the function, into the shadow stack. When returning from a function, DataPlacer pops the function frame on top of the shadow stack. The calling context of any instruction under execution is in the shadow stack. DataPlacer accumulates the number of memory

```
A total of 184016202 memory accesses.

Rank 0 >>>>
Dynamic Data 45703 chunk 1 accessed 33107904 times.

Data allocation call path:
45678:0x7f20c32816e0:pushq %rbp:malloc::0
  45676:0x400dda:callq 0x400bf0:main:[...]/main.c:157
    30599:0x7f20c3227b03:callq %rax:__libc_start_main::0
      29824:0x401b40:callq 0x400ba0:_start: [...]/sysdeps/x86_64/start.S:122
        1:(nil)::THREAD[0]_ROOT_CTXT::0

size = 898.109 KB contribution = 17.9918% F = 36.00

accessed by threads:
12445583 6927600 6867360 6867360

Footprint and accesses per context
Footprint is 4591168 Bytes, #accesses is 22831020
Calling contexts:
24341:0x401cf0:movsxdl  (%r14,%rax,4), %rcx:main._omp_fn.1::0
  24289:0x401590:callq  0x401c10:main::0
    14026:0x7f896debbb03:callq  %rax:__libc_start_main::0
      13647:0x401b40:callq  0x400ba0:_start:/home/abuild/rpmbuild/BUILD/
      glibc-2.19/csu/../sysdeps/x86_64/start.S:122
        1:(nil)::THREAD[0]_ROOT_CTXT::0
```

Figure 4: An output example of DataPlacer when monitoring `srad`.

accesses acc to the function frame on top of the shadow stack, as exclusively to the function (not to its callers).

To efficiently maintain these per-function metrics, all the calling contexts are organized in a compact data structure, called a calling context tree (CCT) [2], by merging all common prefixes. Figure 3 shows a typical CCT. The root node of a CCT is the starting function, typically "main" or "thread start"; the internal nodes (in blue) are functions that have function calls inside; the leaf nodes (in red) are ones with no function call inside. To compute the inclusive metrics for each node (i.e., the aggregate metric of the function and all its callees), DataPlacer traverses the CCT from bottom to top to accumulate the inclusive acc for every node. It then prunes the CCT, leaving the nodes that account for significant proportions of memory accesses during the entire program execution, as shown in Figure 3. For leaf nodes in the pruned CCT, DataPlacer treats them as separate phases, in which the data placement strategy is dynamically adapted according to the array-centric metric $\hat{F}$, computed with memory accesses in the phase. As for the optimization, programmers need to flush the data in the fast HM layers at the beginning of the function and then place the data according to DataPlacer's suggestions.

Hybrid memory subsystem A system with hybrid hardware- and software-managed cache/memory is challenging for data placement. For example, each of KeyStone II's DSPs has a 256 KB L2 cache that is managed by hardware. We find that if the dataset of a program is small enough to fit into the hardware cache, placing data objects into HM does not help its performance. Therefore, DataPlacer collects memory footprints fp for each node in the pruned CCT, like acc. Memory footprint is defined as the unique memory bytes accessed in a calling context. Table 3 shows how DataPlacer makes optimization decisions based on these two metrics. DataPlacer suggests that optimizing data placement in contexts with high acc and fp can lead to significant performance improvement. If acc is low and fp is high, the code exposes little locality. Thus, placing data into HM does not benefit performance much. Moreover, if fp is low, the hardware caches can hold all the data, minimizing the effects of HM.

To collect fp, DataPlacer creates a hash set to maintain all unique memory bytes accessed exclusively to CCT nodes. Like acc, the hash set is merged from bottom to top in the CCT along all call paths for the inclusive footprint of a context. fp of a function is computed as the size of the inclusive hash set associated with the function.

5.3 DataPlacer Output

DataPlacer produces the text output once the program finishes its execution. Figure 4 shows an example output of DataPlacer for `srad`, a case study to be described in the next section, running with four threads. At the beginning of the output file, DataPlacer shows the total number of memory accesses in this execution. Then, DataPlacer ranks all data objects in a descending order by $\hat{F}$. In the figure, we only show one example array. DataPlacer outputs the data ID (for DataPlacer's internal usage) and the number of accesses. For static arrays, DataPlacer displays its name. For dynamic arrays, DataPlacer prints the full call path so that a programmer can associate the data object with the source code. In this example, the data object is allocated on heap by `malloc`. DataPlacer also maps the call paths to the source code for easy interpretation: the `malloc` is called at line 157 in `main.c`.

Moreover, DataPlacer computes the array size in bytes, the contribution of memory accesses (in percentage) to the whole program execution, and $\hat{F}$. DataPlacer lists the number of accesses by each thread and identifies whether the data object is shared or private. In this example, as all the four threads access this array, this array is shared by all the threads and should be placed into the shared fast memory layer.

DataPlacer also reports acc and fp, as shown in Table 4 for the whole program. To give dynamic optimization guidance, DataPlacer further reports acc and fp in all the functions with full calling contexts. Figure 4 shows one example OpenMP function. As this OpenMP function performs the most computation, DataPlacer suggests to target this function with one strategy to place the array in the fast memory, highlighted in the allocation call path.

It is worth noting that the text output of DataPlacer can contain thousands of lines because all the allocations and functions with their full call paths are included. However, by sorting all the items (sorting arrays with $\hat{F}$ and sorting functions with acc), we can successfully shrink the searching space and focus on a few arrays and function contexts. In the future, we plan to build a graphical interface for DataPlacer for easy data interpretation.

6. Evaluation

We evaluate HMBench and DataPlacer on the TI KeyStone II. The configuration of KeyStone II is described in Section 3. The compiler on the host side is `gcc 4.7.2`, while the complier on the device side is TI's OpenMP Accelerator Model Compiler `clacc 1.1.1`. We compile all the benchmarks in HMBench with `-O3`. DataPlacer collects execution profiles of HMBench on an x86 machine, which has 16 Intel Xeon 3.2 GHz cores, with 192 GB memory. DataPlacer monitors the program executions with eight threads. The overhead is 40-60× the native execution. We average the execution time and power consumption with running each experiment five times; we find that variance is negligible.

We discuss our experiments in four aspects. In Section 6.1, we optimize HMBench according to the guidance of DataPlacer. In Section 6.2, we characterize the performance difference of HMBench due to HM in KeyStone II. In Section 6.3, we characterize the difference in power consumption with the utilization of HM in KeyStone II. Finally, we discuss some issues in our experiments in Section 6.4.

6.1 Optimizing HMBench on KeyStone II

With the guidance of DataPlacer, we are able to apply the optimizations to all the benchmarks in HMBench. To evaluate DataPlacer and demonstrate our optimizations, we study four benchmarks in detail. Without specific explanation, the speedups we report are over the default execution of HMBench on KeyStone II, which does not use the fast HM.

Table 3: DataPlacer's optimization decisions based on two metrics.

acc	fp	optimization decisions
high	high	optimization with high priority
low	high	little performance gains (low data reuse)
low/high	low	little performance gains (dataset fit in hardware cache)

```
/* allocation and initialization */
#pragma omp parallel for ...
for( i = 0 ; i < N ; i++)
  for( j = 0 ; j < N ; j++)
    B[j][i] = A[i][j];
```

Listing 2: Code snippet of `mtrans`: matrix `A` is transposed into matrix `B`.

```
float *m;
...
/* file input */
create_matrix_from_file(&m,input_file,&matrix_dim);
...
/ *kernel computation */
lud_omp(m, matrix_dim);
```

Listing 3: Code snippet of `lud`. Array `m` reads the input file and then is passed for kernel computation.

mtrans There are two arrays in this micro benchmark, the original matrix `A` and the transposed one `B`, as shown in Listing 2. Both matrices are shared by all threads. DataPlacer suggests we should place matrix `B` into fast memory if there is not enough room for both, because of the bad locality in matrix `B`. We optimized the application following DataPlacer and observed an 11.51× speedup. In contrast, placing matrix `A` (of good locality) into fast memory obtains only a 5.52× speedup, or half the performance gain of placing matrix `B`.

lud There are two phases in `lud`: a file input phase and a kernel computation phase. DataPlacer identifies that there is only one significant array `m`, which contains all the matrix data for decomposition computation. As shown in Listing 3, m is allocated in `create_matrix_from_file` and used in `lud_omp`, the parallel kernel. Array `m` accounts for ∼18% of total memory accesses. Moreover, `m` is shared by all threads, so DataPlacer suggests placing it into fast memory. We apply the optimization according to DataPlacer's guidance and achieve a 3.95× speedup for the OpenMP parallel region when running on eight DSPs.

nw DataPlacer identifies two significant arrays used in `nw`. As shown in Listing 4, the two arrays `referrence` and `input_itemsets` with the same size, 2.2 MB. Both of them are used in a parallel region, shared by all threads. These two arrays account for ∼32% of total memory accesses. With this performance insights, DataPlacer recommends placing both arrays into fast memory, which leads to a 1.5× speedup for the overall program.

srad Besides the array highlighted in Figure 4, DataPlacer identifies six more significant arrays, as shown in Listing 5, which account for ∼50% of total memory accesses in the program. Threads share six of these arrays in the following parallel region. Ideally, DataPlacer recommends placing all seven arrays in the fast memory. However, due to the limited space, the fast memory cannot hold all the arrays. With the analysis of DataPlacer, we place five arrays with the highest $\hat{F}$ to the fast memory. These arrays are `image`, `dN`, `dS`, `dW` and `c`. As for the optimization, the array `image` needs to be initialized in the host and then passed to the device. For the other four arrays, they can be initialized on the device. With this optimization, we obtain a 1.15× speedup.

```
referrence = (int *)malloc( max_rows * max_cols *
    sizeof(int) );
input_itemsets = (int *)malloc( max_rows * max_cols
     * sizeof(int) );
...
/* process top-left matrix */
#pragma omp parallel for ...
for( idx = 0 ; idx <= i ; idx++){
  ...
  input_itemsets[index]= maximum( input_itemsets[
       index-1-max_cols] + referrence[index],
      input_itemsets[index-1] - penalty,
      input_itemsets[index-max_cols] - penalty);
}
...
/* process bottom-right matrix */
#pragma omp parallel for ...
for( idx = 0 ; idx <= i ; idx++){
  ...
  input_itemsets[index]= maximum( input_itemsets[
       index-1-max_cols] + referrence[index],
      input_itemsets[index-1] - penalty,
      input_itemsets[index-max_cols] - penalty);
}
```

Listing 4: Code snippet of `nw`. Arrays `referrence` and `input_itemsets` are frequently accessed.

Further analysis on speedups Table 4 summarizes the optimization to all the benchmarks in HMBench with the guidance of DataPlacer. In the table, we show the footprint, the number of accesses, and the number of arrays placed into fast memory under the guidance of DataPlacer. We set two baselines to make the comparison. Baseline $B1$ is the default program configuration without utilizing the fast memory. Baseline $B2$ utilizes scratchpad memory with a naive data placement strategy: first come, first served. From the table, we can see that eight of 13 benchmarks benefit from the HM optimization and achieve more than 1.10× speedups over $B1$. Among them, mtrans and lud obtain significant speedups. However, benchmarks like `kmeans`, `lavaMD`, `particlefilter`, and `pathfinder` obtain nearly no performance improvement. Therefore, not all kinds of benchmarks can benefit from HM. We discuss the performance impact of HM to different kinds of benchmarks in the next section. Moreover, we can see that five out of 13 benchmarks (`mtrans`, `mmulti`, `bfs`, `hotspot`, and `nw`) achieve more than 1.10× speedup over $B2$. On average, DataPlacer achieves a speedup of 1.56× and 1.17× over $B1$ and $B2$ baselines, respectively.

6.2 Performance Characterization

In this section, we characterize the performance impact of HM and identify the workload features that can benefit from HM. We mainly focus on the performance of parallel regions in these benchmarks. Common to all the benchmarks that benefit from HM, they have three features. First, their parallel regions should be large enough to avoid parallel overhead in OpenMP from overwhelming the ex-

```
image_ori = (fp*)malloc(sizeof(fp)*image_ori_elem);
...
image = (fp*)malloc(sizeof(fp) * Ne);
...
dN = malloc(sizeof(fp)*Ne); // north direction
     derivative
dS = malloc(sizeof(fp)*Ne); // south direction
     derivative
dW = malloc(sizeof(fp)*Ne); // west direction
     derivative
dE = malloc(sizeof(fp)*Ne); // east direction
     derivative
...
c  = malloc(sizeof(fp)*Ne); // diffusion
     coefficient
...
resize( image_ori, image_ori_rows,...);
...
#pragma omp parallel for ...
for (j=0; j<Nc; j++)
  for (i=0; i<Nr; i++) {
    ...
    // divergence
    D = cN*dN[k] + cS*dS[k] + cW*dW[k] + cE*dE[k];
    // updates image
    image[k] = image[k] + 0.25*lambda*D;
    ...
  }
}
```

Listing 5: Code snippet of `srad`. There are seven arrays with significant accesses in the OpenMP parallel region.

Table 4: The analysis and optimization guidance provided by DataPlacer. The speedups are measured for all benchmarks running with eight threads in KeyStone II.

benchmarks	footprint (bytes)	accesses	#arrays in HM	Speedup over $B1$	Speedup over $B2$
`mtrans`	2.9E6	1.3E7	1	**11.51**×	**2.09**×
`mmulti`	3.7E8	6.4E8	1	**2.18**×	**1.85**×
`bfs`	2.0E6	1.4E8	6	**1.31**×	**1.18**×
`cfd`	6.0E5	1.2E7	2	1.05×	1.03×
`hotspot`	1.3E7	3.7E8	2	**1.17**×	**1.10**×
`kmeans`	6.7E7	1.4E10	1	1.01×	1.01×
`lavaMD`	3.3E6	3.7E6	3	1.01×	1.01×
`lud`	2.1E6	3.1E8	1	**3.95**×	1.00×
`nn`	2.1E6	2.8E9	1	**1.10**×	1.00×
`nw`	4.7E6	8.8E7	2	**1.51**×	**1.30**×
`particlefltr`	7.4E6	8.0E8	10	1.02×	1.00×
`pathfinder`	4.0E6	4.0E8	1	1.01×	1.06×
`srad`	1.0E7	1.2E9	5	**1.15**×	1.02×
Geo.mean	/	/	/	**1.56** ×	**1.17**×

ecution time. For example, the parallel region in lud accounts for almost 100% of the program execution time, so our optimization shows a significant speedup. Second, parallel regions have reasonable memory footprints and accesses. Third, benchmarks have hot arrays, whose placement in the HM can benefit a large number of memory accesses. For example, `lud`, `nw`, and `srad` have hot arrays; placing them in HM can benefit 18-50% of the total memory accesses.

However, as shown in Table 4, there are benchmarks having little performance improvement with HM optimization. With the help of DataPlacer, we obtain the benchmark characteristics that may not benefit from HM.

- Small footprints. If the memory footprint is small, all data can be loaded into KeyStone II's L1 and L2 caches, so optimization does not help. For example, `cfd` has less than 1MB footprints that can fit into the hardware-managed L2 cache. Thus, the speedup for `cfd` is trivial.
- Streaming access patterns. If a benchmark has a streaming access pattern, loading data into HM does not benefit from many reuses. For example, `lavaMD` has a streaming access pattern ($fp \approx acc$); optimizing it shows nearly no speedup.
- Large footprint with a uniform access pattern. If all arrays in a benchmark are uniformly accessed, placing a small number of arrays in the fast HM layers does not significantly improve the performance. For example, `kmeans` has a large footprint with an uniform access pattern. Placing only a small subset of data into HM leads to nearly no speedup.

In addition to the performance, HM can improve program scalability. We evaluate strong scaling [1] of all benchmarks in HMBench. Most of the benchmarks have slightly better scalability when optimized with HM. The reason is that MSMC has much larger bandwidth than DDR in KeyStone II, so contentions in memory bandwidth can be reduced with the use of MSMC.

6.3 Power Characterization

We measure the power consumed by each application on a TI evaluation board that features one Keystone 66AK2H SoC, running with 1, 2, 4 and 8 DSP cores. Without a convenient way to tap into the power rails for the DSP and memory, we measure the board-level power consumption, by sampling the voltage and current with an external digital multimeter, Agilent 34450A. When the board is idle (no workload on CPU and DSP), we measure the board power as the baseline; when workload is being executed, we sample the board power. We repeatedly run each benchmark multiple times to minimize the measurement errors introduced by the system noise. We report the workload energy consumption by integrating the power over time.

Figure 5 compares the energy consumption of the whole system when running the original (without using MSMC) and optimized benchmarks. As shown in the figure, the optimizations with HM for HMBench always reduce energy consumption. We can see that seven benchmarks have more than 20% energy reduction due to our HM optimization. It is worth noting that some benchmarks, such as `cfd`, `kmeans` and `particlefilter`, do not obtain speedups with the utilization of fast HM, but they obtain nontrivial energy reduction, 9-18%.

Due to the design limitation of the evaluation board, its static power is known to be much higher than that of a production device. To further highlight our efficiency benefit, we compare the dynamic energy consumption, which is computed as the difference between the measured energy and the baseline energy on chip. Figure 6 shows the measurement results: most of the benchmarks have significant reduction in energy consumption, more than 2× on average. We further notice that `nn` and `pathfinder` after optimization consume more dynamic energy (Figure 6), but less overall energy (Figure 5). The reason is that the execution time reduction of these benchmarks saves a significant amount of static energy, which surpasses the dynamic energy increment.

[1] Strong scaling means that the problem size is constant and the number of cores increases.

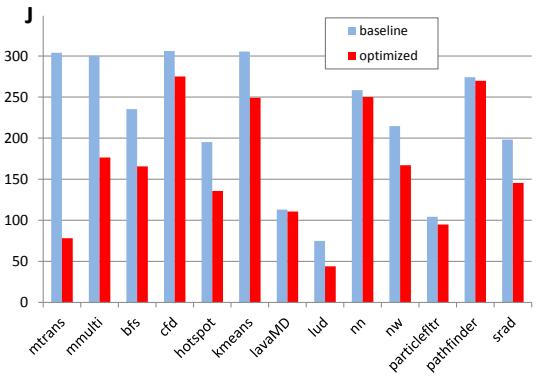

Figure 5: Comparison of whole-system energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules.

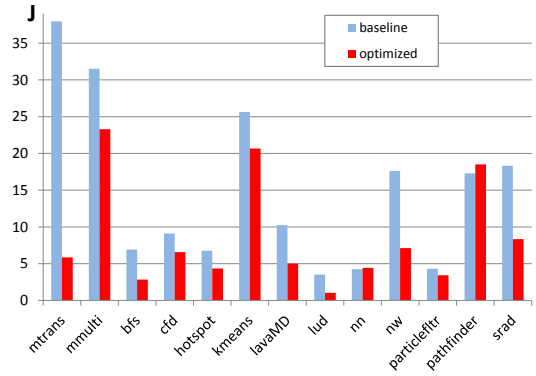

Figure 6: Comparison of dynamic energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules.

6.4 Takeaways from Experimenting KeyStone II

With the evaluation of a benchmark suite running on a real system, KeyStone II, we identify that HM can benefit both performance and power consumption for many, but not all, applications. A performance tool, like DataPlacer, is necessary to guide the use of HM for the best performance.

However, we cannot further characterize HM's impact in performance and energy with hardware performance events on KeyStone II. Such hardware events include L1/L2 cache accesses/misses and MSMC accesses. The reason is that DSPs on KeyStone II lack of performance counters to record such events. Without this information, we cannot explain some phenomena. For example, we cannot directly understand DDR contention when scaling benchmarks to more DSPs. Moreover, we have no idea why nn and pathfinder consume more dynamic energy with HM optimizations. This work motivates TI to provide such support in DSPs to better understand their application behaviors.

7. Limitations with KeyStone II

Our study of HM based on KeyStone II has two limitations. First, the fast memory (L2 and MSMC) on KeyStone II has a small size (6 MB), so we need to tune the HMBench inputs with small sizes to make sure the fast memory can hold a sufficient portion of arrays to affect the performance. We tried large inputs of HMBench, which are difficult for us to obtain the performance gains. We expect 8-16 GB fast memory in the emerging architectures, where we foresee the benefit can be obtained from optimizing HMBench with large inputs.

Second, we lack the insights of the memory behavior in HM-based accelerators because there are no performance monitoring units (PMUs) in DSP. Thus, we cannot precisely explain why several HMBench benchmarks fail to obtain speedups with placing data in the fast memory. As the mainstream CPU architectures will employ HM in the future, we expect to use CPU's PMU to collect rich information to understand the HM performance.

8. Conclusions and Future Work

In conclusion, this paper introduces HMBench and DataPlacer to study the impact of software-managed heterogeneous memory in a real system, the TI KeyStone II. HMBench is the first OpenMP benchmark suite that adopts OpenMP 4.0 standard and works on heterogeneous architectures. DataPlacer is a profiler to provide guidance for data placement in different layers of software-managed cache and memory. Using HMBench and DataPlacer, we observe the insight that HM plays an important role in both boosting performance and reducing energy consumption. Moreover, we leverage HMBench and DataPlacer to characterize the performance gains with HM.

Our future work is twofold. First, we will develop more benchmarks for HMBench to make it as the standard benchmark suite for evaluating HM-based systems and compilers. Second, we will extend DataPlacer to provide low-level guidance for compiler-based optimization for HM. Such low-level information includes the finer granularity of data placement on cache lines or pages, instead of arrays. We believe that optimizations on HM from both high-level source code transformation and low-level compiler-supported code generation can achieve the optimal performance.

Acknowledgements

We would thank Prof. Doug Lea and anonymous reviewers for their constructive comments. We thank Milind Chabbi for proofreading the paper. We also thank Ravi Gupta from Purdue University for his help in setting up power measurement on KeyStone II. Finally, we thank TI for the donation of KeyStone II boards. Without the donation, we could not carry out this research.

References

[1] J. Absar and F. Catthoor. Analysis of scratch-pad and data-cache performance using statistical methods. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[3] K. Bai and A. Shrivastava. Automatic and efficient heap data management for limited local memory multicore architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 593–598. IEEE, 2013.

[4] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 10–22, 1992.

[5] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International*

Symposium on Hardware/Software Codesign, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM. ISBN 1-58113-542-4. doi: 10.1145/774789.774805. URL http://doi.acm.org/10.1145/774789.774805.

[6] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[7] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. 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, pages 13–24, 2012.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 44–54, Washington, DC, USA, 2009.

[9] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 88–100, Dec 2014. doi: 10.1109/MICRO.2014.20.

[10] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada. Prefetching irregular references for software cache on cell. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 155–164, New York, NY, USA, 2008.

[11] J. Cong, H. Huang, C. Liu, and Y. Zou. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 960–965, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0636-2. doi: 10.1145/2024724.2024937. URL http://doi.acm.org/10.1145/2024724.2024937.

[12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[13] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[14] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 321–330, New York, NY, USA, 2006.

[15] Y. Gao, F. Zhang, and J. D. Bakos. Sparse matrix-vector multiply on the keystone ii digital signal processor. In *IEEE HPEC*, 2014.

[16] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos. Software-managed energy-efficient hybrid dram/nvm main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 23:1–23:8, New York, NY, USA, 2015. ACM.

[17] Intel. Knights Landing, the Next Generation of Intel Xeon Phi. http://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/. Last accessed: Dec. 08, 2014.

[18] L. J. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, 2009.

[19] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into GPU on-chip memory resources. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '15, New York, NY, USA, 2015. ACM.

[20] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. 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, pages 945–956, Washington, DC, USA, 2012. IEEE Computer Society.

[21] F. X. Lin and X. Liu. 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, pages 369–383, 2016.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008. doi: 10.1109/HPCA.2008.4658653.

[23] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overheads. In *Proc. of the 2013 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Austin, TX, USA, April 21-23, 2013.

[24] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, 2008.

[25] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065034.

[26] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream.

[27] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 31–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375640. URL http://doi.acm.org/10.1145/1375634.1375640.

[28] M. R. Meswani, G. H. Loh, S. Blagodurov, D. Roberts, J. Slice, and M. Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16. IEEE, 2014.

[29] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[30] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, July 2000. ISSN 1084-4309.

[31] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, 2011. ISBN 978-1-4503-0102-2.

[32] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9.

[33] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415. IEEE, 2002.

[34] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12 (3):66–73, May 2010.

[35] Texas Instruments. 66ak2hx keystone multicore dsp+arm system-on-chips. http://www.ti.com/lit/ml/sprt651a/sprt651a.pdf,.

[36] Texas Instruments. DSP products website. http://www.ti.com/lsds/ti/dsp/overview.page,. Last accessed: Dec. 08, 2014.

[37] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173, Oct 2015.