Pattern-Aware Staging for Hybrid Memory Systems

Eishi Arima^{1,2(✉)} and Martin Schulz²

¹ The University of Tokyo, Tokyo, Japan
arima@cc.u-tokyo.ac.jp

² Technical University of Munich, Munich, Germany
schulzm@in.tum.de

Abstract. The ever increasing demand for higher memory performance and—at the same time—larger memory capacity is leading the industry towards hybrid main memory designs, i.e., memory systems that consist of multiple different memory technologies. This trend, however, naturally leads to one important question: how can we efficiently utilize such hybrid memories? Our paper proposes a software-based approach to solve this challenge by deploying a *pattern-aware* staging technique. Our work is based on the following observations: (a) the high-bandwidth fast memory outperforms the large memory for memory intensive tasks; (b) but those tasks can run for much longer than a bulk data copy to/from the fast memory, especially when the access pattern is more *irregular/sparse*. We exploit these observations by applying the following staging technique *if the accesses are irregular and sparse*: (1) copying a chunk (few GB of sequential data) from large to fast memory; (2) performing a memory intensive task on the chunk; and (3) writing it back to the large memory. To check the *regularity/sparseness* of the accesses *at runtime* with negligible performance impact, we develop a lightweight pattern detection mechanism using a *helper threading* inspired approach with two different **Bloom filters**. Our case study using various scientific codes on a real system shows that our approach achieves significant speed-ups compared to executions with using only the large memory or hardware caching: 3× or 41% speedups in the best, respectively.

1 Introduction

The performance of future computing systems relies less and less on computational power, but directly depends on both memory performance and capacity [26, 29]. At the same time, classical DRAM technologies are at risk in scaling bandwidth/capacity, and thus systems built solely on them will face severe limitations [29]. In order to counteract these trends, new and promising technologies, such as 3D stacking, HMC [9] or HBM [18, 23], have been developed, but face limitations in terms of capacity and scalability [23]. Therefore, to increase the memory capacity, DIMM-based off-package memories including NVRAM, such as Intel’s 3D XPoint memory [16], are still needed, but also face limitations, this

---

The original version of this chapter was revised: The chapter was made open access. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-50743-5_28

© The Author(s) 2020, corrected publication 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 474–495, 2020.
https://doi.org/10.1007/978-3-030-50743-5_24

Fig. 1. Concept of our staging technique

Fig. 2. Target architecture

time in terms of bandwidth-scalability due to power constraints on the memory bus/modules [8] and the number of off-package pins [32]. Driven by these diverging observations, adopting hybrid memory architectures, which combine different memory technologies on a single node, is an important design option for next generation computing systems from supercomputers to main stream systems [15, 17, 18, 34, 35].

While such hybrid memory systems have the potential to improve the performance of memory intensive applications, it is still unclear how to exploit—at the same time—both the available performance and the capacity on such hybrid memory systems. As an answer to this open question, we propose a software-based *pattern-aware staging* technique. Our core concept follows the fundamental observations¹ demonstrated in Fig. 1: (a) the high-bandwidth fast memory outperforms the large memory for the memory intensive *random* updates task, but (b) it takes a much longer time than the *sequential* copy tasks.

We exploit these observations to accelerate memory intensive tasks by using the staging technique shown in the figure *if the accesses are irregular and sparse*: (1) copying a large chunk of data from large to fast memory, (2) performing accesses on the chunk, and (3) writing it back to the large memory. We apply this technique when the data footprint is larger than the fast memory. In this technique, the data is divided into chunks of a few GB, and the staged access is, in turn, applied to each of them. Several recent studies also focus on the data managements for hybrid memory systems [3, 6, 11, 21, 27, 36, 38], but none of them *exploits this large performance impact of the access pattern to improve software-based data placement decisions at runtime*.

To successfully enable our pattern-aware staging technique, we need to detect when it is profitable to apply. For this, we propose a lightweight software-based mechanism that dynamically samples small parts of the access sequence, analyzes the access pattern in terms of regularity/sparseness, and then decides—at runtime—whether to apply staging or not. More specifically, we sample addresses using our new mechanism inspired by *helper threading* [19, 25], and then we efficiently characterize the pattern based on two different detectors implemented using *Bloom filters*: a Page Address Filter (PAF) for sparseness and a Stride

¹ This experiment was performed on a real system described in Sect. 6. The same number/size of memory references are issued for both the random/copy tasks (details are provided in Sect. 2.3).

Filter (SF) for regularity analysis. Finally, we propose a quantitative scheme to detect if an application can likely benefit from staging or not.

The followings are the major contributions of this paper:

- We focus on the observations regarding the impact of access pattern on the effectiveness of staging.
- Based on the above observations, we propose a software-based data management scheme called *pattern-aware staging*.
- We propose a simple dynamic address sampling mechanism inspired by software-based helper threading [19,25].
- We realize a lightweight pattern characterization scheme using two different small Bloom filters: PAF and SF.
- We propose a quantitative approach to make a decision based on the outputs of the above access pattern analysis.
- Finally, we evaluate our pattern-aware staging approach on a real system using scientific kernels.

2 Staging Accesses in Hybrid Memory

Figure 2 illustrates the target architecture of this study: the processor has multiple separated memory controllers, each of which is connected to a set of memories: one consists of fast but small memories; while the other one consists of large but slow memories. Looking forward, this kind of architecture is not only considered indispensable for any next generation high performance computing systems, covering exascale supercomputers and beyond [34,35], but is also poised to find its way into mainstream systems [18]. One example of this is installing both high-bandwidth 3D-stacked DRAMs (e.g., HBM [18,23] or HMC [9]) and conventional DDR modules in one compute board, which is supported in recent products such as Intel Knights Landing (KNL) processors [17] and Intel Agilex SoCs [15], and will be so in the future systems [18,35]. Another example is integrating both DRAM and NVRAM modules into DIMM slots, which is supported in Intel Cascade Lake processors [16]. In general, they are *heterogeneous in terms of bandwidth* as the 3D-stacked DRAMs can offer higher bandwidth scalability [9,17,18,23], and the bandwidth of NVRAM is limited [16] due to the significant memory access overheads [11,38].

2.1 Concept of Memory Staging

The goal of this research is to provide an easy to use way for *memory consuming/intensive applications* to exploit the performance of the fast memory, while also being able to utilize the capacity of the large memory in hybrid memory systems. In particular, to enjoy the bandwidth heterogeneity, we target memory-intensive multi-threaded applications with high instruction/data/thread-level parallelisms, which thus can become *bandwidth-limited*.

Fig. 3. Overview of staging

Fig. 4. Performance boost

Fig. 5. Copy overhead

To achieve this goal, we aim at utilizing coarse-grained data transfers/copies (data chunks in the order of GBs). This is because (1) over GBs of memory space is already available even in the fast memory, (2) accessing a large enough chunk is essential to exploit the bandwidth in the fast memory, and (3) we can allocate larger pages for larger chunks to mitigate the virtual/physical address translation overhead. As few applications naturally expose such coarse-grained accesses, we revisit the concept of *access staging* and adapt and extend it for managing data in hybrid memory systems.

Figure 3 illustrates an overview. First, we reserve a buffer (up to a few GB) in the fast memory and divide the large data, still stored in the slow memory, into the several data chunks matching the buffer size in the fast memory. For each data chunk, we then apply data staging as follows: (1) copy the data from the large memory to the fast memory, (2) perform bandwidth-critical tasks in the fast memory, and (3) return the data to the large memory by copying it back. We then iterate this process across all data chunks, until all chunks are processed.

In this work, we purposely do not consider overlapping or pipelining between the different stages of processing consecutive data chunks. *The detailed reasons behind this will be discussed in Sect. 8.*

2.2 Balancing Performance Boost and Overhead

To achieve performance improvements, we must apply our staging technique only when the performance boost gained in the second stage (T_{boost}) is larger than the copy overhead caused by the first and third stages (T_{copy}). These overheads can be formulated by using the parameters shown in Fig. 3. Here, T_{base} represents the execution time without staging, while T_{1st} , T_{2nd} and T_{3rd} represent the execution time of the first, second, and third stages in the staging technique, respectively. We can obtain a performance improvement when these times meet the following condition: $T_{base} > T_{1st} + T_{2nd} + T_{3rd} \Leftrightarrow T_{boost}(= T_{base} - T_{2nd}) > T_{copy}(= T_{1st} + T_{3rd})$. These times, however, depend on the characteristics of the *memory access patterns* in the targeted code or algorithm, which we need to carefully consider when determining whether we apply the staging or not. Further, to reduce the copy overhead, in certain cases we can *remove* the first stage or third stage in our approach. More specifically, we remove the third stage (writing back a chunk to the large memory) for read only tasks. Likewise, we remove the

first stage (reading a chunk from the large memory) for write only tasks such as overwriting temporary arrays.

2.3 Tradeoff Observations

In Fig. 4, we quantify the performance boost (T_{boost}) by comparing T_{2nd} and T_{base} . For this evaluation, we utilized a real hybrid memory system whose details are shown in Sect. 6. The vertical axis shows the execution time that is divided by the data size, i.e., the inverse of bandwidth. In this evaluation we analyze the performance boost for two different access patterns. For *random* we performed one billion random memory accesses on an 8 GB data array whose data element size is eight bytes; for *sequential* we examined sequential memory references on the same 8 GB data array, also by issuing one billion memory references.

As shown in the figure, the fast memory outperforms the large memory for both tasks. This is because the former has significantly more parallelism in ranks/banks/channels than the latter, and thus can provide data much faster regardless of access patterns if the accesses are intensive.

On the other hand, the random access pattern takes much longer to complete than the sequential one, which is a well-known phenomenon [14] happens also in NVRAMs [16, 38], and hence T_{boost} has to become much longer for the former. This is caused by the fundamental fact that memory systems are optimized in a way that they can exploit the bandwidth for sequential accesses by interleaving data across banks/ranks/channels [5], while utilizing open page policies [20]. Therefore, more *irregular* patterns cause more bank/rank/channel level conflicts [33]. Further, such accesses are very *sparse* (and hence come with very low locality) and thus these contentions can occur very frequently as, under such conditions, on-chip caches cannot help with reducing the number of accesses to memory.

Figure 5 represents the copy overhead (T_{copy}) between the two different memories. By comparing Fig. 4 and Fig. 5, we find that the significance of the copy overhead depends on the access types of the codes. As shown in the figures, it is better to move data for the random access pattern ($T_{boost} > T_{copy}$), but we should not do so for the sequential accesses ($T_{boost} < T_{copy}$). *Note that this pattern-aware comparison is universally valid for any hybrid memories—the application to other systems will be discussed in Sect. 8.*

3 Pattern-Aware Staging

Following the insights in the last section, we developed a lightweight software mechanism called ***pattern-aware staging*** that dynamically detects access patterns and decides on the fly whether to apply data staging or not. Figure 6 shows the overview with block diagrams. By a statical source-to-source transformation, the following functionalities are augmented into the original code as well as the staging: sampling the access sequence for a chunk just before executing the task, characterizing the pattern, and then using this information to make a decision

Fig. 6. Block diagram for Pattern-Aware Staging

Fig. 7. Overall strategy of pattern analysis

on whether we use the staging or not, i.e., we make a pattern-aware decision. The time and memory overhead of this analysis part has to be small enough in order for this scheme to be effective. We achieve this by (1) limiting the number of samples obtained, (2) parallelizing the sampling across multiple threads, and (3) using a filter-based efficient pattern-analysis, as described below. We perform this analysis *at runtime* as it is both more convenient for the user and more flexible to adapt to varying application behavior such as *input dependencies* than performing a static, offline based pattern-analysis. Consequently, *no profile from a previous run is needed for the application of our method*.

Figure 7 describes the concepts behind our pattern analysis component, which consists of three parts: sampling, characterization and decision. Each *Sampling Thread* in the figure acquires a part of the address sequence and analyzes the pattern *at runtime*. For this we use two separate detectors in the form of (Bloom) filters—a Page Address Filter (PAF) and a Stride Filter (SF)—as indicators. These filters keep the recent history of inputs (page-addresses/access-strides) and can thereby provide an answer on whether an input page-address/access-stride exists in the recent access history or not. A **low hit rate in the PAF** indicates low data locality, and thus a **sparse** access pattern. Additionally, a **low hit rate in the SF** indicates that accesses are **irregular**. More specifically, when accesses are more regular, the number of different access strides detected in the SF decreases and hence hits in the SF increase. For example, for an access pattern with only one constant stride, the SF only has one entry and shows a hit for all access except for the initial one.

After completing the sampling, we collect the hit/miss records of these two filters using a reduction operation and with that complete the characterization part. Based on the obtained statistics, we then make a decision based on the following observation: if the accesses are *sparse* and *irregular*, the task is likely to take much longer time than the copy and thus the performance boost brought by data staging will be larger and hence worthwhile.

```

for (i=0; i<num_chunks; i++){
  //processing ith chunk
  #pragma our_directive target(A[])
  for (j=0; j<M; j++){
    for (k=0; k<N; k++){
      A[i][(j*I[k])%L] += 1;
    }
  }
}

```

Fig. 8. Original code + our new directive

```

for (j=0; j<M; j++){
  for (k=0; k<N; k++){
    // Load(&A[i][(j*I[k])%L]); //HT
    PAF.input(&A[i][(j*I[k])%L]);
    SF.input(&A[i][(j*I[k])%L]);
    [Abort when # of sampling
     exceeds threshold]
  }
}

```

Fig. 9. Sampling thread code

```

for (i=0; i<num_chunks; i++){
  //processing ith chunk
  [Put the sampling thread code here (inline)]
  if (decision_making(args)){
    [code with staging]
  } else {
    [code without staging]
  }
}

```

Fig. 10. Pattern-aware staging code

4 Sampling and Characterization

4.1 Sampling Threads

Figure 8 represents a sample pseudo code to apply our pattern-aware staging technique². In this figure, the two-dimensional array ($A[\text{num\_chunks}][L]$) can be divided into chunks, and the outermost for loop then selects one of them turn by turn. The 3rd line in the figure shows our newly introduced directive to specify the target array to apply our technique to. Here, we assume the following scenario: when a compiler comes across this directive, it automatically attempts to transform this original code into the pattern-aware staging code in Fig. 9 and 10 for the target array. Although the transformation is performed by hand in this paper, as in previous studies on compiler-based pre-execution or helper thread prefetching [19, 25], this can be automated using, e.g., a source-to-source compiler [24, 30], similar to previous software-based data management studies [22, 28, 31].

Next, we describe the sampling thread code³ in Fig. 9. This code can be considered a modified version of the inner two loops of Fig. 8. The 6th line, commented-out in Fig. 9, shows the original helper threading approach.

² Here, to simplify the explanation, we utilize the sequential code. But, our codes are actually parallelized with OpenMP in our evaluation.

³ In our evaluation, we parallelize the code as follows: (1) to abort OpenMP *parallel for* loops on the way, we utilize *cancel for* statement; (2) to minimize the communications among threads, we set the statistics of the filters as private variables and collect them using *atomic* statement just after the end of the samplings.

Instead of calculating $A[i][(j \cdot I[k]) \% L] += 1$, it prefetches data using the address $(\&A[i][(j \cdot I[k]) \% L])$, which is achieved by distilling the codes to execute only the address generation paths [19, 22, 25, 31]. Similar to this, our sampling mechanism just obtains the same address and utilizes it as an input for the filters (PAF and SF). Note that, if the array is accessed multiple times in the loop (e.g., unrolled loop), we add the filter input and increment the sample count accordingly. When the total number of sampled addresses exceeds a given threshold, we abort the loops and collect the statistics. Putting it all together, this sampling code is inlined at the 3rd line in Fig. 10, just before the decision making function *decision_making(args)*. While this direct inlining of the code could be optimized by spawning separate sampling threads and overlapping them with the main threads, we decided to avoid this extra complexity due to the negligible sampling and characterization overhead shown in Fig. 13 (Sect. 4.3).

4.2 Access Characterization

We characterize the sampled address sequence in terms of *sparseness* and *regularity* using the PAF/SF as described in Sect. 3. To realize this, these filters have to be *efficient in memory and time overheads*. For this reason we turn to Bloom filters, as they fulfill the requirements, as laid out below.

The Filter Mechanism. We assume each filter has three functionalities: *Test()*, *Set()*, and *Clear* as shown in Fig. 11. First, the *Clear* function is used to initialize/reset the contents of the filter. For each access, we use the *Test()* function to examine whether an incoming element x (page-address/stride for PAF/SF, respectively) is recorded in the filter or not. If it returns a hit, then the corresponding hit counter is incremented, otherwise the miss counter is incremented and *Set()* is called to register x in the filter to detect future accesses.

Bloom Filter Based Implementation. To implement PAF and SF, we utilize Bloom filter, which is a probabilistic data structure that can record a large set of elements with a small memory footprint [4]. Figure 12 shows their principle structure: it consists of a bit array, which stores the elements in the filter, and multiple hash functions, each of which returns an index to the bit array. At first, all of the bits are set to zero. Then, to register input elements (e.g., in our case page-addresses/strides for PAF/SF), we can use the *Set()* function to identify the bits associated with the input using the hash functions and then set them to one. We use the *Test()* function to extract the bits associated with an input element using an AND operation on the bits pointed to by the hash functions: it should return a hit (1) if an element was recorded before, otherwise a miss (0).

In the figure, *Test(x)* returns a hit because x was already registered (*True Positive*). The output of *Test(z)* is a miss, as z has not appeared, yet, at this point (*True Negative*). However, due to the hash collisions, *Test(w)* can return also a wrong answer: a hit for a non-registered element w (*False Positive*). Small numbers of false positives do not have a significant impact, but to avoid too frequent false positives, the size of the bit array must be chosen large enough. Thus, the memory overhead and the false positive probability are an important trade-off, which is further influenced by picking the right hash functions. Further,

Fig. 11. Filter operations

Fig. 12. Bloom filter mechanism

after recording a certain amount of records, the *Clear* function must be used to re-initialize the filter contents; otherwise the filter can be filled with positive values and always return hits.

4.3 Quantitative Analysis

We evaluate the overhead/effectiveness of our sampling and characterization approach using access patterns for various sparse matrices. The matrices are collected from the Florida sparse matrix collection [10] and are listed in Table 2. Assuming SpMV with CRS format [2], we use the column indices of each matrix as an index array to a vector and analyze the access patterns with using our sampling and characterization approach. For this evaluation, we use our hybrid memory system whose detailed configuration is shown in Sect. 6. The configurations for our sampling phase and the filters are summarized in Table 1.

Figure 13 compares the time overhead between 1 or 8 GB copy operations ($T_{1st} + T_{3rd}$) and our sampling and characterization approach. The X-axis indicates the sampled addresses for both PAF and SF in each thread, while the Y-axis represents the time overhead. For the sampling and characterization overhead, each value shows the average time with the standard deviation across workloads.

Fig. 13. Time overhead comparison

Fig. 14. Hit rates v.s. # of sampling

Table 1. Sampling and filter settings

Sampling	
# of samples/thread	2 K (1 K $\rightarrow$ 8 K in Fig. 13/14)
# of threads	64
Filters (PAF/SF)	
Size [B] ($=2^N/8$)	256 (64 $\rightarrow$ 512 in Fig. 15)
Trigger for <i>Clear</i>	256 unique inputs
# of hashes	2
$hash_k(x)$ ($k = 0, 1$)	$(x \gg (N * k)) \& (2^N - 1)$

Fig. 15. Hit rates v.s. filter size

 Fig. 16. Hit rates vs. μ ($\Delta = 0$ —fixed)

 Fig. 17. Hit rates vs. Δ ($\mu = 1K$ —fixed)

As shown in the figure, when we limit the number of sampled addresses to less than 8 K per thread, the overhead of our approach becomes quite small (less than 1%) compared with the few GB of round-trip copy operations. In particular, it takes just **0.025%** of time compared with a 8 GB copy at 1K samples.

Figure 14 shows how many sampled addresses are needed to obtain accurate enough PAF/SF hit rates. The X-axis shows the number of sampled addresses per thread, while the Y-axis represents the PAF/SF hit rates. Each line in the figure is associated with one of the matrices listed in Table 2. As the graph shows, the PAF/SF hit rates are almost constant when we sample more than 2 K/1 K addresses per thread. Based on this result, we limit ourselves to 2 K/1 K addresses per thread for the PAF/SF. The time overhead of this is less than **0.040%** compared to the 8 GB copy operations, as shown in Fig. 13.

Figure 15 presents the PAF/SF hit rates as a function of the filter size. We scale the filter size from 64 B to 512 B (512 bit to 4096 bit) per thread while fixing the maximum number of filter inputs as 256. As shown in the figure, as the filter size scales, the PAF/SF hit rates become smaller, i.e., fewer false positive happen. However, they are almost constant when the size is larger than 256 B. Based on this result, we choose 256 B for both PAF and SF.

Finally, Fig. 16 and 17 demonstrate how well our Bloom filter based detectors can represent the *sparseness/regularity* of memory accesses. In this evaluation, we examine a synthetic memory access code, in which the address of i th memory reference ($Addr_i$) is defined as follows: $Addr_i = Addr_{i-1} + \mu + URAND(-\Delta, \Delta)$ ($i > 0$). Namely, μ is the *average stride* of the accesses, which determines the *sparseness*, while $URAND(-\Delta, \Delta)$ is a *random noise* following a uniform distribution ranging from $-\Delta$ to Δ , thus affects the *regularity*. As shown in those figures, each filter is effective to sense the associated access feature.

Table 2. Selected matrices

Matrix name
2cubes_sphere, audikw_1, eu-2005, europe_osm, F1, FullChip, G_n_pin_pout, GL7d20, Hamrle3, hugebubbles-00020, HV15R, offshore, pkustk14, poisson3Db, pre2, rajat29, road_usa, scircuit, soc-sign-epinions, thermomech_dK, thermomech_dM, tmt_unsym, torso3, tx2010, wiki-Talk, wikipedia-20061104

5 To Stage or Not to Stage?

Figure 18 illustrates the overview of our strategy: on the R_{paf} - R_{sf} plane (PAF/SF hit rates), we consider the Break Even Line (BEL)—at any points on the line, the time reduction gained in the second stage (T_{boost}) is equal to the copy overhead time (T_{copy}). If the pattern feature vector (R_{paf}, R_{sf}) is mapped below the BEL on the plane, we can achieve speed-up with the staging, otherwise not. The BEL is formulated as follows: $T_{boost}(R_{paf}, R_{sf}, P) - T_{copy}(P) = 0$.

In addition to the pattern features, this function also utilizes additional input parameters (denoted through the set $P = \{R_{write}, R_{util}, P_{else}\}$), which help fine tune the shape of the BEL. The definitions of these parameters are listed in Table 3. $T_{boost}()$ (the performance gain) will be shorter if the chunk is less utilized (R_{util} is smaller), and it will also depend on the read/write access rate as read/write bandwidths are different in various memory systems. Furthermore, T_{1st}/T_{3rd} in $T_{copy}()$ can be skipped if the chunk is read- or write-only ($R_{write}=0$ or 1) as described in Sect. 2.2. These parameters can be collected at such as the code transformation time⁴.

5.1 Decision Criterium

First, we formulate $T_{copy}()$ [s/GB] as follows:

$$T_{copy}(P) = \alpha \cdot 1/B_{1st} + \beta \cdot 1/B_{3rd} = \alpha T_{1st} + \beta T_{3rd} \quad (1)$$

In the equation, B_{1st}/B_{3rd} and T_{1st}/T_{3rd} represent the copy bandwidth and the time per GB of the first/third stages, respectively (see also Sect. 2.2). Here, $\alpha = 0/\beta = 0$ stands when the write-/read-only case (namely, $R_{write} = 1/0$), otherwise we set $\alpha = 1/\beta = 1$, respectively. Note that $T_{copy}()$ does not depend on R_{paf} , R_{sf} , R_{util} , or others as it has nothing to do with how the chunk is accessed during the task except for R_{write} .

Second, we define $T_{boost}()$ [s/GB] (time per chunk size) as follows:

$$T_{boost}(R_{paf}, R_{sf}, P) = S(R_{util}, P_{else}) \cdot T'_{boost}(R_{paf}, R_{sf}, R_{write}) \quad (2)$$

Here, we divide $T_{boost}()$ into memory access pattern (or types) dependent/independent parts. $T'_{boost}()$ is a pattern dependent function, which can be regarded as the special case of $T_{boost}()$, namely when $S() = 1$ stands. $S()$ is a scaling factor, which is independent of the access pattern/types. In this paper, we utilize $S(R_{util}, P_{else}) = R_{util}$ assuming that a task takes N times longer when the access sequence also becomes N times longer with the same access pattern/types, which is generally the case. Further extensions of $S()$ will be discussed in Sect. 8.

⁴ When we convert memory references into filter inputs in a target loop (see Sect. 4.1), we can also count the number of loads/stores in the loop and thus can estimate the write rate (R_{write}). As for R_{util} , we assume the chunk size is designated by the programmers, but we need to know the number of iterations of the target loop to obtain the number of references. One option for doing this is, like the parallel for statement in OpenMP, limiting the applicable targets to the canonical form loops. Another option is providing a mechanism to predict it.

Fig. 18. Overview of decision making strategy

Table 3. Functions and parameters

Time Functions/Parameter [s/GB]	
$T_{boost}()$	Speed-up gained in the second stage
$T_{copy}()$	Time overhead of the copy operations
T_{th}	Threshold to set the decision aggressiveness
Pattern Features	
R_{paf}	Page Address Filter (PAF) hit rate [0:1]
R_{sf}	Stride Filter (SF) hit rate [0:1]
Additional parameters for fine tuning (P)	
R_{write}	# of writes per reference on the chunk [0:1]
R_{util}	# of references per element on the chunk
P_{else}	Empty set, reserved for further tuning

Then, we utilize the following linear approximation:

$$T'_{boost}(R_{psf}, R_{sf}, R_{write}) = R_{paf}C_0(R_{write}) + R_{sf}C_1(R_{write}) + C_2(R_{write}) \quad (3)$$

We determine the coefficients (C_i) by testing the following three patterns on each memory (fast/large) for a fixed R_{write} : (1) random accesses on a large enough array ($R_{paf} \simeq 0, R_{sf} \simeq 0$), (2) accesses with a long enough stride ($R_{paf} \simeq 0, R_{sf} \simeq 1$), and (3) sequential streaming accesses ($R_{paf} \simeq 1, R_{sf} \simeq 1$). By acquiring $T'_{boost}()$ with measurements for these patterns (here put as T_{rand} , T_{strd} , and T_{seq}) and solving the given linear equations, we can gain C_i for a fixed R_{write} .

We decide on whether to stage or not, based on these functions, combined with a threshold T_{th} . More specifically, we apply the staging if the following condition holds:

$$T_{boost}(R_{paf}, R_{sf}, P) - T_{copy}(P) > T_{th} \quad (4)$$

When T_{th} is set lower/higher, the staging is applied more aggressively/conservatively, respectively. We assume the parameter is predetermined, but as an option, this should also be controllable by users depending on their confidence.

5.2 Accuracy Analysis

We evaluate the accuracy of our staging criteria using synthetic workloads. The system/coefficients setups will be described in Sect. 6, and the sampling thread settings are based on the evaluation in Sect. 4.3. We apply our staging technique to the source vectors in SpMV operations (CRS format) whose matrices are listed in Table 2 in Sect. 4.3. In this evaluation, we utilize multiple vectors and organize a chunk by using consecutive vectors. The number of vectors is set so that the total data size becomes around 90 GB. Also, we scale the number of rows of the matrices from 1 to 1/32 to change the chunk utilization (R_{util}).

Figure 19 demonstrates the performance impact of false decisions. The horizontal axis represents workload number, while the vertical axis indicates relative performance, which is normalized to that of *Large Mem Only* (the pure large memory only solution). The workloads appear in the left side of the figure have smaller R_{util} but higher R_{sf} and R_{paf} —chunks are less utilized and more regularly accessed with higher locality. In this graph, the threshold parameter T_{th}

Fig. 19. Impact of false decisions

Fig. 20. Correctness of decisions

is set to 0. In the figure, *Always Staging* means the staging is always applied regardless of the access features.

According to the figure, the false decisions (*False Positive/Negative* in the figure) occur more often when the performance impact of decision makings is less significant (*Always Staging* and *Large Mem Only* are closer), which is a preferable feature for our approach. This is because (1) our approach basically compares T_{boost} and T_{copy} , which is equal to comparing the performance of *Always Staging* and *Large Mem Only* as $|T_{boost} - T_{copy}| = |(T_{1st} + T_{2nd} + T_{3rd}) - T_{base}|$ (see also Sect. 2.2); and thus (2) this comparison becomes more error tolerant when the performance difference of the two approaches becomes larger.

Figure 20 shows the breakdown of decision types as a function of T_{th}/T_{copy} (T_{th} : the threshold parameter used in decisions). In the figure, “True” means the decision is correct, and “Positive” represents the staging is conducted—the equation $T_{boost}() - T_{copy}() > T_{th}$ is expected to stand. As shown in the figure, 79% of the decisions are correct (“True Positive/Negative”) at $T_{th}/T_{copy} = 0$. We can trade-off “False Positive” and “False Negative” by changing the threshold T_{th} . According to the figure, scaling T_{th}/T_{copy} from 0 to 1 has no significant impact on the decision accuracy, allowing users to freely choose the right tradeoff. To balance false positives/negatives, we choose 0.5 in Sect. 6 and 7.

6 Evaluation Setup

Table 4 summarizes the environment for our experiments. We utilize a KNL-based system whose nodes provide a hybrid memory system [17]. The fast memory in the system supports both software-based scratch pad mode (*Flat*) and hardware-based data management (*Cache*), and we choose the former for our approach. The operating system used for the evaluation is Cent OS 7 and we use Intel C/C++ compiler (ICC) with the listed options. The sampling thread settings are based on the evaluation in Sect. 4.3, and the threshold parameter T_{th}/T_{copy} is set to 0.5. Through this evaluation we set the number of threads to 256 for all of the applications. In our implementation, a 16 GB buffer is allocated to the fast memory using the memkind library [7], which is designed to use different kinds of memories in a node.

6.1 Coefficients Calibration

Before applying our approach, we have to correctly set the coefficients described in Sect. 5.1: T_{1st} , T_{3rd} , T_{rand} , T_{strd} and T_{seq} . Here, we summarize how to acquire them. First, to obtain T_{rand} , T_{strd} and T_{seq} for a given R_{write} , we measure the bandwidth of the following tasks on both fast and large memories: (1) 1G times random accesses on an 8 GB array, (2) 2 M times stride accesses on 8 GB array (4K + B stride) and (3) a streaming task on 8 GB array. In this paper, the measurements are performed for $R_{write} = \{0, 0.5, 1\}$ by changing the rate of load/store operations in the main loop of the test tasks. Second, to determine T_{1st} and T_{3rd} , we just measure the copy bandwidth between the memories.

6.2 Implementation and Workloads

Our proposal is implemented manually in each application, following the example of various published studies of software-based data management [19, 25]. In this evaluation, to represent widely used kernels for a range of applications especially in scientific computing, we choose various benchmarks from HPC Challenge (HPCC), NAS Parallel Benchmarks (NPB), and also use stencil codes (Jacobi2D/3D). The followings are the details:

RandomAccess (HPCC): This application randomly updates a big table. We repeat the main update loop multiple times, and, in the loop, we filter the update accesses: only the accesses to a target area (chunk) pass the filter [14]. By doing so, we can restrain the accesses within the buffer in the fast memory and, at the same time, can conduct all the update accesses. Note that we apply this to all methods that we compare. In this evaluation, the total table size and the chunk size are set to 64 GB and 16 GB, respectively.

PTRANS (HPCC): This application transposes a matrix and adds it to another ($T+ = A^T$). These matrices are dividable into sub-matrices (chunks), and we apply our technique to the source matrix A , which is accessed with a long stride. In this evaluation, the total size of the matrices, and the chunk size are 96 GB (=48 GB $\times$ 2) and 16 GB, respectively

Table 4. Evaluation environment

Name	Remarks
CPU	XeonPhi 7210, 64cores, 1.3 GHz, quadrant mode
Memory	Fast Memory: MCDRAM, 8ch, <u>450 GB/s</u> , 100 + ns, 16 GB, flat/cache mode Large Memory: DDR4 DRAM, 2ch, 90 GB/s, 100 + ns, <u>96 GB</u> Measured Copy Bandwidth: 79.7 [GB/s] (1st stage), 42.1 [GB/s] (3rd stage)
OS	CentOS 7
Compiler	ICC 19.0, options: -O3, -qopenmp, -lmemkind, -xMIC-AVX512

FFT (HPCC): This workload calculates one dimensional FFT using two 32 GB arrays: input and output array. We apply our staging technique to the output array by dividing it into $4\text{ GB} \times 8$ chunks. Through the evaluation, a temporal array is located at the fast memory.

STREAM (HPCC): In this workload, a simple vector operation $dst = src$ is performed, and we applied our method to the destination vector dst . In the evaluation, the total data size is around 96 GB, and the chunk size is 16 GB.

Jacobi2D/3D: We utilize 5/7-point 2D/3D Jacobi stencil codes. In these codes, we keep the results of all time steps to different arrays (= chunks). We apply our technique to the source array, which is heavily loaded in the stencil operations. In this evaluation, the chunk size is set to 8 GB (the array size for one time step), and the total data size is 80 GB.

IntegerSort (NPB): This workload sorts an integer array by counting the distribution of the elements (bucket sort). Our approach is applied to the array of the distribution (16 GB = the chunk size). The total data size is around 64 GB.

ConjugateGradient(NPB): In this kernel, we focus on the iterative SpMV operations, as it is the major performance bottleneck. We apply our technique to the source vector for the SpMV operations whose size is 2 GB (= the chunk size). The total data size is 90 GB, which includes multiple different vectors.

6.3 Compared Methods

For the above workloads, we compare the performance of the following methods:

LO: The execution with the Large memory Only (baseline).

NP: The execution with a Numactl command with Preferred option, which preferentially stores data on the fast memory [17].

HC: The fast memory works as a direct mapped Hardware Cache [17].

PS: The execution with our Pattern-aware Staging.

7 Experimental Result

Figure 21 compares the performance among the methods across all applications. The vertical axis indicates relative performance that is normalized to *LO* for each application. **GeometricMean** in the figure shows the geometric mean of performance across all the workloads for each method. Our method (*PS*) achieves a factor of three performance improvement over *LO* at the best case and on average, it improves performance by a factor of 1.9. As the data management policy of *NP* is naive, it does not improve performance except for **STREAM**. Compared to hardware cache (*HC*), our approach has the following benefits: (1) ours purposely puts the useful chunk of data on the fast memory based on the pattern-analysis thus can avoid unnecessary allocations/conflicts on it; (2) ours can fully utilize the hardware resources of the fast memory, but the hardware cache wastes the available bandwidth/storage due to the hardware overheads such as tags. Thanks

Fig. 21. Performance comparison among methods across applications

Fig. 22. Comparison of memory access traffic

to these characteristics, our *PS* outperforms *HC* for almost all workloads in this evaluation (up to 41%).

Figure 22 demonstrates the memory access traffic (or bandwidth consumption) on the two different memories. The measurement was accomplished through Intel PCM, a well-known performance monitoring tool. Although our approach increases the data traffic compared with *LO/NP* due to the additional data transfer between the memories, it reduces 36% of the traffic compared with *HC* on average. This is because *HC* induces unnecessary data conflicts on the fast memory, while ours not as described later. This traffic reduction will lead to a considerable power reduction on the memory system as a consequence.

One exception in Fig. 21 is *ConjugateGradient* (the hardware cache works better than ours), and we can see the reason in Fig. 23: in the figure, the X-axis represents the total data size, while the Y-axis indicates the relative performance normalized to *LO* at 16 GB. When the data footprint size is small enough, the hardware cache approach can keep almost all the useful data on the fast memory, thus it works well. However, as we scale the data size, more conflicts happen on the fast memory, which degrades performance significantly. In contrast to this, ours can explicitly hold the useful data without conflicts on the fast memory no matter how much we scale the total footprint. Therefore, if we would scale the data size more (over 96 GB, the capacity limitation in our system), ours would work better than the hardware cache for this workload.

Finally, we summarize the statistics of our approach in Table 5. The pattern features (R_{paf} , R_{sf}) are taken from our sampling technique, while the other

Table 5. Statistics of our pattern-aware staging approach

Name	R_{paf}	R_{sf}	R_{util}	R_{write}	Estimated $T_{boost}/T_{copy} - 1.5$	Measured $T_{boost}/T_{copy} - 1$	Decision Correctness
RandomAccess	0.0388	0.0620	8	0.5	22.4 (>0)	20.3 (>0)	Correct
PTRANS	0.00297	0.999	1	0	6.10 (>0)	3.73 (>0)	Correct
FFT	0.968	0.998	4.5	1	5.04 (>0)	9.08 (>0)	Correct
Jacobi2D	0.998	0.998	5	0	3.01 (>0)	1.67 (>0)	Correct
Jacobi3D	0.996	0.998	7	0	4.87 (>0)	2.18 (>0)	Correct
IntegerSort	0.691	0.687	6	0.5	10.2 (>0)	10.6 (>0)	Correct
STREAM	0.998	0.999	1	1	-0.315 (<0)	-0.331 (<0)	Correct
ConjugateGradient	0.368	0.0947	7	0	31.3 (>0)	17.8 (>0)	Correct

Fig. 23. Performance vs. data size (ConjugateGradient) **Fig. 24.** Performance/overhead comparison

parameters (R_{util} , R_{write}) are acquired by manually counting the number of load/store instructions to the target array in the target loop. As for the decision makings, if $T_{boost} - T_{copy} - T_{th}$ is greater than zero, which is equal to $T_{boost}/T_{copy} - 1.5 > 0$ in our T_{th} setting, we use the staging technique; otherwise not (see also Eq. (4) in Sect. 5.1). From this point of view, *as long as the signs of the estimated $T_{boost}/T_{copy} - 1.5$ and the measured $T_{boost}/T_{copy} - 1$ are the same, our approach is correct*, and our approach is correct for all the workloads. Note that by adjusting T_{th} based on the observation in Sect. 5.2, our approach successfully avoids slow down for **STREAM** unlike **HC**.

8 Discussions

Applicability of the Approach: The most significant restriction to apply our approach to a kernel is that the data structure of a potential target array has to be transformable into a multi-dimensional array form (e.g., into a matrix or SoA). Note that several access optimization approaches, such as multi-pass gather/scatter [14], are useful to meet this requirement. For multi-dimensional arrays, we can choose a chunk by designating the indices for the higher dimensions, and at the same time, we can ensure the size of the chunk and the area of the accesses. After copying the chunk to the fast memory, the pointer to the data is replaced to go to the fast memory, while the remaining data stays in the large memory. With this, any complicated access pattern that includes

accesses to both inside/outside of the chunked area are handled correctly, but the performance gain will be less if too many accesses don't hit the target chunk.

We assume the indices to choose the target chunk are manually assigned by the programmer just before the target loop by using a specific function to set them. However, it is not always easy for the programmer to set the right indices. One promising option to cope with this issue is providing a functionality to automatically choose the chunk that is most likely to be intensively accessed. We can support this option by extending our sampling and characterization approach to include additional filters to store the indices.

Our approach is applicable regardless of the number of arrays the target kernel accesses. In this work, we assume the programmer chooses one array by designating the variable in the directive shown in Fig. 8, and then the compiler generates a distilled version of the code that executes only the address generation path for the target while ignoring the others, which relies on the prior helper threading works [19, 22, 25, 31]. However, our approach is extensible to multiple arrays: (1) listing them in the directive; (2) creating the address generation paths for all the targets; and (3) storing the addresses in their own unique filters separately. To this end, the decision making part needs several modifications (in both the decision function and the control structure after it).

When multiple different array/pointer variables are used in a kernel, pointer aliasing can potentially happen, i.e., different variables point to the same memory. Namely, even if a chunk is moved to the fast memory for a variable, another pointer may point to the old data stored on the large memory. One option to cope with this is applying our technique only when the programmer specifies that they are free from the aliasing by putting a keyword like *restrict* supported in C99. Such a keyword is widely utilized to allow compilers aggressive optimizations, and our approach can be considered one of them in a broad view.

Overlapping and Pipelining: Pipelining is a well-known technique to hide the communication latency between components/nodes by overlapping computation and data transfer [28]. In our case, the second stage for one chunk and the first/third copy stages of other chunks can be overlapped (see also Sect. 2). However, we purposely do not consider this optimization in our approach due to *significant hardware contention* on the fast memory, as all of the stages access it intensively for *memory intensive tasks*.

We quantify the impact of the contention using the same environment and workloads as in Sect. 2.3, which clarifies that the performance benefit of overlapping is limited or even harmful (Fig. 24)⁵. This is due to the following reasons: the overlapping does not reduce the amount of traffic on the memory subsystem;

⁵ * For “Staging w/ Overlap”, we refer to the contention overhead as $C \times T_{copy}$, i.e., this approach is beneficial only when both $C < 1$ and $T_{boost} > CT_{copy}$ stand, which is **not the case in the figure: neither (a) nor (b)**.

* “Ideal” or “Staging w/o Overlap” are executed by 64 threads, while for “Staging w/ Overlap”, additional 64 copy threads also run in parallel and are distributed to all 64 cores to balance the loads. The contention in core resources does not matter as the memory is the bottleneck.

it can cause more conflicts on the memory resources (e.g., at row buffers [38]) for case (a) ; and the copy time is too large to hide for case (b).

Interaction with Hardware Caching: In our evaluation, when applying our technique, we utilized the fast memory as a scratchpad region instead of a hardware cache. This is because the major benefit of our technique is *selectively* allocating a useful chunk on the fast memory, which should be *conflict free*, but the cache mode evicts data placed on the fast memory by automatically allocating the others (even more so for larger data, as demonstrated in Fig. 23).

Application to Other Platforms: Our methodology is applicable to any hybrid memory systems including the configuration of DRAM+NVRAM [16, 38]. This is because ours is based on the fundamental architectural principle: memories are optimized and thus operate significantly faster for sequential accesses [5, 16, 20, 33, 38] regardless of the memory cell implementation. Based on the above, our decision criterion estimates the impact of access pattern/types using several system-dependent coefficients. Thus, what we have to do when applying ours to different platforms is just updating the coefficients, i.e., the calibration process performed in Sect. 6.1, which is *needed only once for a system*.

Automation: Although we quantify the effectiveness of our proposal, some parts, such as the sampling and the staging, are hand coded. In future work, we will automate them in the compilers/runtime tool chain such as LLVM [24] or the ROSE compiler [30], similar to previous software-based data management studies [22, 28, 31]. For this automation, our approach needs to obtain some parameters (P) at the code generation time or by using augmented codes at runtime as described in Sect. 5 (see the footnote). As for the staging part, existing compiler techniques to apply pipelining to CPU-GPU systems will be useful [28]. In addition, acquiring more parameters (P_{else}) at compilation or runtime and updating the scaling function $S()$ accordingly is a promising direction to cover more aspects in the decision making. One example for this is counting floating operations and memory access instructions in the target loops, calculating the arithmetic intensity based on the results, and tuning $S()$ following our existing models.

9 Related Work

Since hybrid memory systems have become a significant design choice recently, various software-based data placement techniques for them have been proposed. Due to their limited availabilities, we couldn't compare our approach with them quantitatively in the evaluation. However, our technique qualitatively has the following uniqueness/benefit compared with them: (1) ours does not require any application profiles; and thus (2) ours can detect the pattern of both input-dependent/independent memory accesses well, while the others cannot. Especially, when the pattern heavily depends on the input such as the problem set-

tings, which is often the case for the scientific computing, our runtime pattern analysis approach becomes essential.

Data Tiering API provides a memory allocation interface that optimizes the page allocations automatically, but the decisions are based on the application statistics that depend on the inputs [13]. Unimem API provides a similar memory allocation interface and optimizes the placements at the granularity of data objects. However, it does not target the chunking except for sequential accesses [36]. A prior study proposed a compiler-based technique that attempts to optimize the initial data allocations, but it does not handle the data transfer and relies on a statical analysis [21]. Some runtime-based approaches target different programming model, such as task parallel programming [1, 37], which is out of our scope. Other studies focus on application specific solutions [6, 27], but ours aims at covering general applications. OS/HW-level page managements have been widely studied for hybrid memory systems, but they require hardware modifications [11, 38]. A recently proposed page scheduler does not require such hardware, but needs a large number of profiles to work [12].

10 Conclusions

This paper proposed and made a case for a software-based data management technique called pattern-aware staging to exploit both the high performance and the large capacity components of hybrid main memory systems. Our technique dynamically examines the pattern of memory accesses and, in case of irregular/sparse patterns, fetches chunks of data from large memories to fast memories, just before they are referenced. The experimental results using scientific codes on a real system show that our approach enables 300% improvements compared to using only large memory and still 41% compared to hardware caching.

Acknowledgments. We are grateful to all the reviewers of ISC’20, CGO’20, ICS’19, and IPDPS’19 for their valuable suggestions. In particular, we would like to express our sincere gratitude to our shepherd in ISC’20 for helping us with completing this work. Also, we would like to thank folks in CAPS at TU Munich, ITC at U Tokyo, and CASC at LLNL for their helpful comments to carry out this study. This work is partly supported by JSPS Grant-in-Aid for Research Activity Start-up (JP16H06677), JSPS Grant-in-Aid for Early-Career Scientists (JP18K18021), and Research on Processor Architecture, Power Management, System Software and Numerical Libraries for the Post K Computer System of RIKEN.

References

1. Alvarez, L., et al.: Runtime-guided management of stacked DRAM memories in task parallel programs. In: ICS, pp. 218–228 (2018)
2. Bell, N., et al.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC, pp. 18:1–18:11 (2009)
3. Benoit, A., et al.: A performance model to execute workflows on high-bandwidth-memory architectures. In: ICPP, pp. 36:1–36:10 (2018)

4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13**(7), 422–426 (1970)
5. Burnett, G.J., et al.: A study of interleaved memory systems. In: *AFIPS 1970 (Spring)*, pp. 467–474 (1970)
6. Butcher, N., et al.: Optimizing for KNL usage modes when data doesn't fit in MCDRAM. In: *ICPP*, pp. 37:1–37:10 (2018)
7. Cantalupo, C., et al.: User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies (2015)
8. Chatterjee, N., et al.: Architecting an energy-efficient dram system for GPUs. In: *HPCA*, pp. 73–84 (2017)
9. Consortium, H.M.C.: Hybrid Memory Cube Specification 2.1. Last Revision January (2015)
10. Davis, T.A., et al.: The university of Florida sparse matrix collection. *ACM TOMS* **38**(1), 1:1–1:25 (2011)
11. Dhiman, G., et al.: PDRAM: a Hybrid PRAM and DRAM main memory system. In: *DAC*, pp. 664–669 (2009)
12. Doudali, T.D., et al.: Kleio: a hybrid memory page scheduler with machine intelligence. In: *HPDC*, pp. 37–48 (2019)
13. Dullloor, S.R., et al.: Data tiering in heterogeneous memory systems. In: *EuroSys* (2016)
14. He, B., et al.: Efficient gather and scatter operations on graphics processors. In: *SC*, pp. 1–12 (2007)
15. Intel: Intel®Agilex™FPGA Advanced Information Brief: (Device Overview). INTEL (2019)
16. Izraelevitz, J., et al.: Basic performance measurements of the intel Optane DC persistent memory module. *arXiv preprint [arXiv:1903.05714](https://arxiv.org/abs/1903.05714)* (2019)
17. Jeffers, J., et al.: Intel Xeon Phi Processor High Performance Programming: Knights, Landing edn. Morgan Kaufmann Publishers Inc., San Francisco (2016)
18. Jun, H.: HBM (High Bandwidth Memory) for 2.5D. SEMICON Taiwan (2015)
19. Kamruzzaman, M., et al.: Inter-core prefetching for multicore processors using migrating helper threads. In: *ASPLOS*, pp. 393–404 (2011)
20. Kaseridis, D., et al.: Minimalist open-page: a DRAM page-mode scheduling policy for the many-core era. In: *MICRO*, pp. 24–35 (2011)
21. Khaldi, D., et al.: Towards automatic HBM allocation using LLVM: a case study with knights landing. In: *LLVM-HPC 2016*, pp. 12–20 (2016)
22. Kim, D., et al.: Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In: *CGO*, p. 27 (2004)
23. Kim, J., et al.: HBM: memory solution for bandwidth-hungry processors. In: *Hot Chips 26 Symposium (HCS)*, pp. 1–24 (2014)
24. Lattner, C., et al.: LLVM: a compilation framework for lifelong program analysis & transformation. In: *CGO*, p. 75 (2004)
25. Lee, J., et al.: Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE TPDS* **20**(9), 1309–1324 (2009)
26. Matsuoka, S., et al.: From FLOPS to BYTES: disruptive change in high-performance computing towards the post-moore eera. In: *CF*, pp. 274–281 (2016)
27. Miao, H., et al.: StreamBox-HBM: stream analytics on high bandwidth hybrid memory. In: *ASPLOS*, pp. 167–181 (2019)
28. Mokhtari, R., et al.: BigKernel - high performance CPU-GPU communication pipelining for big data-style applications. In: *IPDPS*, pp. 819–828 (2014)
29. Mutlu, O., et al.: Research problems and opportunities in memory systems. *SUPERFRI* **1**(3), 19–55 (2014)

30. Quinlan, D.: ROSE: compiler support for object-oriented frameworks. *Parallel Process. Lett.* **10**(02n03), 215–226 (2000)
31. Song, Y., et al.: Design and implementation of a compiler framework for helper threading on multi-core processors. In: *PACT*, pp. 99–109 (2005)
32. Stanley-Marbell, P., et al.: Pinned to the walls: impact of packaging and application properties on the memory and power walls. In: *ISLPED*, pp. 51–56 (2011)
33. Tang, X., et al.: Improving bank-level parallelism for irregular applications. In: *MICRO*, pp. 1–12 (2016)
34. Vetter, J.S., et al.: Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Comput. Sci. Eng.* **17**(2), 73–82 (2015)
35. Vijayaraghavan, T., et al.: Design and analysis of an APU for exascale computing. In: *HPCA*, pp. 85–96 (2017)
36. Wu, K., et al.: UNIMEM: runtime data management on non-volatile memory-based heterogeneous main memory. In: *SC*, pp. 58:1–58:14 (2017)
37. Wu, K., et al.: Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In: *SC* (2018)
38. Yoon, H., et al.: Row buffer locality aware caching policies for hybrid memories. In: *ICCD*, pp. 337–344 (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.