# A Pattern Aware Optimization for Hybrid Main Memories (Unrefereed Workshop Manuscript*)

Eishi Arima[1,a)] Martin Schulz[2]

***Abstract:*** The ever increasing demand of High Performance Computing (HPC) applications for higher memory bandwidth and, at the same time, larger memory capacity is leading the industry towards hybrid main memory designs, i.e., main memories that consist of multiple different memory technologies. This trend, however, leads to one important question: how can we efficiently utilize such hybrid memories? We propose a novel approach to solve this challenge by deploying a software-based pattern-aware staging technique for memory intensive HPC applications. More specifically, our approach samples small parts of the address sequence, characterizes the pattern, and then determines whether to apply the staging or not at runtime to maximize performance. Our experimental results using HPC kernels on a KNL processor achieve a 3x speed-up in the best case compared to an execution using only the large memory.

***Keywords:*** Hybrid Main Memories, Pattern Analysis, Access Staging

## 1. Introduction

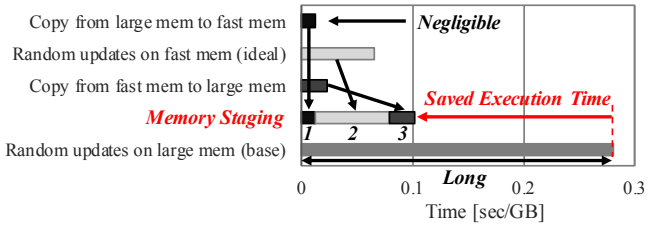
*Guter Anfang ist halbe Arbeit.* — ***Anonym***

The performance of future High Performance Computing (HPC) systems relies less and less on the number of Flop/s provided, but rather directly depend on both memory bandwidth and capacity [1]. As a consequence, systems built solely on classic memory technologies, which normally only favor one of the two properties, will face severe limitations. In order to counteract this trend, new and promising technologies, such as 3D stacking, HMC [2] or HBM [3], have been developed, but face limitations in term of capacity and scalability [4]. Therefore, to increase the memory capacity, DIMM-based off-package memories including NVRAM, such as Intel's 3D XPoint memory [5], are still needed, but also face limitations, this time in terms of bandwidth-scalability due to power constraints on the memory-bus [6] and the number of off-package pins [7]. Driven by these diverging observations, hybrid memory architectures, which combine different memory technologies on a single processor, are an important design option for upcoming HPC generations, including exascale systems [8]. For example, the Intel's Knights Landing (KNL) processor offers such a hybrid approach with two different memories: the high-bandwidth, but small MCDRAM, which is a form of HBM, and the conventional low-bandwidth, but large DIMM-based DDR4 memory [9].

While such hybrid memory systems have the potential to improve the performance of bandwidth-critical applications, it is still unclear how to exploit—at the same time—both the available bandwidth 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 observation illustrated in Figure 1: (1) the fast memory outperforms the large memory for random updates task, but (2) it takes a much longer time than the sequential copy tasks. This is because, thanks to significant amounts of parallelisms due to a large number of channels, the fast memory can successfully accelerate any kinds of bandwidth critical tasks. However, the effective bandwidth for each task highly depends on the access pattern, as more irregular and sparse (low locality) tasks utilize fewer elements within a cache-line, which significantly wastes memory bandwidth [10] and also causes more bank/row-buffer conflicts [11].

We exploit this phenomenon to accelerate bandwidth critical 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 bandwidth-heterogeneous hybrid memory systems [12], [13], [14], [15], [16], [17], [18], but none of them considers *this large performance impact of the access pattern nor exploits it to improve performance.*

[1] The University of Tokyo, Japan
[2] Technical University of Munich, Germany
[a)] arima@cc.u-tokyo.ac.jp
* SIG Technical Reports are nonrefereed and hence may later appear in any journals, conferences, symposia, etc.




* This experiment was performed on the KNL-based system described in Section 6. The same number/size of memory references are issued for both the random and the copy task (details are provided in Section 2.3).

Fig. 1: Concept of our staging technique

To successfully enable our pattern-aware staging technique, we need to decide when it is profitable to apply. For this, we propose a lightweight software-based mechanism that 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, to characterize access patterns based on sampled addresses, we propose an efficient approach consisting of two different detectors implemented with ***Bloom filters***: a Page Address Filter (PAF) for sparseness analysis and a Stride Filter (SF) for regularity analysis. Finally, we propose a quantitative methodology to use the analysis to decide if an application can likely benefit from staging or not.

The followings are the major contributions of this paper:

- We report the following important observations for our staging technique: (1) the high-bandwidth fast memory outperforms the large memory for bandwidth-critical tasks in a hybrid memory system, and (2) the overhead of a sequential copy operation between them can be relatively small when the memory access pattern of the executed task is irregular and sparse.
- Based on the observation, we propose a software-based approach called ***pattern-aware staging*** that samples parts of an access sequence with multiple threads, characterizes the access pattern, and determines whether to apply the staging or not.
- We propose a simple address sampling mechanism, which can easily be integrated into the compilers/runtime system tool chain.
- We realize a lightweight pattern characterization mechanism using two different small Bloom filters: PAF and SF. Our evaluation suggests that sampling and analyzing only 2K/1K addresses with only 256B Bloom filter per thread is enough for PAF/SF, which takes less than 0.040% of time compared with an 8GB round-trip copy operation between the memories.
- We propose a quantitative approach to make a decision based on the outputs of the above access pattern analysis. Our experimental result shows that our approach provides and accuracy of 79.0%.
- Finally, we implement and evaluate our pattern-aware staging approach on a KNL-based system using HPC kernels. The result clarifies that our proposal offers a 3x times speed-up in the best case compared to the conventional large memory only approach.

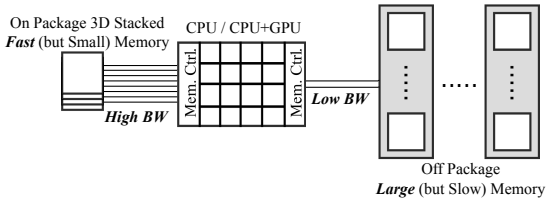



Fig. 2: Hardware architecture with bandwidth-heterogeneous hybrid main memory

| | Bandwidth | Capacity |
|---|---|---|
| **Fast Memory**(MCDRAM) | ***up to 450GB/s*** | up to 16GB |
| **Large Memory**(DDR4) | up to 90GB/s | ***up to 384GB*** |

Table 1. An example of bandwidth-heterogeneous hybrid main memory (supported in KNL processors [9])

## 2. Staging Accesses in Hybrid Memory

*Look deep into nature, and then you will understand everything better. — **A. Einstein***

To support applications with both high bandwidth and large capacity memory requirements, HPC systems have begun to support hybrid main memories with heterogeneous bandwidth properties. Looking forward, this kind of architecture is not only considered indispensable for any next generation HPC systems, covering exascale and beyond [8], but is also poised to find its way into mainstream systems. Figure 2 shows a typical hardware architecture for such a system: the memory consists of two different parts: one consists of fast (in terms of bandwidth), but small on-package memory; while the other one consists of large, but slow off-package memory.

Table 1 shows a sample configuration of a hybrid memory system, as it is implemented in KNL-based systems. As evident from the numbers, these different memory technologies have a trade-off between bandwidth and capacity (their latency is comparable [9]). MCDRAM is a 3D-stacked fast memory technology offering high on-package internal bandwidth with TSV (Through Silicon Via) and silicon interposer technologies [19]. Although this kind of 3D-stacking technology is promising in terms of the bandwidth, the capacity is limited compared to conventional DIMM-based memories [4]. Therefore, the KNL also supports conventional DIMM-based DDR4 memory. Although this technology is better in terms of the capacity it can offer, it faces limitations wrt. power budgets for the memory bus [6] and off-package pin counts [7], leading to less scalable bandwidth properties. Consequently, only both memories combined offer a viable path forward, leading us to hybrid memory systems.

### 2.1 Concept of Memory Staging

The goal of this research is to provide an easy to use way for ***memory-consuming bandwidth-critical applications*** to exploit the high-bandwidth of the fast mem-

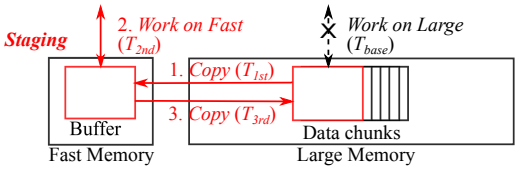



Fig. 3: Concept of staging

ory, while also being able to utilize the capacity of the large memory. To achieve this goal, we aim aim at utilizing corse-grained data transfers/copies (data chunks in the order of GBs) as accessing large enough data is essential to exploit the available bandwidth. 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 few GB) in the fast memory and divide the large data, still stored in the slow memory, into the several data chunks of matching 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 on the fast memory, and (3) return it to the large memory by copying it back. We then iterate this process across all data chunks, until all chunks are processed.

### 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 ($T_{copy}$). They can be formulated by using the parameters shown in Figure 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 conditions:

$$T_{base} > T_{1st} + T_{2nd} + T_{3rd}$$
$$T_{boost}(= T_{base} - T_{2nd}) > T_{copy}(= T_{1st} + T_{3rd}) \quad (1)$$

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 wether 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 Observation

In Figure 4, we quantify the performance boost ($T_{boost}$) by comparing $T_{2nd}$ and $T_{base}$. For this evaluation, we utilized a KNL-node for which the details are shown in Section 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 8GB data array whose data element size is eight bytes; for *sequential* we examined sequential memory references on the same 8GB data array, also by issuing one billion memory references.

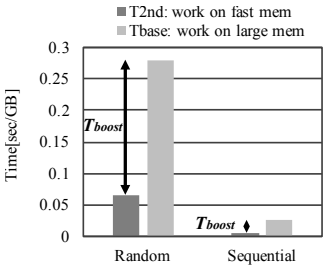



Fig. 4: Performance boost

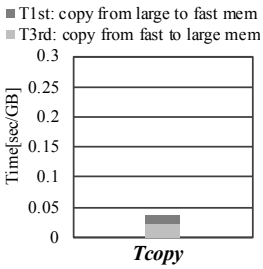



Fig. 5: Copy overhead

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 a much higher bandwidth regardless of access patterns if the accesses are intensive.

The random access pattern, on the other hand, takes much longer to complete than the sequential one, and hence $T_{boost}$ has to become much longer for the former. This phenomenon is caused by the fact that memory systems are usually optimized in a way that they can exploit the bandwidth for sequential accesses by interleaving data across banks/ranks/channels [20], while utilizing open page policies [21]. Therefore, more *irregular* patterns cause more bank/rank/channel level conflicts [11]. 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 Figure 4 and Figure 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 ($T_{boost} < T_{copy}$).

### 2.4 Overlapping and Pipelining

Pipelining is a well-known technique to hide the communication latency between components/nodes by overlapping computation and data transfer [22], [23], [24], [25], [26]. In our case, the second stage for one chunk and the first/third copy stages of other chunks can be overlapped. By overlapping them, the time of processing one chunk becomes *ideally* $T_{2nd}$.

However, when we apply overlapping to *bandwidth-critical tasks*, the effectiveness is limited due to *signifi-*

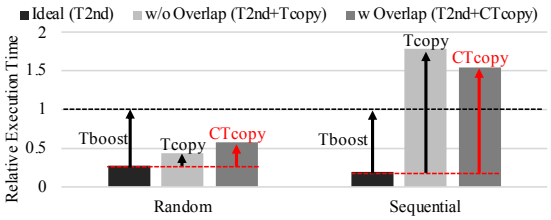



Fig. 6: Overhead comparison

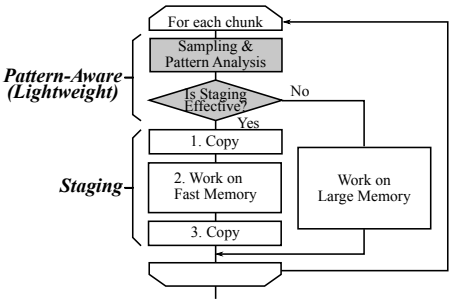



Fig. 7: Block diagram for Pattern-Aware Staging

*cant hardware contention* (referred to as $C$) on the memory bandwidth resources, particularly on the fast memory, as all of the stages access it intensively. We refer to this contention overhead as $T_{oh} = C * T_{copy}$, and *compared to pure staging*, the overlapping is effective only when $C$ is less than 1.

Figure 6 compares execution time among three methods on the KNL-based system: `Ideal` (no copy overhead), `w/o Overlap` (staging without overlapping), and `w Overlap` (staging with overlapping). The X-axis indicates the workloads (`Random` and `Sequential`), which are the same as those used in the previous subsection, while the Y-axis represents relative execution time which is normalized to $T_{base}$ for each workload. In this evaluation, `Ideal` or `w/o Overlap` are executed by 64 threads, while for `w Overlap`, additional 64 copy threads also run in parallel[*1], which are implemented using the OpenMP nested parallelism.

As shown in the figure, the performance benefit of overlapping and pipelining is limited or even harmful for bandwidth-critical applications, which are the major targets of this work. This is because (1) the overlapping fundamentally does not reduce the loads on the memory subsystem, (2) it can cause more conflicts on the memory resources (e.g., at row buffers [27]) for `Random`, and (3) the copy time is too large to hide for `Sequential`. *Due to this limited effectiveness, we purposely do not consider them in our approach.*

## 3. Pattern-Aware Staging

*Die Natur liebt Einfachheit und Einheit.* — ***J. Kepler***

Following the insights in the last section, particularly Section 2.3, which shows that staging can be highly beneficial in some cases, but harmful in others when always

[*1] The copy threads are launched across all cores to balance the loads among them. In this approach, two different threads share the same core resources (e.g., local caches), but the impact is small because all the workloads are memory bound with very little cache locality — thus, the memory limits performance in this case.

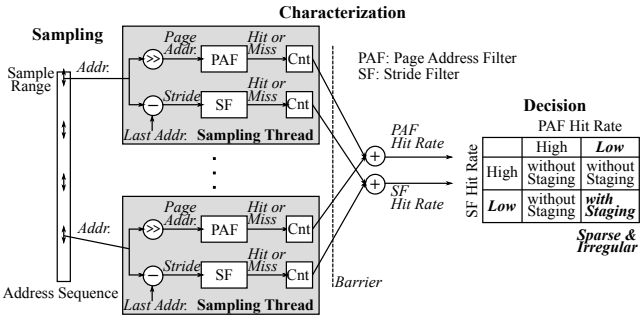



Fig. 8: Overall strategy of pattern analysis

applied, 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 7 shows the overview with a block diagram: we sample the access sequence for a chunk just before executing the task, analyze the pattern, and then use this information to make a decision on whether we use the staging or not, i.e., we make a pattern-aware decision. The time and memory overhead of this part has to be small enough (compared to the copy overheads of few GB of big chunk copy) 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 flexible to adapt to varying application behavior than performing a static, offline based pattern-analysis. Consequently, *no profile from a previous run is needed for the application of our method.*

Figure 8 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. 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 (but the first 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

```
1  for(i=0;i<num_chunks;i++){
2    //processing ith chunk
3    #pragma our_directive target(A[][])
4    for(j=0;j<M;j++){
5      for(k=0;k<N;k++){
6        A[i][(j*I[k])%L]+=1;
7      }
8    }
9  }
```

Fig. 9: Original code + newly introduced directive

```
1  for(j=0;j<M;j++){
2    for(k=0;k<N;k++){
3      PAF.input(&A[i][(j*I[k])%L]);
4      SF.input(&A[i][(j*I[k])%L]);
5      sample++;
6      if(sample >= max_sample) {
7        goto SAMPLE_END;}
8    }
9  }
10 SAMPLE_END:
```

Fig. 10: Sampling threads

```
1  for(i=0;i<num_chunks;i++){
2    //processing ith chunk
3    [Put the sampling threads code here]
4    if(decision_making(args)){
5      [code with staging]
6    }else{
7      [code without staging]
8    }
9  }
```

Fig. 11: Pattern-aware staging code

the copy and thus the performance boost brought by data staging will be larger and hence worthwhile.

## 4. Sampling and Characterization

*Zeit ist Geld. — **Anonym***

In this section, we explain the details of our sampling and characterization approach in Figure 8, and then quantify the overhead. More specifically, we describe how we realize the sampling in Section 4.1, the details of the filter mechanism in Section 4.2, and the overhead analysis in Section 4.3.

### 4.1 Sampling threads

Figure 9 represents a sample code to apply our pattern-aware staging technique[*2]. In this figure, the 2D array (A[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 Figure 10/11 for the target array. Although this transformation is performed by hand in this paper, as in previous software-based data management studies [28], [29], [30], this can be automated using, e.g., a source-to-source compiler, similar to previous studies on compiler-based pre-execution or helper thread prefetching [31], [32], [33]—they generate both load instructions and addresses, while ours needs the latter only.

Next, we describe the code for the sampling threads in Figure 10[*3]. This code can be considered a modified version of the inner two loops of Figure 9. More specifically, instead of calculating A[i][(j*I[k])%L]+=1, our sampling threads just obtain the address (&A[i][(j*I[k])%L]) and replace with the calculation with the code for their own filters (PAF and SF) in the innermost for loop. Note that, if the array is accessed multiple times in the loop (e.g., unrolled loop), we add the filter input and sample counter increment lines to each. 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 placed at the 3rd line in Figure 11, just before the decision making function (decision_making(args)).

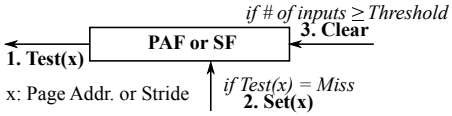



Fig. 12: Filter operations

### 4.2 Access Characterization

We characterize the access pattern in terms of *sparseness* and *regularity* using the sampled address sequence. To do so, as described in Section 3, we convert the address sequence into page/stride addresses for the PAF/SF filters, respectively, which keep their own address history, and then acquire the hit rates. To realize this, the filters have to be *efficient in terms of memory and time overheads*. For this reason we turn to Bloom filters, as thy fulfill these requirements, as laid out below.

#### 4.2.1 The Filter Mechanism

We assume each filter has three functionalities: $Test()$, $Set()$, and $Clear$ as shown in Figure 12. First, the $Clear$ function is used to initialize the contents to initialized and reset the filter, as described later. 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.

#### 4.2.2 Bloom Filter Based Implementation

To implement PAF and SF, we utilize Bloom filters, which are a probabilistic data structure that can record a large set of elements with a small memory footprint [34]. Figure 13 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-

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

[*3] This code is also parallelized with OpenMP in our implementation. Note that we apply the following modifications to parallelize the code: (1) as OpenMP does not allow GOTO statement in *parallel for* loops, we utilize *cancel for* statement as a substitution for GOTO; (2) as we set the statistics of the filters as private variables to minimize the communications among threads, we collect them using *atomic* statement just after the end of the samplings.

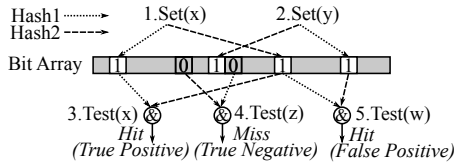



Fig. 13: Bloom filter mechanism

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 in the filter 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 return 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, 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. Nevertheless, due to their probabilistic nature and in combination with well chosen hash functions, Bloom Filters (using the right array sizes and hash functions) have been shown to be highly effective in recording such information in a compact fashion [34].

### 4.3 Overhead Analysis

We evaluate the overhead of our sampling and characterization approach using access patterns for various sparse matrices. The matrices are collected from the Florida sparse matrix collection [35] and are listed in Table 2. Assuming SpMV with CRS format [36], we use the column indices of each matrix as an index array to a vector and

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 |

Table 3. Sampling and filter settings

| Sampling | |
|---|---|
| # of sampling / thread | 2K ([1K, 2K, 4K, 8K] in Fig. 14/15) |
| # of threads | 64 |
| **Filters (PAF/SF)** | |
| Size [B] ($=2^N/8$) | 256 ([64, 128, 256, 512] in Fig. 16) |
| Max # of inputs | 256 |
| # of hashs | 2 |
| $hash_k(x)$ $(k=0,1)$ | $(x >> (N*k))\&(2^N-1)$ |

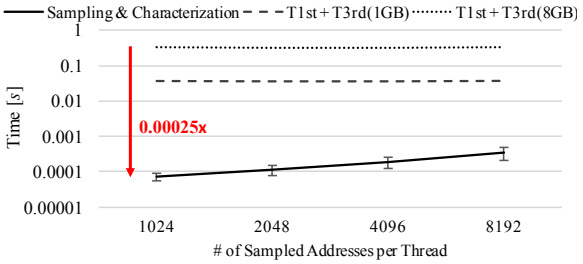



Fig. 14: Time overhead comparison

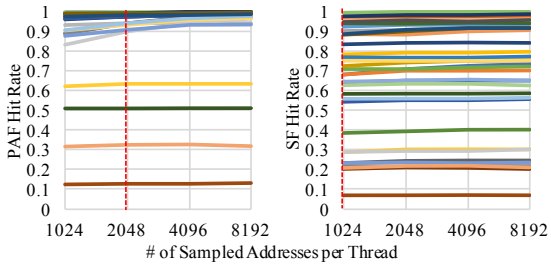



Fig. 15: PAF/SF hit rates v.s. the number of sampling

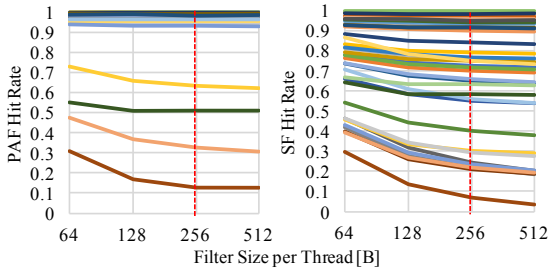



Fig. 16: PAF/SF hit rates v.s. filter size

analyze the access patterns with using our sampling and characterization approach. For this evaluation, we use a KNL-based system whose detailed configuration is shown in Section 6. The configurations for our sampling phase and the filters are summarized in Table 3.

Figure 14 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.

As shown in the figure, when we limit the number of sampled addresses to less than 8K 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 8GB copy at 1K samples.

Figure 15 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 2K/1K addresses per thread. Based on this result, we limit ourselves to 2K/1K addresses per thread for the PAF/SF. The time overhead of this is less than ***0.040%*** compared to the 8GB

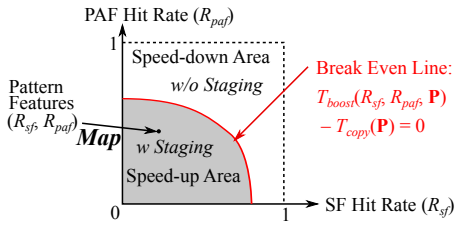



Fig. 17: Overview of decision making strategy

copy operations, as shown in Figure 14.

Figure 16 presents the PAF/SF hit rates as a function of the filter size. We scale the filter size from 64B to 512B (512bit to 4096bit) 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 256B. Based on this result, we choose 256B for both PAF and SF.

## 5. To Stage or Not To Stage?

*Wer die Wahl hat, hat die Qual.* — ***Anonym***

Based on the pattern features we determine whether it is likely beneficial to apply our staging technique or not. In this section, we first show the overview of the method and then discuss the criteria to make decisions. Finally, we analyze the accuracy of our approach.

### 5.1 Overview

Figure 17 illustrates the overview of our strategy: on the $R_{sf}$-$R_{paf}$ plane, we consider the break even line (BEL) — at any points on the line, the performance improvement gained in the second stage ($T_{boost}$) is equal to the copy overhead ($T_{copy}$). If the pattern feature vector $(R_{sf}, R_{paf})$ is mapped below the BEL on the plane, we can gain speed-up with the staging, otherwise not. The BEL is formulated as follows:

$$T_{boost}(R_{sf}, R_{paf}, \mathbf{P}) - T_{copy}(\mathbf{P}) = 0 \quad (2)$$

In addition to the pattern features, this function also utilizes additional input parameters (denoted through the vector $\mathbf{P}$), which help fine tune the shape of the BEL. In particular we choose three additional parameters $(U, R, W)$ — how many times chunks will be accessed and whether they will be read or written, also see Table 4 — on which $T_{boost}()$ and $T_{copy}()$ highly depend on for bandwidth critical applications. $T_{boost}()$ (the performance gain) will be shorter if the chunk is less utilized ($U$ is smaller) and the access direction is relevant due to read/write bandwidth differences. Furthermore, $T_{1st}/T_{3rd}$ in $T_{copy}()$ can be skipped if the chunk is write/read-only as described in Section 2.2. In this work, we set the parameters $(\mathbf{P})$ manually, but we assume this can be given by a compiler-based code analysis.

### 5.2 Decision Criterium

First, we formulate $T_{copy}()$ [s/GB] with $R$ and $W$:

| Time Functions/Parameter | |
|---|---|
| $T_{boost}()$ | Speed-up gained in the second stage [s/GB] |
| $T_{copy}()$ | Time overhead of the copy operations [s/GB] |
| $T_{th}$ | Threshold to set the aggressiveness of decisions [s/GB] |
| **Pattern Features** | |
| $R_{paf}$ | Page Address Filter (PAF) hit rate [0:1] |
| $R_{sf}$ | Stride Filter (SF) hit rate [0:1] |
| **Given parameters ($\mathbf{P}$)** | |
| $U$ | # of accesses to the chunk / # of elements in the chunk |
| $R$ | 1 (if the chunk is read), 0 (otherwise) |
| $W$ | 1 (if the chunk is written), 0 (otherwise) |

Table 4. Functions and parameters

$$T_{copy}(\mathbf{P}) = R/B_{1st} + W/B_{3rd} = R \cdot T_{1st} + W \cdot T_{3rd} \quad (3)$$

In the equation (3), $B_{1st}/B_{3rd}$ or $T_{1st}/T_{3rd}$ represent the copy bandwidth or the time per GB of the first/third stages (see also Section 2.2). Note that $T_{copy}()$ does not depend on $R_{sf}$, $R_{paf}$, or $U$ as it has nothing to do with how the chunk is accessed during the task except for $R$ and $W$.

Second, the following shows the definition of $T_{boost}()$ [s/GB] (time per chunk size):

$$T_{boost}(R_{sf}, R_{paf}, \mathbf{P}) = T_{boost}(R_{sf}, R_{paf}, U, R, W)$$
$$= U \cdot T_{boost}(R_{sf}, R_{paf}, 1, R, W) \quad (4)$$
$$= \begin{cases} U \cdot T^{read}_{boost}(R_{sf}, R_{paf}) & (R = 1, W = 0) \\ U \cdot T^{write}_{boost}(R_{sf}, R_{paf}) & (R = 0, W = 1) \\ U \cdot T^{rw}_{boost}(R_{sf}, R_{paf}) & (R = 1, W = 1) \end{cases} \quad (5)$$

Equation (4) assumes $T_{boost}$ to be proportional to $U$ for the same access features $(R_{sf}, R_{paf})$, i.e., it assumes that a task takes N times longer when the access sequence also becomes N times longer with the same pattern, which is generally the case. We then (Equation 5) split $T_{boost}$ into three functions depending on the access type $(R, W)$ as read/write bandwidths are different in several memory systems.

In order to describe $T^*_{boost}(R_{sf}, R_{paf})$, we utilize the following linear approximation $(* = read, write, rw)$:

$$T^*_{boost}(R_{sf}, R_{paf}) = a^*_0 R_{sf} + a^*_1 R_{paf} + a^*_2 \quad (6)$$

We choose this simple function as it is easy to determine the coefficients $(a^*_i)$ by just testing the following access patterns on each memory (fast/large): (1) random accesses on a large enough array $(R_{sf} \simeq 0, R_{paf} \simeq 0)$, (2) accesses with a long enough stride $(R_{sf} \simeq 1, R_{paf} \simeq 0)$, and (3) sequential streaming accesses $(R_{sf} \simeq 1, R_{paf} \simeq 1)$. By getting $T^*_{boost}()$ for these patterns, we can solve the following linear equations:

$$\begin{cases} a^*_2 = T^*_{boost}(0,0) \ (= T^*_{brand}) \\ a^*_0 + a^*_2 = T^*_{boost}(1,0) \ (= T^*_{bstrd}) \\ a^*_0 + a^*_1 + a^*_2 = T^*_{boost}(1,1) \ (= T^*_{bseq}) \end{cases} \quad (7)$$

By using $T^*_{brand}$, $T^*_{bstrd}$, and $T^*_{bseq}$ of the above equations, the equation (6) is deformed as follows:

$$T^*_{boost}(R_{sf}, R_{paf}) = T^*_{brand} - (T^*_{brand} - T^*_{bstrd})R_{sf} - (T^*_{bstrd} - T^*_{bseq})R_{paf} \quad (8)$$

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:

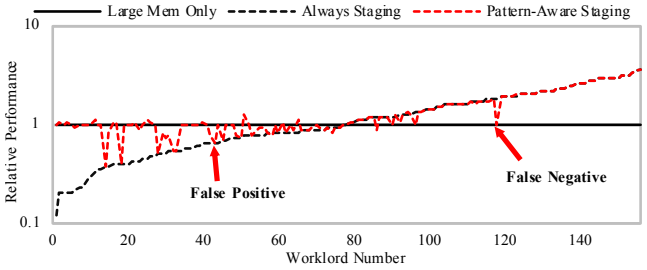



Fig. 18: Performance impact of false decisions

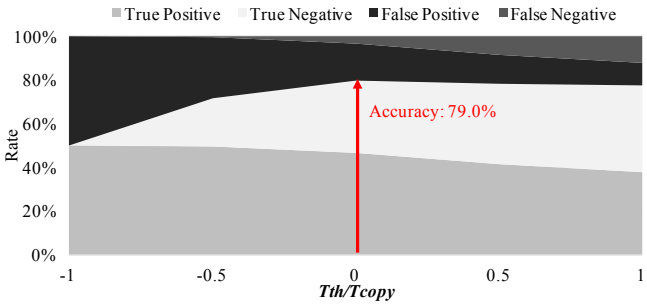



Fig. 19: Correctness of decisions

$$T_{boost}(R_{sf}, R_{paf}, \mathbf{P}) - T_{copy}(\mathbf{P}) > T_{th} \quad (9)$$

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

### 5.3 Accuracy analysis

In this section, we evaluate the accuracy of our staging criteria using synthetic workloads. The system configurations will be described in Section 6, and the sampling thread settings are based on the evaluation in Section 4.3. We apply our staging technique to the source vectors in SpMV operations (CRS format) whose matrices are listed in Table 2 of Section 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 90GB. Also, we scale the number of rows of the matrices ranging from 1 to 1/32 to change the chunk utilization ($U$).

Figure 18 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 $U$ but higher $R_{sf}$ and $R_{paf}$ — chunks are less utilized and more regularly accessed with higher locality. In this graph, the threshold parameter $T_{th}$ 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 Section 2.2); and thus (2) this comparison becomes more error tolerant when the performance difference of the two approaches becomes larger.

Figure 19 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 parameter $T_{th}$. 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.

## 6. Evaluation setup

*An experiment is a question which science poses to Nature* — ***M. Planck***

We evaluate our proposal using various bandwidth-critical HPC kernels. In this section, we clarify the environment. First, we explain the system configuration, the details of the calibration to obtain the coefficients, and the compared methods in the evaluation (Section 6.1). Then, we describe our implementation on benchmark applications (Section 6.2).

### 6.1 Methodology

#### 6.1.1 Configuration

Table 5 summarizes the environment for our experiments. We utilize a KNL-based system whose nodes provide a hybrid memory system. In particular, we use *Flat mode* for the memory system except for the hardware cache evaluation and *Quadrant mode* for the in-node cluster mode setting (one cluster per node). Note that our method is effective and extensible to any other in-node cluster settings such as *SNC mode* (dividing the 64-cores within a node into multiple clusters) [9]. The operating system used for the evaluation is Cent OS 7 [37] and we use Intel C/C++ compiler (ICC) [38] v19.0.1.144 with using the following options: -O3, -qopenmp, -lmemkind, and -xMIC-AVX512. The sampling thread settings are based on the evaluation in Section 4.3, and the threshold parameter $T_{th}$ is set to 0. Through this evaluation we set the number of threads to 256 for all of the applications.

Table 5. Evaluation setting

| Name | Remarks |
|---|---|
| CPU | XeonPhi 7210, 64cores, 1.3GHz, quadrant mode |
| Memory | MCDRAM: 16GB 450GB/s, DDR4: 96GB 90GB/s |
| OS | CentOS 7 |
| Compiler | ICC 19.0.1.144 |

In our implementation, the buffer is allocated to the fast memory using the memkind library, which is designed to use different kinds of memories in a computing node [39].

#### 6.1.2 Calibration

To conduct the pattern analysis for applications, we have to correctly set the coefficients described in Section 5.2: $T_{1st}$, $T_{3rd}$, $T^*_{brand}$, $T^*_{bstrd}$, and $T^*_{bseq}$ ($* = read, write, rw$). Here, we summarize how to acquire them. First, to obtain $T^*_{brand}$, $T^*_{bstrd}$, and $T^*_{bseq}$, we measure the bandwidth of (1) 1G times random accesses on an 8GB array, (2) 2M times stride accesses on 8GB array (4K+B stride), and (3) a streaming task on 8GB array. These measurements are performed for read-only, write-only, and read-and-write cases on both fast and large memories. Second, to determine $T_{1st}$ and $T_{3rd}$, we just measure the copy bandwidth between the memories.

#### 6.1.3 Performance Comparison

We compare the performance of the following methods:

***Large-only:*** The execution only with the large DDR4 memory.

***NumactlP:*** The execution with a *Numactl* command (numactl –preferred) which preferentially stores data on the fast MCDRAM memory [9], [40]. If the node runs out of the fast memory, then it allocates data on the large DDR4 memory.

***Hardware$:*** The KNL-based systems support hardware cache mode, with which the fast memory works as an usual direct map cache [9].

***Proposal:*** The execution with the proposed pattern-aware staging.

### 6.2 Implementation

Our proposal is implemented manually in each application, following the example of various published studies of software-based data management [28], [29], [30]. In this evaluation, we choose bandwidth critical kernels from HPC Challenge (HPCC) [41], NAS Parallel Benchmarks (NPB) [42], and also use stencil codes (Jacobi2D/3D) [43]. 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 [44]. 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 64GB and 16GB, 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 96GB (=48GBx2) and 16GB, respectively

***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 8GB (the array size for one time step), and the total data size is 80GB.

***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 2GB (= the chunk size). In this evaluation, the total data size is 90GB, which includes multiple different vectors.

***FFT (HPCC)*:** This workload calculates one dimensional FFT using two 32GB arrays: input and output array. We apply our staging technique to the output array by dividing it into 4GBx8 chunks. Through the evaluation, a temporal array is located at the fast memory.

## 7. Experimental result

*A measurement is the recording of Nature's answer.*
*— **M. Planck***

Figure 20 compares the performance among the methods across all applications. The vertical axis indicates relative performance that is normalized to the *Large-only* approach for each application. Our method achieves a factor of three performance improvement over *Large-only* at the best case, and on average, it improves performance by a factor of 2.0. As the data management policy of *NumactlP* is naive, it does not improve performance for most cases. Compared to *Hardware$*, our approach has the following benefits in terms of performance: (1) ours purposely puts the useful chunk of data on the fast memory based on the pattern-analysis thus can avoid unnecessary conflicts on it; and (2) ours can fully utilize the hardware resources of the fast memory, but the hardware cache wastes the available bandwidth/storage due to the hardware overheads such as tags. Thanks to these characteristics, our pattern-aware staging outperforms *Hardware$* for almost all workloads in this evaluation.

One exception in Figure 20 is `ConjugateGradient` (the hardware cache works better than ours), and we can see the reason in Figure 21: in the figure, the X-axis represents the total data footprint size, while the Y-axis indicates the relative performance which is normalized to *Large-only* at 16GB. 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, ours would work better than the hardware cache for this workload.

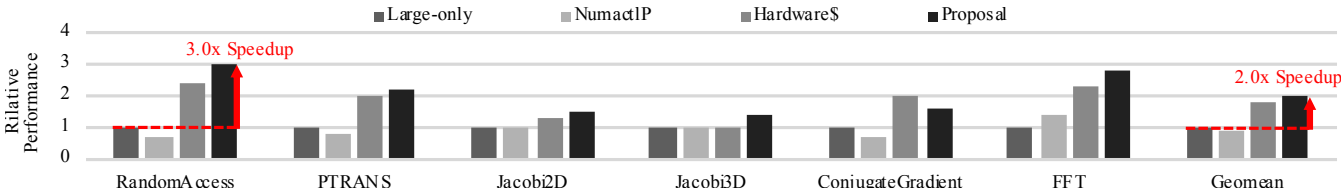



Fig. 20: Performance comparison among methods across applications

| Name | $R_{paf}$ | $R_{sf}$ | $U$ | $R$ | $W$ | Estimated $T_{boost}/T_{copy}-1$ | Measured $T_{boost}/T_{copy}-1$ | Decision Correctness |
|---|---|---|---|---|---|---|---|---|
| RandomAccess | 0.0388 | 0.0620 | 4 | 1 | 1 | 22.9 (> 0) | 20.3 (> 0) | **Correct** |
| PTRANS | 0.00297 | 0.999 | 1 | 1 | 0 | 6.60 (> 0) | 3.73 (> 0) | **Correct** |
| Jacobi2D | 0.998 | 0.998 | 5 | 1 | 0 | 3.51 (> 0) | 1.67 (> 0) | **Correct** |
| Jacobi3D | 0.996 | 0.998 | 7 | 1 | 0 | 5.37 (> 0) | 2.18 (> 0) | **Correct** |
| ConjugateGradient | 0.368 | 0.0947 | 7 | 1 | 0 | 31.8 (> 0) | 17.8 (> 0) | **Correct** |
| FFT | 0.968 | 0.998 | 4.5 | 0 | 1 | 5.54 (> 0) | 9.08 (> 0) | **Correct** |

Table 6. Statistics of our pattern-aware staging approach

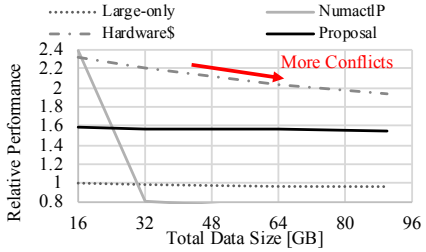



Fig. 21: Performance vs. data size (ConjugateGradient)

Finally, we summarize the statistics of our approach in Table 6. The pattern features ($R_{paf}$, $R_{sf}$) are taken from our sampling technique, while the other parameters ($U$, $R$, $W$) are set by manually counting the number of read-/write references to the target array in the target loop. As for the decision makings, if $T_{boost}/T_{copy}$ is greater than 1, we should use the staging technique; otherwise not. From this point of view, *as long as the signs of the estimated/measured* $T_{boost}/T_{copy}-1$ *are the same, our approach is correct,* and our approach is correct for all the workloads. In most cases, the estimated values are higher (our decisions are aggressive), which is adjustable by setting $T_{th}$ higher like we did in Section 5.3.

## 8. Discussion

*Nihil est sine ratione.* — ***G. W. Leibniz***

***Automation:*** Although we quantify the effectiveness of our proposal, some parts, such as the sampling and the staging, are hand coded, and a few parameters used in our criteria are given by the programmer. In future work, we will automate them in the compilers/runtime tool chain. Other promising options for this automation are hardware and/or operating system side approaches based on our idea. By using the automation, we may find yet another optimization opportunity, such as combining conventional prefetches and our staging technique to exploit their synergistic effects.

***Latency critical cases:*** In this paper, we purposely did not consider latency critical cases in the formulation/evaluation due to the following reason: HPC applications generally have plenty of parallelisms at various levels (e.g., thread, instruction, and data level) thanks to less dependencies among instructions (or data), thus they are rather throughput/bandwidth critical regardless of the access patterns — otherwise they cannot be parallelized well. However, to consider latency or bandwidth criticality in future work, we demonstrate the relationship between the parallelisms/dependencies and the criticality on our KNL-based system using a synthetic code shown in Figure 22. In this evaluation, we scale the number of references to the index array (I[]) needed to update an element of the data array (A[]) — we define this number as *Depth*. The higher we set *Depth*, the more the code has instruction/data dependencies like pointer chasing workloads.

```
1  # pragma omp parallel for simd
2  for(i=0; i<N; i++){
3    A[I[I[I[I[i]]]]] += 2;
4  }
```

Fig. 22: Tested synthetic code ($Depth = 4$)

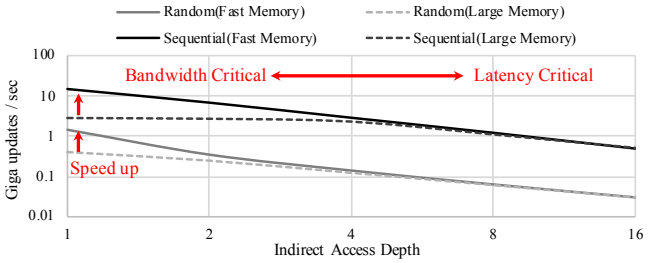



Fig. 23: Bandwidth/latency criticality v.s. dependency

Figure 23 illustrates the result. Through the evaluation, we set both the number of elements (for A[] and I[]) and the number of iterations (N) as 1G. We put random/sequential numbers ranging from 0 to 1G-1 on the index array (I[]) for `Random`/`Sequential` workloads. The code can be considered latency limited when the speed-up brought by the fast memory is small (i.e., $Depth \geq 4$) because the latencies of the 3D stacking and the DIMM technologies are comparable [9], [45]. The evaluation result suggests that *regardless of the access patterns*, the instruction (or data) dependencies determine whether the code is bandwidth or latency critical. Therefore, applying dependency/parallelism analysis (e.g., data/control-flow analysis [46], [47], [48]) on a code region of interest at compilation time and adjusting $T_{boost}$ in accordance with the dependency/parallelism will be a promising way to cover it.

## 9. Related work

*Wissen ist Macht.* — ***Anonym***

***Data management on hybrid main memories:*** Various data management techniques have been proposed for hybrid main memories. We classify them into bandwidth-aware [12], [13], [14], [15], [16], [17], [18] or latency-aware [14], [49], [50], [51], [52].

Several recent publications tackle bandwidth-aware data managements for hybrid memory systems. However, our approach is unique in terms of *analyzing access patterns at runtime* and for being a pure *software-based approach*. Recent studies provided software-based techniques [12], [13], [14], [18], but they did not exploit *the large performance impact of access patterns* to optimize coarse grained data managements. Others proposed hardware-based approaches for hybrid memories: prefetch buffers in the logic of 3D-stacked fast memory [15], access partitioning technique for multi-program workloads [16], and a data compression hardware for 3D-stacked fast memory (used as cache mode) [17]. Although these approaches are promising, they require hardware modifications.

Latency-aware techniques have been proposed for DRAM + NVRAM based hybrid memory systems focusing on the latency difference in the memories. Although these techniques are useful for *latency-critical applications*, they are less effective for *bandwidth-critical applications*. Generally, HPC applications are more bandwidth-critical due to their higher data/instruction/thread-level parallelism, and thus processor cores issue more memory requests in parallel while executing them.

***Data management between on/off-chip memories:*** Prior studies have proposed various hardware/software techniques for data management between on-chip memories (caches or scratchpads) and off-chip main memories. Some of them focus on hardware/software-based prefetching techniques that attempt to store data on upper levels of the memory hierarchy before the requests [28], [29], [30], [31], [32], [33], [53], [54], [55]. Others have proposed software-based explicit data managements for on-chip scratchpad memories [56], [57], [58], [59]. The major differences from our study are: (1) they do not perform the pattern analysis at runtime for coarse grained data managements as their caches/scratchpads are *too small* for this approach, and (2) they generally attempt to reduce latency in exchange for the available memory bandwidth.

***Data management on NUMA-based systems:*** NUMA (Non-Uniform Memory Architecture) based systems also have multiple memories, and several data management techniques have been proposed for them [60], [61], [62]. These techniques partition a node into sets of local cores and memories, and attempt to allocate a pair of threads and data on the same part to counteract the fact that local memory is faster than the remote memory. However, such allocation techniques are not useful for hybrid memory system management in a node (or in a local part of NUMA + hybrid memory) since in this case *the smaller memory is faster regardless of thread location*.

***Data management on CPU-GPU hybrid systems:*** CPU-GPU hybrid computing, where discrete GPUs are attached via PCIe, also utilizes multiple memories: CPU host memory and GPU device memory. The major difference from our target is that current GPUs cannot directly access CPU host memory [63]. Thus, it is necessary to move data from the host to the device memory for any data requests *regardless of the access patten* [24], [25], [26]. In contrast, this is not required in our targets, opening additional opportunities that can be leveraged with our novel dynamically applied pattern-aware technique. Further, recent GPUs support unified memory spaces for CPU-GPU systems to automate the data transfer [63], [64]. Although the memories are unified in virtual address spaces, GPUs still *physically* utilize only their own device memories. If host memories are physically shared, we can extend our work to GPUs.

## 10. Conclusions

*Ende gut, alles gut.* — ***Anonym***

This paper proposed and made a case for a software-based data management technique called patten-aware staging to exploit—at the same time—both the high bandwidth and the large capacity components of hybrid main memory systems. Our technique dynamically examines the irregularity of memory accesses and, in case of irregular patterns, fetches chunks of data from large memories to fast memories, just before they are referenced. More specifically, we sample parts of the address sequence with multiple threads, analyze the pattern with two different Bloom filters, and then make a decision by quantifying the performance benefit based on outputs of the filters. The experimental result using HPC kernels on a KNL node show that our Bloom filter based detector enables a 3x speed-up in the best case compared to a large memory only approach.

## Acknowledgment

We would like to thank the members of CAPS at TU Munich and the staffs of SRD, ITC, U Tokyo for valuable discussions. We also appreciate Dr. Edgar A. Leon at LLNL for his insightful suggestions in the early stage of this work. Part of this work is under the auspices of the U.S. Department of Energy by LLNL under Contract DE-AC52-07NA27344, while Eishi Arima was a visiting scientist at LLNL until Oct. 2017 and before Martin Schulz moved from LLNL to TUM in Oct. 2017. 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] S. Matsuoka *et al.*, "From FLOPS to BYTES: Disruptive Change in High-performance Computing Towards the Post-

moore Era," in *CF*, 2016, pp. 274–281.
[2] H. M. C. Consortium, "Hybrid Memory Cube Specification 2.1," *Last Revision Jan*, 2015.
[3] J. Standard, "High Bandwidth Memory (HBM) DRAM," *JESD235*, 2013.
[4] J. Kim *et al.*, "HBM: Memory Solution for Bandwidth-Hungry Processors," in *Hot Chips 26 Symposium (HCS)*, 2014, pp. 1–24.
[5] K. Bourzac, "Has Intel Created a Universal Memory Technology? [News]," *IEEE Spectrum*, vol. 54, no. 5, pp. 9–10, 2017.
[6] N. Chatterjee *et al.*, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017, pp. 73–84.
[7] P. Stanley-Marbell *et al.*, "Pinned to the Walls: Impact of Packaging and Application Properties on the Memory and Power Walls," in *ISLPED*, 2011, pp. 51–56.
[8] T. Vijayaraghavan *et al.*, "Design and Analysis of an APU for Exascale Computing," in *HPCA*, 2017, pp. 85–96.
[9] J. Jeffers *et al.*, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.
[10] N. Tanabe *et al.*, "A Memory Accelerator with Gather Functions for Bandwidth-bound Irregular Applications," in *IA3*, 2011, pp. 35–42.
[11] X. Tang *et al.*, "Improving Bank-Level Parallelism for Irregular Applications," in *MICRO*, 2016, pp. 1–12.
[12] A. Benoit *et al.*, "A Performance Model to Execute Workflows on High-Bandwidth-Memory Architectures," in *ICPP*, 2018, pp. 36:1–36:10.
[13] N. Butcher *et al.*, "Optimizing for KNL Usage Modes When Data Doesn't Fit in MCDRAM," in *ICPP*, 2018, pp. 37:1–37:10.
[14] K. Wu *et al.*, "Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory," in *SC*, 2017, pp. 58:1–58:14.
[15] M. Islam *et al.*, "HBM-Resident Prefetching for Heterogeneous Memory System," in *ARCS*, 2017, pp. 124–136.
[16] J. Gaur *et al.*, "Near-Optimal Access Partitioning for Memory Hierarchies with Multiple Heterogeneous Bandwidth Sources," in *HPCA*, 2017, pp. 13–24.
[17] V. Young *et al.*, "DICE: Compressing DRAM Caches for Bandwidth and Capacity," in *ISCA*, 2017, pp. 627–638.
[18] R. J. Huber *et al.*, "Evaluating Best and Worst Case Scenarios on Two-Level Memory Systems," *SC, poster session*, 2016.
[19] G. H. Loh *et al.*, "Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems," in *MEMSYS*, 2015, pp. 3–10.
[20] G. J. Burnett *et al.*, "A Study of Interleaved Memory Systems," in *AFIPS '70 (Spring)*, 1970, pp. 467–474.
[21] D. Kaseridis *et al.*, "Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era," in *MICRO*, 2011, pp. 24–35.
[22] M. J. Clement *et al.*, "Overlapping Computations, Communications and I/O in Parallel Sorting," *Journal of Parallel and Distributed Computing*, vol. 28, no. 2, pp. 162–172, 1995.
[23] J. C. Sancho *et al.*, "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications," in *SC*, 2006, pp. 17–17.
[24] B. Van Werkhoven *et al.*, "Performance Models for CPU-GPU Data Transfers," in *CCGRID*, 2014, pp. 11–20.
[25] R. Mokhtari *et al.*, "BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications," in *IPDPS*, 2014, pp. 819–828.
[26] J. Hestness *et al.*, "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors," in *IISWC*, 2015, pp. 87–97.
[27] O. Mutlu *et al.*, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007, pp. 146–160.
[28] J. D. Collins *et al.*, "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *ISCA*, 2001, pp. 14–25.
[29] C. Zilles *et al.*, "Execution-based Prediction Using Speculative Slices," in *ISCA*, 2001, pp. 2–13.
[30] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *ISCA*, 2001, pp. 40–51.
[31] M. Kamruzzaman *et al.*, "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads," in *ASPLOS*, 2011, pp. 393–404.
[32] Y. Song *et al.*, "Design and Implementation of a Compiler Framework for Helper Threading on Multi-core Processors," in *PACT*, 2005, pp. 99–109.
[33] D. Kim *et al.*, "Design and Evaluation of Compiler Algorithms for Pre-execution," in *ASPLOS*, 2002, pp. 159–170.
[34] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[35] T. A. Davis *et al.*, "The University of Florida Sparse Matrix Collection," *ACM TOMS*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
[36] N. Bell *et al.*, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *SC*, 2009, pp. 18:1–18:11.
[37] M. Alibi *et al.*, *Mastering CentOS 7 Linux Server*. Packt Publishing, 2016.
[38] K. Dickinson, "Intel®C++ Compiler 17.0 Developer Guide and Reference," *INTEL*, 2016.
[39] C. Cantalupo *et al.*, "User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," 2015.
[40] "numactl," https://github.com/numactl/numactl.
[41] P. Luszczek *et al.*, "Introduction to the HPC Challenge Benchmark Suite," *Lawrence Berkeley National Laboratory*, 2005.
[42] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
[43] N. Maruyama *et al.*, "Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *SC*, 2011, pp. 1–12.
[44] B. He *et al.*, "Efficient Gather and Scatter Operations on Graphics Processors," in *SC*, 2007, pp. 1–12.
[45] D. W. Chang *et al.*, "Reevaluating the Latency Claims of 3D Stacked Memories," in *ASP-DAC*, 2013, pp. 657–662.
[46] B. K. Rosen, "High-level Data Flow Analysis," *Communications of the ACM*, vol. 20, no. 10, pp. 712–724, 1977.
[47] D. W. Wall, "Limits of Instruction-level Parallelism," in *ASPLOS*, 1991, pp. 176–188.
[48] M. S. Lam *et al.*, "Limits of Control Flow on Parallelism," in *ISCA*, 1992, pp. 46–57.
[49] G. Dhiman *et al.*, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *DAC*, 2009, pp. 664–669.
[50] L. E. Ramos *et al.*, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011, pp. 85–95.
[51] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012, pp. 337–344.
[52] Z. Wu *et al.*, "Efficient Memory Management for NVM-Based Hybrid Memory Systems," *International Journal of Control and Automation*, vol. 9, no. 1, pp. 445–458, 2016.
[53] K. J. Nesbit *et al.*, "Data Cache Prefetching Using a Global History Buffer," in *HPCA*, 2004, pp. 96–.
[54] D. Callahan *et al.*, "Software Prefetching," in *ASPLOS*, 1991, pp. 40–52.
[55] T. Chen *et al.*, "Prefetching Irregular References for Software Cache on Cell," in *CGO*, 2008, pp. 155–164.
[56] R. Banakar *et al.*, "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," in *CODES*, 2002, pp. 73–78.
[57] M. Kondo *et al.*, "SCIMA: Software Controlled Integrated Memory Architecture for High Performance Computing," in *ICCD*, 2000, pp. 105–111.
[58] L. Li *et al.*, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management," in *PACT*, 2005, pp. 329–338.
[59] A. Größlinger, "Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes." *CC*, vol. 9, pp. 236–250, 2009.
[60] W. Bolosky *et al.*, "Simple but Effective Techniques for NUMA Memory Management," in *SOSP*, 1989, pp. 19–31.
[61] H. Löf *et al.*, "Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System," in *ICS*, 2005, pp. 387–392.
[62] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, 2013.
[63] T. Zheng *et al.*, "Towards High Performance Paged Memory for GPUs," in *HPCA*, 2016, pp. 345–357.
[64] NVIDIA, "NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU," *GP100 Pascal Whitepaper (WP-08019-001_v01.1 ed.)*, 2016.