Dynamic Prefetcher Reconfiguration for Diverse Memory Architectures

Junghoon Lee
Samsung Advanced Institute of Technology
jh1512.lee@samsung.com

Taehoon Kim
School of Computing, KAIST
thkim@calab.kaist.ac.kr

Jaehyuk Huh
School of Computing, KAIST
jhuh@calab.kaist.ac.kr

***Abstract*—With the advent of stacked memory and new memory architectures, the heterogeneity of memory has been increasing. In the diverse memory technologies, each memory architecture has its own advantages and weaknesses. Considering the trade-offs, future systems are expected to support multiple memory architectures with a hybrid memory system. However, such diversity of memory architectures complicates the performance optimization of on-chip memory hierarchy. One of the key components affected by this trend is the hardware prefetcher. The available memory bandwidth highly affects the effectiveness of prefetchers, and the aggressiveness of prefetchers must be tuned for memory architectures as well as application behaviors. This paper investigates the effect of memory diversity on the prefetcher parameter selection, and proposes a dynamic parameter search mechanism to adjust the prefetch aggressiveness under various memory architectures. Using a general hill climbing scheme periodically, the mechanism adapts to the memory architectures and application behaviors effectively. In addition to such automatic tuning, the study improves the solution for cache pollution exacerbated by the increase of speculative data from more aggressive prefetchers in higher bandwidth memory. With the dynamic parameter search and pollution mitigation, the proposed framework improves the performance of applications by 12.4% on average compared to the prior scheme for tuning prefetch parameters.**

I. Introduction

For the past several decades, the memory technology has been advancing fast with ever higher density and more bandwidth. However, for each generation of DRAM, one type of DRAM architecture has been dominantly adopted in commercial systems, although each system may have some differences in the number of available channels and their organization. Due to the relatively homogeneous DRAM configurations, on-chip memory hierarchy in each generation has been optimized for one type of memory architecture. However, with the advent of 3D-stacked DRAM architectures and new non-volatile memory, the heterogeneity of memory architectures will increase tremendously. With the 3D-stacked DRAM architectures, the range of configurable memory channels will be much higher than the current DDR3 where most of the commercial processors use 2-4 channels. New non-volatile memory which may partly replace DRAM as the secondary main memory, will also increase the available memory choices. Furthermore, many future systems are expected to have a hybrid memory system combining two or more from stacked DRAM, external DRAM, and non-volatile memory.

However, the heterogeneity of memory architectures complicates the prefetcher design. Prefetchers are known to be sensitive to memory architectures as well as application behaviors [1]. The available memory bandwidth and latency highly affect how aggressively prefetchers must generate prefetch requests. A prior study advocated that the aggressiveness of prefetchers must be dynamically adjusted considering the application behaviors for a given fixed memory architecture [1]. In the study, the feedback-directed prefetcher chooses one of 5 different levels of aggressiveness. However, such pre-selected candidate configurations become no longer the best ones, if the underlying memory architectures change or multiple different architectures are used for the processor.

This study investigates the effect of the diversity of memory architectures on the prefetcher parameter selection, showing that any fixed prefetch parameters cannot always be the best ones under the increasing memory diversity. Based on the observation, this study advocates that a dynamic parameter selection mechanism must be used to accommodate such diverse memory architectures. To prove that such a dynamic parameter selection is feasible, this paper presents a live prefetcher reconfiguration mechanism, which searches the best aggressiveness configuration online. Instead of relying on a few pre-selected configurations, the new framework uses a general hill climbing algorithm based on random walks to find the best configuration dynamically. The prefetch controller periodically tries different aggressiveness configurations to adjust the parameters on the fly.

Another potential problem of prefetching is the cache pollution problem, since prefetched data are stored in the on-chip caches shared by prefetch and demand data. For higher bandwidth memory architectures, prefetchers generate requests more aggressively than traditional DDR3 architectures. Due to the increased speculation, the worsened cache pollution can offset the benefit of more aggressive prefetchers. However, contrary to the concern on the problem, this study shows that the cache pollution causes a relatively minor effect, since the cache size has been increasing, and the prefetchers are tuned to become aggressive only when necessary. However, to further improve the prefetcher performance, we modified the prior solution for our parameter selection mechanism, achieving a modest but meaningful performance improvement [1].

The experimental results show that the proposed dynamic parameter adjustment with random sampling can find good

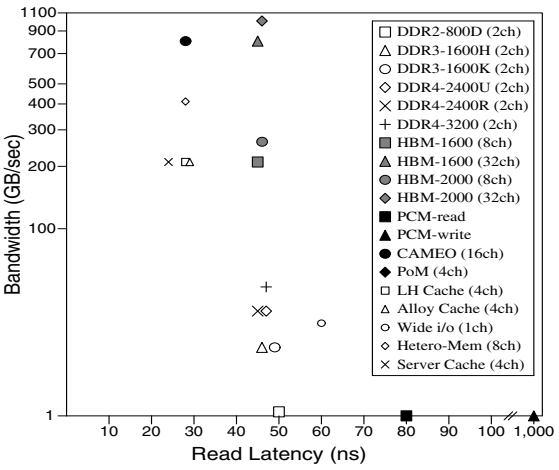


Fig. 1: Latency and bandwidth for different memory types : DDR, HBM [2], PCM [3], CAMEO [4], PoM [5], LH Cache [6], Alloy Cache [7], Hetero-Mem [8], and Server Cache [9]

prefetcher parameters for various applications under seven different memory organizations. Even under such diverse memory architectures, the dynamic prefetcher reconfiguration can adjust the prefetcher parameters effectively. With a new high bandwidth memory architecture, the proposed scheme can improve the performance of applications by 12.4% on average compared to the prior approach.

II. Background

A. New Generation Memory

There have been significant advancements in memory architectures. 3D stacked DRAM technologies have matured for commercial systems, increasing the memory bandwidth by an order of magnitude. New non-volatile memory is expected to partly replace DRAM for its better density and non-volatility. With distinct performance and capacity trade-offs in different memory architectures, systems will most likely adopt a hybrid memory system consisting of multiple memory architectures.

3D-stacked memory assembles DRAM chips vertically using Through Silicon Vias(TSVs) [10], [11]. There have been several different projections about the bandwidth and latency characteristics of such stacked memory. The `true-3D` module is projected to improve memory access latencies by 32.5% over the traditional memory [10]. Subsequent studies also expected fast DRAM parameters with stacked memory, which has a roughly two times faster latency than the current DDR3 memory [4], [5].

High Bandwidth Memory (HBM) is an industrial standard for stacked DRAM, which connects multiple memory stacks and processors with a silicon interposer [2]. Each stack of DRAM has 8 channels, and a package can have up-to 4 stacks on an interposer with the total 32 memory channels supporting 1TB/s aggregate memory bandwidth. However, one of the critical difference between HBM and the projected stacked DRAM in academic research is its access latency. HBM uses the same DRAM organization as DDR4 and thus it does not directly improve the DRAM access parameters significantly. Therefore, HBM improves the memory bandwidth by an order of magnitude, but has little improvement in unloaded memory access latency.

Even with a similar 3D-stacked memory technology, many academic projections and industrial standards diverge for latency and bandwidth characteristics. Figure 1 presents the diversity of latency and bandwidth of various memory architectures. For some memory types, the supported bandwidth is not disclosed, and thus they are shown on the x-axis. Many of the plotted memory architectures may co-exist for the same generation of processor architectures, and they will be combined to provide hybrid memory systems.

Hybrid Memory: Along with the new DRAM and NVRAM technologies, the main memory system is expected to consist of multiple different architectures. Such hybrid memory architecture further complicates the prefetcher optimization. A data request from an LLC miss can be served by different memory architectures with distinct latency and bandwidth characteristics. The hybrid memory can be managed by the pure hardware mechanisms advocated by recent studies [5], [4]. In the approaches, a hardware translation table maps the logical memory space to the system memory space consisting of multiple memory types. However, a more evolutionary way to support the hybrid memory is to expose the heterogeneity of memory to the operating system, and the OS manages the memory mapping [12].

B. Prefetching

Hardware prefetching can convert the high bandwidth of stacked memory to actual performance of applications. In this paper, we use one of the most widely used prefetchers, *stream prefetcher*, as our target prefetcher.

1) Stream Prefetcher: Stream prefetcher have been used extensively both in commercial [13], [14] and research fields [1], [15], [16]. Stream prefetchers prefetch a contiguous chunk of memory if locality is found for address streams. Although its potential coverage for prefetching can be high as a locality-based prefetching, a large portion of prefetched data may not be used, if the accuracy is low. The prefetcher can track multiple streams simultaneously with stream entries, and an initial miss to a new region inserts a new stream entry. Once the entry is created, any subsequent access to the nearby locations (within +/- N cacheblocks) from the stream start point will train the entry. Once the entry is trained, the prefetcher monitors the memory regions from the beginning point (address A) to the end point (address A + P), and if any demand access to the region occurs, the prefetcher generates prefetch requests from address A+P+1, A+P+2, ... , A+P+N.

The prefetcher has two important parameters governing its aggressiveness behavior. First, the *prefetch distance* (P in the aforementioned description) determines how far future the prefetcher predicts, and second, the *prefetch degree* (N) determines how many prefetch requests are generated for each trigger. In this paper, the prefetch distance ranges from 4 to 64, and prefetch degree ranges from 1 to 32.

2) Feedback-directed Prefetcher: Feedback-directed prefetching proposed by Srinath et al. adjusts the two parameters (distance and degree) depending on how well the prefetcher works for the current application [1]. In the study with a single-core setup, instead of selecting two parameters independently, it defines five different aggressive levels, *very conservative (4,1)*, *conservative (8,1)*, *middle-of-the road (16,2)*, *aggressive(32,4)*, and *very aggressive (64,4)*. In the parenthesis, the first number denotes the prefetch distance, and the second number denotes the prefetch degree.

The feedback-directed framework monitors the accuracy (high, middle, low), prefetch lateness (high, low), and cache pollution (yes or no) with several estimation mechanisms. Based on the three monitored states of prefetcher performances, one of the five aggressiveness parameters is selected. In addition to the aggressiveness adjustment, the framework also uses a dual insertion policy to determine the position of prefetch data in the LRU chain. If the prefetched data are determined to be useful, they are inserted to the MRU position. Otherwise, they are inserted to the LRU position not to pollute the demand data.

The feedback-directed prefetching adjusts its aggressiveness parameters based on how well a running application exploits the prefetcher capability. However, when the aggressiveness levels are defined, it is based on a given fixed memory architecture and how many cores can share it. For each memory architecture, the five levels of pre-selected parameters may not cover the optimal configuration for the memory type. As will be discussed in the next section, different memory architectures require different aggressiveness controls. With many possible memory configurations, it can be difficult to cover all the available memory architectures, when the processor and prefetcher are designed. In the next section, we present how difference in memory architecture affects the optimal prefetcher aggressiveness.

III. Motivation

A. Methodology

To evaluate this work, we use the McSim [17] simulator based on the Pin binary instrumentation. The simulator uses the RUBY model in the GEMS simulator [18] for the detailed memory system model. The core model is a 4-way out-of-order processor. The private L2 cache capacity is 256KB per core, and the shared L3 cache has 2MB capacity for each core, with an 16MB LLC for the 8-core configuration. For the external memory, we use DRAMSim2 with different timing parameters for each memory type. The rest of system configurations are shown in Table I. For workloads, we use 16 applications from the SPEC CPU2006 benchmark suite. We use both single-application runs where the same application runs on every core in a multi-core configuration, and mixed runs with different co-running applications. The details of mixed run combinations are shown in in Table II.

Memory Architecture: In this paper, we evaluate seven memory configurations. Table III lists the seven configurations with their parameters. `DDR` uses DDR3-1600 parameters, and two memory channels are used for 8 cores. `HBM` uses HBM-1600 parameters with 16 channels. In this memory configuration derived from the industry specification, it uses a similar timing as DDR4. In this paper, we use the HBM memory as our primary new memory architectures. However, in addition to the baseline HBM from the standard, we also explore additional variants of HBM with lower latency or with lower bandwidth. `fast HBM` is a hypothetical stacked DRAM memory, which mimics the parameters used in recent stacked DRAM studies in academia [4], [5]. Although the channel bandwidth is the same as `HBM`, the internal DRAM frequency is doubled, so that it can reduce the unloaded latency by half compared to `HBM`. `half HBM` is similar to `HBM`, but uses 8 channels instead of 16, reducing the overall bandwidth by half. HBM can be configured in various ways, and `half HBM` represents one possible configuration. The rest three hybrid memory architectures combine the DDR memory and one of the three HBM memory architectures.

Parameter	Value
Processor	8 out-of-order cores (3.2GHz, 4 ways, 128 ROB size, x86 ISA support)
L1 Caches	32KB per-core, 4-way set associative 64B block size, 1-cycle latency
L2 Caches	256KB per-core, 8-way set associative 64B block size, 9-cycles latency
L3 Caches	2MB per-core, 16-way set associative 64B block size, 21-cycles latency
Coherence	Directory-based coherence, MOESI protocol
Memory Controller	Open-page policy FR-FCFS scheduling [19]
Prefetcher	Stream prefetcher with 8 streams [13]

TABLE I: System configurations

Prefetcher Configuration: The evaluated stream prefetcher has 8 stream entries, and there is a stream prefetcher for each core, since each core may run different workloads. Each stream is trained by LLC misses, but triggered by LLC accesses. We use 5 distance parameters (4, 8, 16, 32, and 64), and 6 degree parameters (1, 2, 4, 8, 16, and 32). The framework must find a pair of distance and degree parameters from 30 possible combinations.

B. The Effect of HBM on Prefetcher Designs

One of the most significant benefits of stacked memory is the increase of memory bandwidth. Although the prior studies of stacked memory show that the application performance can be improved both from the reduced latencies and increased bandwidth, the commercial realization of stacked DRAM most

Name	Benchmarks in each mixed workload
mix-0	astar,povray,sphinx3,GemsFDTD
mix-1	omnetpp,bzip2,GemsFDTD,libquantum
mix-2	gcc,libquantum,omnetpp,soplex
mix-3	xalancbmk,mcf,milc,gcc
mix-4	sphinx3,mcf,hmmer,soplex
mix-5	libquantum,milc,soplex,bzip2

TABLE II: Workload mixes

Parameter	Value
DDR	4GB, 2 channels, DDR3-1600(800MHz) [20] tCL=11, tRCD=11, tRP=11, tRAS=28 8 banks/rank, 2 ranks/dimm, 1 dimm/channel
HBM	16 channels, HBM-1600(800MHz) [2] tRCD=11, tRP=12, tRAS=23 16 banks/channel 8 channels per stack
fast HBM	HBM with x2 frequency : 1600MHz
half HBM	HBM with /2 channels : 8 channels
hybrid 1	HBM + DDR
hybrid 2	fast HBM + DDR
hybrid 3	half HBM + DDR

TABLE III: Memory configurations

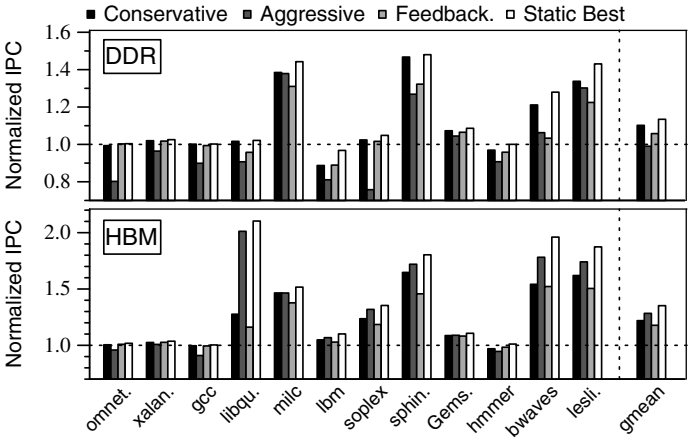


Fig. 2: Prefetcher performance with DDR and HBM

likely provides the bandwidth improvement with little changes in latencies [2].

To quantify how the increased bandwidth affects the prefetcher design, we first show the performance changes by different prefetcher parameters for DDR and HBM memory. Figure 2 presents the performance of single-type applications with DDR or HBM memory. All results are normalized to the run without prefetching for each memory type individually. For prefetch configurations, the figure shows four settings. First, a `conservative` prefetcher uses a distance of 8 and degree of 1, which generates only one prefetch at a time. Second, an `aggressive` prefetcher uses a distance of 8 and degree of 16. Third, a `feedback-directed` prefetcher uses the same 5-level aggressiveness adjusted dynamically as the prior work [1]. `static best` represents the ideal performance when the best parameters are chosen for each application by exhaustive runs with all possible combinations of the two parameters, distance and degree.

When two memory types are compared, the most critical difference is that the aggressiveness of prefetchers must be increased significantly for HBM. With the high bandwidth HBM, the demand accesses from general purpose cores with multiple caches may not fully utilize the huge bandwidth, and thus speculative accesses from prefetching help the performance significantly. In DDR, the conservative prefetcher provides a good performance compared to the aggressive prefetchers.

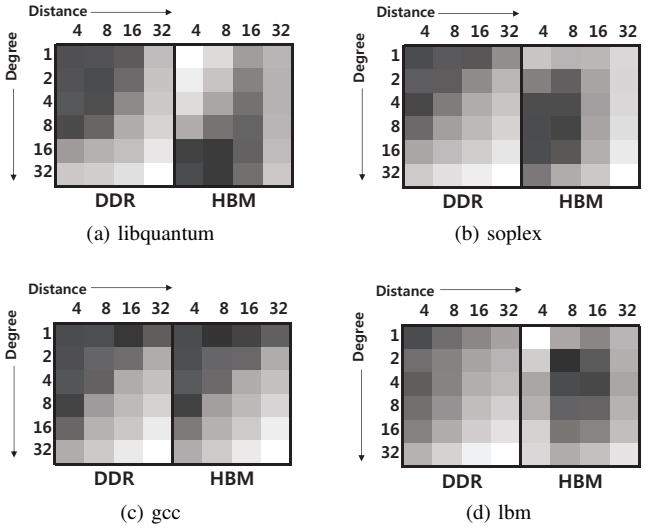


Fig. 3: Application performance with different distances (x-axis) and degrees (y-axis): darker point represents higher performance

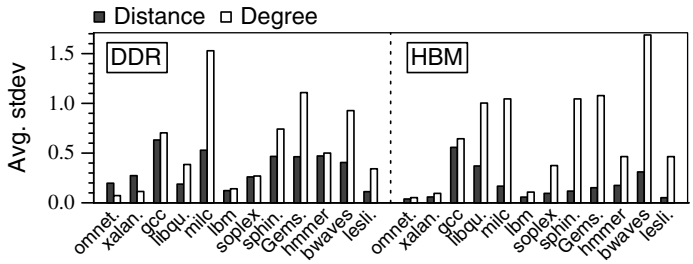


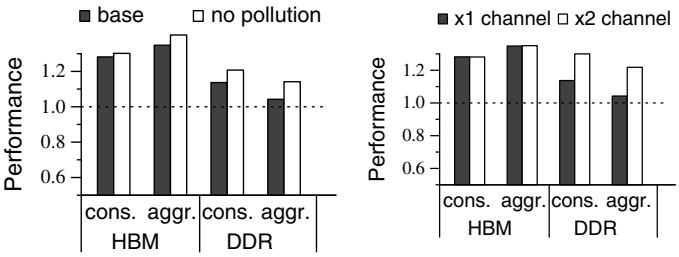
Fig. 4: Average standard deviation of performance by varying distances and degrees

However, the aggressive prefetcher performs significantly better with HBM.

With HBM, the prior feedback-directed prefetcher does not perform well, since HBM requires different parameter settings. The pre-selected aggressiveness and the fixed decision tree based on the usefulness of prefetch data, can be very sensitive to the memory architecture. Whenever the cache hierarchy and memory change, the prefetch parameters must be re-tuned. Considering the diversity of future memory architectures, it will be very difficult to consider all possible prefetcher settings.

Figure 3 presents how distance and degree parameters affect the performance of four applications. The x-axis is the prefetch distance and the y-axis shows the prefetch degree. The darker each rectangle is, the better the corresponding parameters perform. As shown in the four examples, HBM generally pushes the best parameters towards more aggressive sides. Furthermore, the best configuration differs in the four applications. A simple aggressive level may not cover the complexity of the parameter selection space.

The Effect of Distance and Degree: Between the two parameters (distance and degree), we investigate which one incurs more variance in performance. Figure 4 presents the average



(a) Pollution effect (b) Channel effect

Fig. 5: Performance impact by pollution and the number of memory channels

of standard deviation of normalized performance for each application, when only one of the two parameters changes. The gray bars represent the variance caused by *distance*. For each degree level, all possible distance parameters are evaluated, and the standard deviation of normalized performance is calculated to represent the variance by distance. Similarly, The white bars represent the variance caused by *degree*. As shown in the figure, the degree has a much higher impact on performance, since degree controls parallelism in prefetch requests which are highly affected by available bandwidth. Based on this observation, we optimize our search algorithm in Section IV.

The Effect of Pollution and Channel: Figure 5 presents how pollution and extra memory channels affect the prefetch performance in HBM and DDR. Figure 5 (a) shows the effect of mitigating cache pollution. For HBM and DDR, both `conservative` and `aggressive` settings are evaluated. The first dark bar shows the baseline cache, and the second bar shows the performance by storing the prefetched data in a hypothetical separate cache, whose size is the same as the baseline cache. Once the prefetched data are accessed by demand requests, the prefetched data are moved to the normal cache. Unused prefetch data never pollute the demand cache. As shown in the figure, the pollution still affects the effectiveness of prefetching modestly, even though the on-chip cache sizes have been increasing. When HBM and DDR are compared, the memory bandwidth has much a higher impact on the performance of prefetching. However, mitigating the pollution effect will improve the prefetching effectiveness further. Considering the relatively modest performance potential of pollution mitigation, the solution for the pollution problem is necessary, but it should not require any significant complexity and area increase in the design.

Figure 5 (b) shows how doubling memory channels affect the performance in HBM and DDR. As expected, HBM already has enough bandwidth to support 8 cores and prefetchers, and thus performance impact is small, while in DDR, the performance gain with doubling memory channels is significant.

IV. Dynamic Prefetcher Reconfiguration

To find the best prefetch parameters dynamically, we apply a search mechanism based on random sampling. Starting from a random point, the profiling execution tries several different parameters, and use the best one. This section describes the overall algorithm and how to implement it in the prefetch controllers.

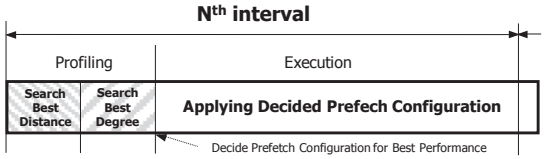


Fig. 6: Execution process

A. Search by Random Profiling

To find the best prefetch parameters dynamically with different applications and memory architectures, we employ an approach based on random sampling. The prefetcher controller for each prefetch engine periodically enters a profiling phase when the searching process is conducted. Once the best parameters are found in the profiling phase, the controller uses them for the next execution phase. Figure 6 describes the profiling and execution phases during an interval.

During the profiling phase, the controller picks a few sets of random parameters, and find the best one. Instead of using indirect metrics to evaluate the prefetch performance, such as accuracy, lateness, and pollution as used by the prior work [1], we use a direct performance metric, instruction throughput, to assess the prefetch performance. Although such random sampling will eventually find the best one with a sufficient number of trial runs, reducing the number of trials is important not to spend much of the application execution time with sub-optimal prefetch parameters. To reduce the number of trial runs, we employ several optimizations:

First, with two aggressiveness parameters (distance and degree), the search space is two-dimensional. However, to reduce the search space, the profiling phase uses two-step one-dimensional search processes. As discussed in Section III, the performance variance by distance is smaller than prefetch degree (Figure 4). The controller first tries to find the best distance with the lower performance variance, and it further searches the best degree with the higher variance.

Second, for each search of distance or degree, the searching process exploits the common forms of performance curves when distance or degree increases while the other parameter is fixed. The curves rarely exhibit multiple local maximums which differ from the global optimal point. Exploiting the common behavior, the controller starts from a random point, and tries both directions (smaller and larger parameters). Until performance starts dropping, the search process is repeated.

Third, to reduce the required steps to find the maximum performance, each profiling phase starts with the previously used best parameter. It allows the search process to finish quickly, since many workloads exhibit relatively stable behaviors across intervals. Even if the execution phase changes, the controller eventually finds the best one for the current interval from the previous best one just with increased numbers of trial runs to reach the optimal point. Furthermore, for every

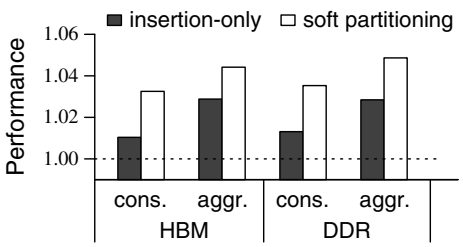


Fig. 7: Insertion-only policy vs. soft partitioning

10 intervals, the profiling run starts from a fresh random point, to avoid using obsolete parameters.

In our implementation, each trial run is 200,000 cycles. The number of trial runs vary dynamically, the average number of trial runs and its standard deviation are 3.77 and 0.42. The random profiling runs take only several steps until it reaches the best parameters. Once the profiling runs complete, the execution phase begins and lasts for four times of the profiling cycles.

B. Mitigating Cache Pollution

Although the cache pollution incurs a relatively minor performance degradation, this study investigates that how adopting a pollution mitigating solution affects the prefetcher performance. The prior study by Srinath et al. investigated the pollution problem, and their solution is to control the insertion position of prefetch data [1]. Depending on the pollution caused by of prefetch data, the prefetch data can be inserted to one of half, LRU-4, and LRU position. LRU-4 denotes the one fourth position from the LRU position in the LRU stack of each set. In the scheme, called *insertion-only*, if a demand hit occurs to a prefetched cacheline, the prefetched cacheline is promoted to the MRU position directly.

In this study, we modify the prior approach for the prefetch reconfiguration mechanism based on random walks. Furthermore, even though the prefetched data are useful, they are often not reused after the initial demand hit to the prefetched data. To avoid the pollution of caches by such patterns, we further optimize the technique by soft-partitioning for prefetch and demand data.

Instead of changing just insertion point, we propose to create a soft cache partition where the prefetch data can be stored, and the size of the partition is determined dynamically. Not to increase the complexity of cache architectures significantly, we adopted a simple pseudo-partitioning from PIPP [21], which uses both insertion point and incremental promotion to mimic the effect of partitioning. Depending on the importance of prefetched data, the prefetched data are inserted into one of the positions in the LRU chain from the MRU to LRU position. However, when hits occur, the cachelines both from demand and prefetch, are promoted just by one step toward the MRU position, instead of jumping to the MRU position in the traditional LRU policy.

To justify the effectiveness of soft partitioning for prefetch and data cachelines, we exhaustively evaluate all possible combinations of insertion points for demand and prefetch data. Figure 7 shows the average performance improvement by using the insertion-only policy and by using both insertion and incremental promotion. The figure is the average of the static best performance for all the applications, where the best insertion point is selected for each application. As shown in the figure, using the soft partitioning shows slightly higher performance than the insertion-only policy, when the best insertion point is selected.

Insertion Position for Prefetched Data: The decision process for determining the insertion position for the prefetch data must not be complicated. Furthermore, to simplify the hardware support and to be applicable to pseudo-LRU implementations, fine-grained insertion control, which puts cachelines to any position between MRU and LRU is not desirable. Therefore, from the exhaustive experiments for attempting all possible insertion positions for demand and prefetch data, we concluded that two most beneficial policies can reap most of the benefits of pollution mitigation. For the two top policies, the demand data are always inserted to the MRU position. The first policy is to put prefetch data in the quarter position to the LRU position (LRU-4: 3/4 from the MRU position), and the second policy is to put the prefetched data in the LRU position. Once both demand and prefetch data are inserted, they are promoted only by one step in the LRU stack to provide pseudo-partitioning.

The prefetcher controller selects one of the two policies, depending on the current accuracy of prefetchers. The prefetch accuracy is tracked in the same way as the prior feedback-directed prefetcher. Each cache tag has a prefetch bit, which is set for each prefetched data, and cleared when the data is accessed by a demand access. Whenever a prefetch bit is cleared, the controller increases the used prefetch counter to assess the accuracy of prefetchers.

V. Evaluation

A. Performance with Single Memory

The first results show the effectiveness of the proposed technique for four different single-type memory architectures. Figure 8 presents the performance with the proposed technique, normalized to the runs without prefetching. For each application, the first bar (`feedback`) shows the performance with the prior feedback-directed technique, and the second (`RP`) and third (`RP+PP`) bars represent the proposed random profiling (RP) without and with pollution mitigation by prefetch partition (PP).

Comparing DDR and HBM in Figure 8, the overall benefit of prefetching increases significantly with HBM, since the extra bandwidth with HBM allows more aggressive prefetching without the negative impact of wasting memory bandwidth. For example, `libquantum`, `bwaves`, and `leslie3d` show significant increases of prefetch benefit with HBM. With DDR, the feedback-directed prefetcher performs effectively and the difference between the feedback prefetcher and the proposed technique is very small, except for `sphinx3` and `leslie3d` where the proposed one has a clear advantage. However, with HBM, the performance gap between the proposed one and the

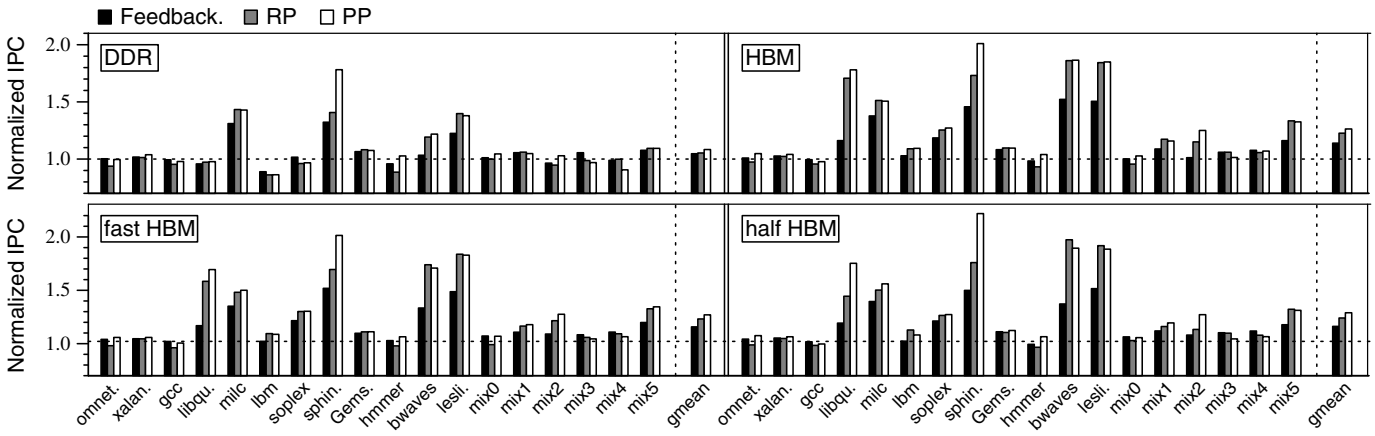


Fig. 8: Performance with four single type memory architectures: DDR, HBM, fast HBM, half HBM

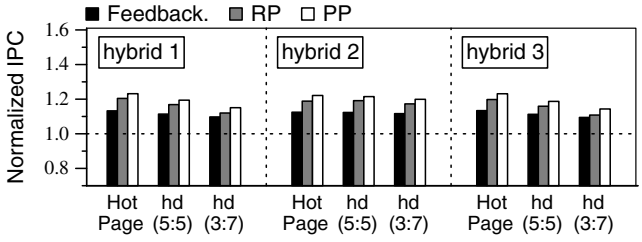


Fig. 9: Average performance with hybrid memory

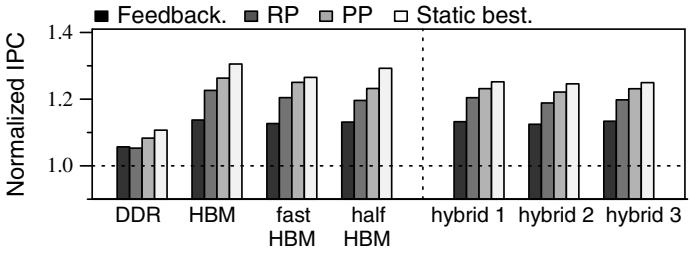


Fig. 10: Performance summary

prior work increases significantly, since the prior approach was optimized towards the DDR memory.

The two other memory architectures show similar trends as HBM as shown in Figure 8, since one of the key resources necessary for effective prefetching is memory bandwidth, and the two memory types (fast HBM and half HBM) provide ample bandwidth. In general, performance trends are similar to those with HBM, although the optimal configuration for each case may differ.

As expected, the mitigation of pollution adds extra performance improvements, although the improvement is modest by 3.6% on average. However, supporting the prefetch partitioning requires a low extra complexity with modified insertion and promotion policies. For all four types of memory, the proposed technique finds a good configuration consistently, while the feedback-directed prefetcher fails to find a good one for new high bandwidth memory architectures.

B. Performance with Hybrid Memory

Hybrid Memory Mapping: In the hybrid memory architecture, actual memory traffic patterns may differ by how the operating system allocates fast memory pages (HBM memory) and DDR memory pages to applications. In this paper, we investigate three policies. The first policy (`hotpage`) maps hot pages to the stacked memory, and the rest of pages to the DDR memory. The policy represents the memory allocation status if a dynamic page migration mechanism is extensively used for the hybrid memory. The rest two (`5:5` and `3:7`) statically partition the application memory to 5:5 and 3:7 ratios between HBM and DDR. Such page placement mimics scenarios when the operating system cannot allocate enough HBM pages to an application due to the limited HBM capacity, or the operating system decides to map more important applications to HBM by a priority policy [4].

Figure 9 shows the average performance with all hybrid memory architectures with three allocation policies. The graphs show the normalized performance with `hotpage`, `hd(5:5)`, and `hd(3:7)` placements. In the figure, the first bar is with `feedback` prefetcher. Across different hybrid memory and placement scenarios, the proposed scheme improves the feedback prefetcher by from 4.1% to 9.6%. The performance improvement is slightly higher with `hotpage`, since with the policy, the high bandwidth memory is accessed predominantly.

C. Summary

Figure 10 presents the overall performance improvement with the proposed techniques from all applications. For all cases, the proposed scheme outperforms the prior feedback-directed prefetcher by from 3.6% (DDR) to 12.4% (HBM). The overall performance improvement is close to the static per-application best configuration found by exhaustive runs with all possible parameters. Prefetch partitioning to reduce pollution further improves the performance by 3.6% on average. The proposed prefetcher reconfiguration framework achieves the performance improvement just by tuning prefetch parameters on the fly, which require little extra costs and no apparent negative impact.

VI. Related Work

There have been several studies for the importance of considering the prefetch aggressiveness. Srinath et al. designed a mechanism to dynamically adjust the aggressiveness of prefetchers with feedback information [1]. Pugsley et al. proposed a technique to evaluate the prefetching effectiveness based on sandboxing, which can determine the benefit of prefetching without initiating real prefetch requests [15]. Jimenez et al. proposed mechanisms to control and selectively use the hardware prefetcher based on a SW-based sampling technique [22].

For multi-cores, Ebrahimi et al. proposed to coordinate multiple prefetchers to prevent interference by applications running simultaneously [23]. Prefetch-aware DRAM controllers schedule demand and prefetch requests properly to utilize limited memory bandwidth efficiently [24]. Lee et al. explored the interaction between prefetchers and on-chip networks [25]. There have been several studies for managing caches considering prefetch data. Wu et al. proposed a prefetch-aware cache management to adjust insertion and promotion for prefetching [16]. Seshadri et al. reduced cache pollution by the prefetcher considering low re-usability of prefetched data [26].

VII. Conclusion

Unlike many prior prefetch studies based on a fixed memory architecture, this study investigated how the differences in memory architecture affect the optimal prefetching scheme, and how the prefetching parameters can be dynamically adjusted. The proposed search scheme based on random sampling is not tied to a fixed memory architecture, and it does not require to re-tune prefetchers or controllers for different memory systems.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (NRF-2013R1A2A2A01015514 and NRF-2016R1A2B4013352)

References

[1] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[2] JEDEC, "JESD235 High Bandwidth Memory (HBM) DRAM," 2013.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture(ISCA)*, 2009.

[4] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[5] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[6] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[7] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[8] G. H. Loh, N. Jayasena, M. O'Connor, S. Reinhardt, and J. Chung, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory System," in *the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*, 2012.

[9] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[10] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *International Symposium on Computer Architecture (ISCA)*, 2008.

[11] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.

[12] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[13] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. Res. Dev.*, 2002.

[14] Intel, "Intel Core i7 Processors,"
Online at http://www.intel.com/products/processor/corei7/.

[15] S. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[16] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "PACMan: Prefetch-aware Cache Management for High Performance Caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[17] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, 2005.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters(CAL)*, 2011.

[21] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[22] V. Jimenez, A. Buyuktosunoglu, P. Bose, F. O'Connell, F. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[23] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[24] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[25] J. Lee, H. Kim, M. Shin, J. Kim, and J. Huh, "Mutually Aware Prefetcher and On-Chip Network Designs for Multi-Cores," *Computers, IEEE Transactions on*, vol. 63, no. 9, pp. 2316–2329, Sept 2014.

[26] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM Trans. Archit. Code Optim. (TACO)*, 2015.