A Hybrid Approach to Cache Management in Heterogeneous CPU-FPGA Platforms

Liang Feng*, Sharad Sinha†, Wei Zhang* and Yun Liang‡
*Hong Kong University of Science and Technology, Hong Kong, {lfengad, wei.zhang}@ust.hk
†Nanyang Technological University, Singapore, sharad_sinha@ieee.org
‡Peking University, Beijing, China, ericlyun@pku.edu.cn

***Abstract*—Heterogenous computing is gaining increasing attention due to its promise of high performance with low power. Shared coherent cache based CPU-FPGA platforms, like Intel HARP, are a particularly promising example of such systems with enhanced efficiency and high flexibility. In this work, we propose a hybrid strategy that relies on both static analysis of applications and dynamic control of cache based on static analysis to minimize the contention on the FPGA cache in the emerging CPU-FPGA platforms with shared coherent caches. In the static analysis, we analyze memory access patterns of the accelerated kernels on FPGA using reuse distance theory and generate kernel characteristics called Key values. Thereafter, a dynamic scheme for cache bypassing and partitioning control based on these Key values is developed to increase the cache hit rate and improve the performance. We validate our proposed strategy using a system-level architectural simulator for CPU-FPGA heterogeneous computing systems. Experiments show that the proposed strategy can increase the cache hit rate by 22.90% on average and speed up the application by up to 12.52% with negligible area overhead.**

1. Introduction

Nowadays, as the traditional power and performance scaling benefits following Moore's law are diminishing, heterogeneous computing system is gaining popularity to satisfy the increasing demand of energy-efficient high-performance computing. Among various heterogeneous computing systems, the CPU-FPGA platform is considered to be one of the most promising systems thanks to FPGA's overall advantages on high performance, low power and reconfigurability to implement different acceleration functions (denoted as kernels). Microsoft has deployed customized FPGA boards called Catapult in its data center to work with CPU as accelerator and improve the page ranking throughput of the Bing search engine by 2x with only 10% more power [12], which reflects the potential of CPU-FPGA platforms.

Cache coherent CPU-FPGA platform is a new architecture proposed to improve the data communication efficiency and simplify the programming model, where CPUs and the FPGA fabric share the memory space via a cache coherent memory sub-system. In such platforms, the FPGA not only shares the L2 cache with CPUs, but also owns one L1 cache within the coherence domain, so that the FPGA can act as a coherent peer to CPUs and access the full memory space with coherence guaranteed. Several industry vendors have prototyped such type of CPU-FPGA platforms as one of their future architectural development directions. For example, Intel has shipped the HARP system integrating the Xeon E5 server processors and Stratix V FPGA fabric via the QuickPath Interconnect (QPI) bus [5], where there is a coherent L1 cache on FPGA side for quick memory access of the accelerated kernels. IBM has been developing the Coherent Accelerator Processor Interface (CAPI) on the POWER8 platform to integrate the FPGA to the CPUs [17], which also provides a coherent FPGA L1 cache. Several works have demonstrated the significant benefits in performance and energy efficiency of such systems in various application domains [14] [19], such as big data, genomics, etc. With the shared coherent cache, the data movement and coherence maintenance explicitly controlled by software can be eliminated. The data communication delay between CPU and FPGA via the shared coherent cache could be reduced compared with the traditional DMA-based data transfer, especially for the accelerated kernels with irregular memory access patterns. Therefore, the cache coherent CPU-FPGA platform is highly promising to meet today's fast-growing demand for energy-efficient high-performance computing.

However, these cache coherent CPU-FPGA systems may suffer from severe FPGA L1 cache contention when the number of the concurrent kernels running on the FPGA increases. The memory accesses issued by different kernels could have conflicts, resulting in a large quantity of cache misses on the FPGA cache. Worse still, the FPGA cache miss penalty is particularly non-trivial in most designs due to the loose coupling between the private L1 cache of FPGA and the shared L2 cache which usually resides on the CPU side. As a result, the FPGA cache contention will substantially degrade the system performance and raise the power consumption, weakening the original benefits of the CPU-FPGA platforms. Therefore, it's crucial to alleviate the FPGA L1 cache contention to maintain the performance and energy efficiency benefits for cache coherent CPU-FPGA platforms.

Different than the cache management methods proposed for CPU systems, since the accelerated kernels are pre-given in cache coherent CPU-FPGA systems, the pre-analysis of the data reuse requirement of the kernels is enabled, and hence the cache requirement of each kernel can be derived in an offline step. The execution pattern of the FPGA kernels provides the opportunity to incorporate the reuse distance based pre-analysis results into the runtime cache management, which can capture the memory access features not only more accurately, but also with less overhead due to the elimination of the runtime profiling need. In addition, the kernel based execution on FPGA makes it suitable to perform cache bypassing at kernel granularity so that the bypassing granularity can be consistent with the partitioning granularity, which enhances the feasibility to combine bypassing and partitioning for a better cache utilization. Motivated by these unique advantages of cache coherent CPU-FPGA systems on cache management, in this work

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

we propose a static and dynamic combined kernel-aware cache management scheme which integrates cache bypassing and cache partitioning techniques based on the reuse distance analysis to alleviate the FPGA L1 cache contention and optimize the cache utilization. To the best of our knowledge, this is the first work aimed at the cache management for cache coherent CPU-FPGA systems. The main contributions of this work are as follows:

- A static analysis flow using LLVM to analyze the memory access pattern of the accelerated kernels in terms of the reuse distance theory
- A dynamic cache bypassing control at accelerated kernel level with the consideration of kernel priority
- A novel cache partitioning and replacement policy that allows utilization of free cache space, partition saturation and release of non-active partitions
- A comprehensive management scheme to combine static analysis, cache bypassing, cache partitioning to enhance cache access efficiency

2. Related Work

Although there is no previous work on cache partitioning or cache bypassing customized for CPU-FPGA platforms, a lot of work has been done for cache partitioning and cache bypassing in the aspect of multi-core CPUs and GPUs.

2.1. Cache Bypassing

In [6], the cachelines with little data reuse are identified by access count based predictors and the cache access will bypass the L2 cache if there is no dead cacheline in L2 cache. A gradually decremented protecting distance is calculated for each cacheline based on runtime reuse history in [4], where the coming access will bypass the cache if the protecting distance of all the cachelines currently in the cache is larger than 0. The work in [8] tracks the reuse history of the incoming cacheline and records the optimal bypassing decision for previous references with a PC indexed counter to give the current bypassing decision. In [1], the instructions which lead to the highest miss rate and bring non-reused cachelines are marked to bypass the cache. The work in [18] bypasses the micro-cache when a load is not reused in a certain cycle or it brings a non-reused cacheline according to the compiler analysis and profiling.

While most existing solutions perform bypassing at instruction level, our scheme makes the bypassing decision at accelerated kernel level which is customized for the FPGA execution pattern. Moreover, our scheme not only guarantees the data coherence, but also takes into account the priority of the kernels.

2.2. Cache Partitioning

In [13], a runtime monitor records the miss number change for each task when varying its partition size and greedily assigns the partition size based on the monitored value. The work in [10] implements a page coloring method for cache partitioning and proposes three partitioning algorithms with different objectives to assign the partitions. The paper [3] proposes a method to dynamically choose between two partitioning strategies, where a greedy algorithm based on runtime monitored data generates multiple candidate partitioning patterns to be applied in a time sharing way. The work in [7] introduces two different partitioning methods, a static one searching all possible partitioning patterns and a dynamic one gradually adjusting the partitioning pattern based on the cache miss feedback.

Different from most previous cache partitioning methods where every task's data are needed to be allocated in cache, our scheme allows kernels not to utilize any space in cache, which can further alleviate the cache contention. In addition, our scheme is specially aimed at the partitioning for accelerated kernels and allows the cachelines occupied by each partition to vary from its assigned cachelines, which offers more flexibility to fully utilize the space.

Moreover, our proposed scheme is fundamentally new for cache management in the following aspects:

- Customized for FPGA L1 cache in cache coherent CPU-FPGA platforms, which is an untouched research area.
- Comprehensively integrate cache bypassing, cache partitioning and reuse distance analysis, while most previous work only covers one of these techniques.
- Combine static analysis and dynamic control, while most previous work only focuses on one perspective.

3. Background

3.1. Reuse Distance Theory

A reuse distance profile [2] can be calculated given an ordered memory access trace. For each access, the reuse distance is the number of unique addresses or cachelines accessed between current access and the most recent previous access to the same address or the same cacheline. The distance will be set to infinity (∞) if there is no previous access. The reuse distance analysis can be done at address granularity or at cacheline granularity. The cache hit/miss rate can be derived according to the cacheline level reuse distance profile. For a full-associative cache of n cachelines with least recently used (LRU) replacement policy, any access with a reuse distance d larger than or equal to n ($d \geq n$) will miss. Also, any access with a reuse distance d smaller than n ($d < n$) will hit. In this way, the total hit rate and miss rate can be obtained from the cacheline level reuse distance profile. For set-associative cache, the reuse distance analysis can be done by treating each set as a small full-associative cache. The analyzed memory access trace for each set is extracted from the whole memory access trace with the order kept, but only includes the accesses to the corresponding set.

Based on the reuse distance profile, the reuse distance histogram can be obtained. The total number of the accesses with a certain distance is called the frequency of that distance. The reuse distance histogram counts the frequency of each distance from 0 to ∞. The total hit and miss number can be easily calculated from a cacheline level reuse distance histogram. For a full-associative cache with n cachelines, the total hit number is the sum of the frequencies in the histogram from distance 0 to distance $n-1$, while the sum of the frequencies from distance n to distance ∞ is equal to the total miss number. For set-associative case, since each set is treated as a small full-associative cache, the total hit/miss number can be calculated by summing up the hit/miss number of all the sets.

3.2. Cache Coherent CPU-FPGA Platform

In emerging cache coherent CPU-FPGA platforms such as HARP and POWER8 CAPI, multi-core CPUs and the FPGA fabric are integrated via bus connection such as QPI. An example architecture for such system is shown in Figure 1. The FPGA fabric and multi-core CPUs share the memory subsystem including L2 cache, main memory, etc. Similar as in each CPU, an L1 cache is also provided in an FPGA enabling quick memory accesses for the accelerated kernels implemented on the FPGA. The CPU L1 caches, FPGA L1 cache, L2 cache and main memory are within the same coherence domain with the support of coherence protocol like MESI. Multiple accelerated kernels can be instantiated on the FPGA and share the FPGA L1 cache, which is a common case since different threads on the multi-core CPUs can all request the FPGA acceleration. Usually, the L2 cache resides on the CPU side, which causes an unbalanced L2 cache access delay for CPUs and the FPGA. Therefore, the overhead for handling the FPGA L1 cache miss will be relatively larger compared with the CPU L1 cache miss overhead. In addition, a bypassing path can be designed in the FPGA control logic to enable the response data for a memory request to bypass the FPGA cache and be directly sent to the kernel from L2 cache.

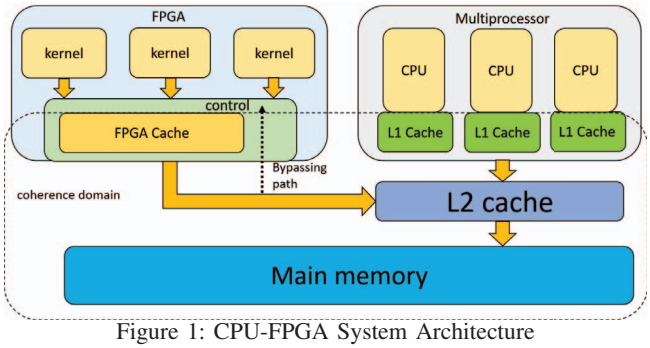

Figure 1: CPU-FPGA System Architecture

The programming model is simplified in such platforms due to the coherent cache hierarchy, which eliminates the need of explicit software control for data movement and coherence maintenance. Usually, the CPU initiates bus based transfer to the FPGA for sending the control messages to start the execution of the FPGA accelerated kernel. Each kernel will be assigned a continuous memory region from the whole virtual memory space to allocate the kernel related data, which is called workspace. Such assigned workspace is specified by setting the base address and space size in the control messages. As long as the kernel receives the start signal from the control messages, it will start execution and generate memory requests to the FPGA L1 cache. The kernel and its associated CPU thread share the same virtual memory space, which eases the programming of the accelerated kernel. The virtual to physical address translation for kernel memory request is accomplished through checking a preloaded page table on FPGA. When completing execution, the kernel sets a finish signal, which can be polled by CPU to broadcast the status of the kernel execution.

4. Proposed Methodology

Our scheme consists of a reuse distance based static analysis and a runtime control mechanism combining cache bypassing and partitioning. As the first step, an automatic LLVM analysis will be applied to each accelerated function to generate its memory access trace and corresponding reuse distance histograms. In terms of the reuse distance histograms, the LLVM analysis calculates 4 values for each accelerated function, which are denoted as Key values. Each kernel will be associated with one unique partition, thus each set of the 4 Key values describes the features of one partition. These Key values will guide the runtime dynamic control.

The dynamic control mechanism consists of cache bypassing, cacheline replacement and partition retirement, where the cache bypassing control includes the bypassing decision making, partition replacement and partition insertion. The pre-calculated Key values from the static analysis will be configured into the dynamic control to be used when performing the controls. The Key values will guide the cacheline replacement policy and the cache bypassing control. Section 5 and Section 6 describe the static analysis and the dynamic control in detail, respectively.

5. Static Analysis

During each kernel's execution, it will generate memory requests to the FPGA L1 cache with the associated addresses. With the memory access trace of each kernel, the reuse distance histograms can be obtained, which can be further used to calculate the 4 Key values. The first step of the static analysis is to obtain the memory access trace. Although the memory request address is the sum of the base address and the address offset, the generated reuse distance histograms will not be affected if we only analyze the address offset trace regardless of the base address, since the reuse distance histograms are determined only by the relative difference among the addresses. Therefore, we treat the trace of the address offset of the ordered memory requests as the memory access trace and derive the reuse distance histograms.

Since high level synthesis (HLS) is usually used to generate the HDL file for the accelerated kernel based on the C code of the accelerated kernel function, the memory offset trace can be obtained by directly analyzing the C code with an LLVM pass. In the LLVM pass, each memory request in the kernel function with its corresponding address offset will be extracted in order, which forms the memory offset trace. Alternatively, if the kernel is not generated through HLS but directly specified by a given HDL code, the Verilator [15] simulation can be applied to the HDL code to extract the memory requests in order together with the associated address offset.

After obtaining the memory offset trace of each kernel, the reuse distance histogram is generated for each set of the cache at cacheline granularity. The method to generate the reuse distance histograms follows the discussion in Section 3.1. We denote the reuse distance histogram for set i as H_i, then for a cache with 64 sets, there will be H_0 to H_{63}. In each reuse distance histogram H_i, we denote the frequency of distance j as $H_i(j)$, which means there are totally $H_i(j)$ accesses to set i with reuse distance j in that set. N_i represents the total number of all the accesses coming to set i as in equation (1), which sums up the frequencies of all different distances in histogram H_i. S_i defines a set of distances which satisfy two conditions as in equation (2) with a tunable threshold α, where $Asso$ defines the total number of ways in the cache. The first condition eliminates the distances whose frequencies are too small compared with the total access number N_i, which means there are only negligible number of accesses

with such distance. The second condition makes the set S_i only focus on the distances smaller than the cache associativity, which means all the accesses with distance in S_i can hit in cache. With these two conditions, the S_i only includes the distances whose access frequency is not negligible and whose corresponding accesses can hit in the cache.

For each kernel, based on the corresponding reuse distance histograms, 4 Key values are calculated: K, K_{min}, K_{max} and K_{coin}. Since each kernel corresponds to a unique partition, a kernel also means a partition here. For the histogram of each set H_i, we can derive $kmin_i$ and $kmax_i$ using S_i as in equation (3) and (4). $kmin_i$ is the minimal distance in set S_i ($min(S_i)$) plus 1, which indicates that if the number of cachelines assigned to the kernel in set i is smaller than $kmin_i$, there will be nearly no hit in set i for the accesses from the kernel. $kmax_i$ is the maximal distance in S_i ($max(S_i)$) plus 1, which means that assigning more than $kmax_i$ cachelines in set i to the kernel will no longer improve the hit rate. If we consider all the sets together, the K_{max} and K_{min} can be obtained as in equation (5) and (6). Here Set is the number of sets in the cache. K_{max} indicates that if the number of occupied cachelines of the kernel in a set reaches K_{max}, no more cachelines in this set should be occupied by the kernel. Otherwise, the extra occupied cachelines will not improve the hit rate but only waste the cacheline resource. As well, the intuition behind K_{min} is that the kernel should be assigned at least K_{min} cachelines in a set. Otherwise there will be nearly no hit for the accesses from this kernel due to the insufficient cacheline assignment, and the cachelines assigned to this kernel will be wasted.

$$N_i = \sum_{s=0}^{\infty} H_i(s) \tag{1}$$

$$S_i = \{s | H_i(s)/N_i > \alpha, 0 \leq s < Asso\} \tag{2}$$

$$kmin_i = min(S_i) + 1 \tag{3}$$

$$kmax_i = max(S_i) + 1 \tag{4}$$

$$K_{min} = \sum_{i=0}^{Set} kmin_i/Set \tag{5}$$

$$K_{max} = \sum_{i=0}^{Set} kmax_i/Set \tag{6}$$

C_{ij} is defined as in equation (7), which quantifies the cache benefit when the number of cachelines assigned to the kernel in set i is j. In the first term, $\frac{\sum_{m=0}^{j-1} H_i(m)}{j}$ represents the cache hits per assigned cacheline when j cachelines are assigned to the kernel in set i. β is a tunable weight parameter and the denominator in the first term is a normalization factor. A larger first term indicates a better utilization of the assigned cachelines due to more covered cache hits per cacheline. The second term means the proportion of the cache hits covered by the j assigned cachelines to the cache hits covered by all the cachelines in the set. A larger second term indicates that more cache hits can be enabled. To assign a best suitable number of cachelines in a set to a kernel, not only the hits per assigned cacheline should be large, but also the assigned cachelines would better cover more hits. Therefore, combining the first and second terms with two tunable weight parameters β and γ, the cache benefit for assigning j cachelines in set i to the kernel can be measured with C_{ij}. β and γ can be chosen according to the degree of emphasis on different terms and both are set to 0.5 in this work. Considering all the sets together, K can be obtained following equation (8), which is the value for j among 1 to $Asso$ maximizing the sum of cache benefit metric C_{ij} for all sets i. K gives the best suitable cacheline number assigned to the kernel to make the cache resource utilization as efficient as possible, because of the maximal aggregated cache benefit. Accordingly, K_{coin} can be derived following equation (9), which quantifies the benefit per cacheline with the best assigned cacheline number K from a view of combining all sets. In summary, K decides the best partition size for the kernel and K_{coin} is a metric to quantify the potential benefit of allocating the partition into cache.

$$C_{ij} = \beta \frac{Asso \frac{\sum_{m=0}^{j-1} H_i(m)}{j}}{\sum_{m=0}^{Asso-1} H_i(m)} + \gamma \frac{\sum_{m=0}^{j-1} H_i(m)}{\sum_{m=0}^{Asso-1} H_i(m)} \tag{7}$$

$$K = \{j \mid maximize \sum_{i=0}^{Set-1} C_{ij},\ 1 \leq j \leq Asso\} \tag{8}$$

$$K_{coin} = \sum_{i=0}^{Set-1} \frac{\sum_{m=0}^{K-1} H_i(m)}{N_i \times K}/Set \tag{9}$$

A set of K, K_{coin}, K_{min} and K_{max} will be generated for each kernel based on its corresponding reuse distance histograms for all the sets in cache. Since each kernel corresponds to one partition, a set of Key values also describe the features of a partition, which will guide the dynamic control for bypassing and partitioning.

6. Dynamic Control

The dynamic control logic includes a bypassing control block, a partition table and a replacement block. The architecture of the dynamic control logic is shown in Figure 3, where the black arrows represent the data request/response flow of an access and the blue arrows represent the control interactions among different blocks. The bypassing control block is in charge of the bypassing decision making, partition insertion, partition replacement and partition retirement. Cache bypassing can be performed on the FPGA L1 cache when there is a cache miss. When the missing cacheline is fetched back from the L2 cache, it can bypass the FPGA L1 cache and be directly sent to the kernels without allocated in the cache. A bypassing decision is made in parallel with the normal cache access process, which maintains the low access latency of the cache. It means that when there is a memory request, the cache is accessed normally, meanwhile the bypassing control block makes the bypassing decision in case the cache access misses. The detailed rules of the bypass decision making will be introduced in Section 6.1. If the decision is not bypassing, under the cache miss case the replacement block will select an existing cacheline and replace it with the newly fetched cacheline according to the replacement policy described in Section 6.2. The bypassing decision is no use for the cache hit case.

Besides making the bypassing decision, the bypassing control block also performs the operations including partition insertion, partition replacement and partition retirement to control the cache utilization at the partition granularity to maximize the benefits of cache for all the kernels. To control the cache utilization of each partition, two states are defined for a partition in a set: $active$ and $negative$, and the states are recorded in the partition

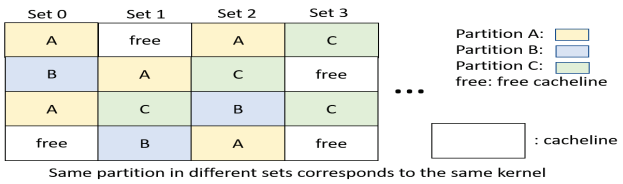

Figure 2: Example for Partitions in a 4-way Cache

table. An *active* partition is allowed to occupy the cache while a *negative* partition should not occupy the cache. An assigned size is given for each *active* partition, which represents the number of cachelines in a set assigned to the partition. The number of cachelines actually occupied by the partition will be controlled following the assigned size but may be different from it depending on the cacheline's availability. The assigned size of a *negative* partition is 0. In the beginning, all the partitions are *negative*. The *active* bit will be updated only when inserting or releasing a partition. Inserting a partition means changing a partition from *negative* to *active*, which indicates allocating this partition into cache while the assigned size will be decided for the inserted partition. Releasing a partition means changing it from *active* to *negative*, indicating this partition should no longer occupy the cache. Partition release happens during partition replacement or partition retirement to increase the $free\ space$ in the cache, which we will introduce in more details in Section 6.1 & Section 6.3. Partition replacement is to replace some existing *active* partitions for priority consideration and the counter based partition retirement mechanism is to release the partitions from cache when they are not accessed recently. Only updating the *active* bit and assigned size in the partition table is needed to insert or release a partition. For a cache set, the *capacity* is equal to the associativity of the cache. The sum of the assigned size for all *active* partitions will not exceed the *capacity*. $free\ space$ defines the number of remaining cachelines except the assigned ones in the *capacity*, which can be assigned to new partitions.

The bypassing control block and the replacement block will need to check the partition table for each partition's status and assigned size to make corresponding decisions. After execution of operations, for example, partition insertion, partition replacement and partition retirement, the bypassing control block will also update the partition table correspondingly. The detailed control steps of the dynamic control can be illustrated as in Figure 4, we will discuss these steps in the next subsections.

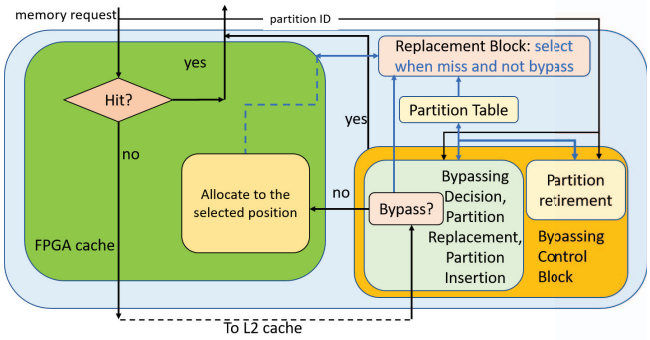

Figure 3: Architecture of Dynamic Control

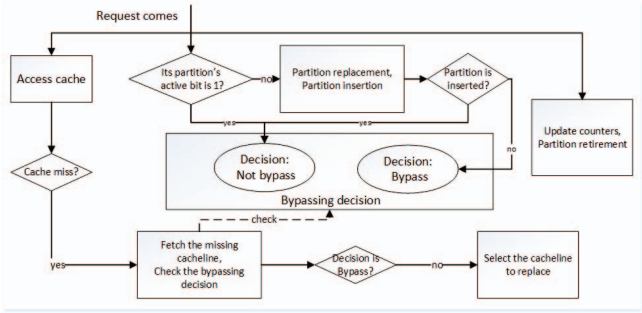

Figure 4: Dynamic Control Flow

6.1. Cache Bypassing Control

As discussed above, the operations of bypassing control block include the bypassing decision making, partition insertion, replacement and retirement when needed. When a memory request arrives, a partition ID is associated with the coming request, which is set by the requesting kernel to indicate which partition the request belongs to. With this partition ID, the partition table is checked by the bypassing control block. As shown in Figure 4, the *active* bit of the partition is checked first. If the *active* bit is 1, the request will not bypass the FPGA L1 cache, but the cacheline replacement will be incurred under the cache miss case according to the policy given in Section 6.2. But if the *active* bit is 0 which means the partition is *negative*, we should consider whether to insert this partition into the cache as analyzed in the next paragraph.

Algorithm 1 Bypassing Decision When an Access to Partition A is Made

```
1:  if A.active bit = 1 then
2:    return not bypass
3:  else
4:    for each active partition B do
5:      if B.K_coin + T < A.K_coin then
6:        release B :
7:        B.acitve bit = 0
8:        B.assigned size = 0
9:        update free space
10:     end if
11:   end for
12:   if free space >= A.K_min then
13:     insert A :
14:     A.active bit = 1
15:     A.assigned size = min(A.K, free space)
16:     update free space
17:     return not bypass
18:   else
19:     return bypass
20:   end if
21: end if
```

The partition with larger K_{coin} will benefit more from cache than a partition with smaller K_{coin}. Therefore, for the priority consideration, the *negative* partition for the coming request with larger K_{coin} can replace the current *active* partitions with smaller K_{coin}, which is called partition replacement so that the whole cache benefit would be enlarged. In the control scheme, we set a tunable threshold T. The current *active* partitions whose K_{coin} are smaller than K_{coin} of the coming partition minus T will be released. By releasing the lower priority partitions, the $free\ space$

will be expanded, so that the cachelines originally assigned to the lower priority partitions now can be assigned to the coming partition. In this way, the partition replacement can be achieved. The available $free\ space$ will be then compared with K_{min} of the coming partition. If the $free\ space$ is smaller than K_{min}, the coming partition will not be inserted into cache since from static analysis, we know that K_{min} is the minimal requirement for number of cachelines to enable data reuse for the corresponding kernel. Therefore, the request will bypass the FPGA L1 cache when its corresponding cacheline is fetched back from L2 cache, and the coming partition remains $negative$. If the $free\ space$ is no smaller than K_{min}, the partition should be inserted into cache with assigned size $min(free\ space, K)$. The partition table will be updated and the selected available cacheline will be replaced with the new cacheline of the coming partition under cache miss case. The bypassing decision and partition replacement/insertion control can be summarized in Algorithm 1.

Here, we notice that the low priority partitions will be released first even though the new partition may not be inserted. It is for the ease of the control and counting the available $free\ space$. Since the partition release only updates the status in partition table, it is easy to do the release. If the new partition is not inserted, when the low priority partition is requested again in the later memory access, its status can be quickly updated and it will still be $active$ in cache as it had not been released.

6.2. Partitioning Oriented Replacement Policy

A cacheline replacement policy is proposed to select the cacheline to be replaced when a non bypassing cache miss occurs. Partition insertion, partition replacement and partition retirement only update the partition table, they need go through cacheline replacement following the proposed replacement policy to achieve their desired status and control. By restricting which partition the replaced cacheline can be from, the number of occupied cachelines of each partition will be controlled, so that the cache partitioning can be achieved. A partition ID is associated with each cacheline in our scheme. If a cacheline is not occupied by any partitions, it's $free$. In the beginning, all cachelines are $free$. If a cacheline is occupied by an $active$ or $negative$ partition, the cacheline is called $active$ or $negative$, correspondingly.

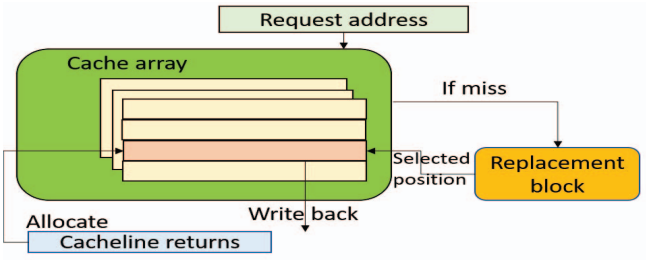

Figure 5: Process of Cacheline Replacement

The partition of an access not bypassing the FPGA L1 cache must be $active$. As shown in the Figure 4, there could be two situations for the not bypass decision which require the cacheline replacement. One is that the partition is just decided to be inserted. Another is that the partition is already $active$, but its existing cachelines do not contain the requested data and give a cache miss. No matter which situation, when the bypassing decision is not bypassing and there is a cache miss, the K_{max} and occupied cacheline number of this partition is compared first. If the number of occupied cachelines of this partition reaches K_{max}, no more cachelines should be occupied by this partition. Thus, the replaced cacheline can only be from the cachelines already occupied by this partition and we choose the LRU cacheline among all cachelines of this partition to be replaced. Note that the occupied cacheline number is the number of cachelines actually occupied by the partition and may be different from the assigned size.

Algorithm 2 Replaced Cacheline Selection when an Access of Partition A Results in a Miss

```
1: if A.occupied cachelines >= A.K_max then
2:     return the LRU cacheline of partition A
3: else if free or negative cacheline exists then
4:     return the LRU cacheline among free and
       negative cachelines
5: else if saturated partition exists then
6:     return the LRU cacheline among saturated
       partitions
7: else
8:     return the LRU cacheline of partition A
9: end if
```

Then if the number of occupied cachelines is smaller than K_{max}, this partition can occupy more cachelines. In this case, we first check whether there are $free$ or $negative$ cachelines since these cachelines are free to be used. If any such cacheline exists, the LRU cacheline among all eligible cachelines is replaced. Otherwise, whether there is saturated partition needs to be further checked. The saturated partition means the $active$ partition whose number of actually occupied cachelines are more than its assigned size. In our scheme, the number of cachelines occupied by a partition may exceed its assigned size when there are enough available cachelines, which enables a full utilization of all available cacheline resource. If there is any saturated partition, the LRU cacheline among the cachelines of all saturated partitions will be replaced, which can balance the cacheline occupation and alleviate the saturation. Then, if there is no $free$ or $negative$ cacheline and no saturated partition, each $active$ partition can only use the occupied cachelines of itself, since all other cachelines have been occupied by other non saturated $active$ partitions and cannot be used. Therefore, the LRU cacheline among the cachelines occupied by the partition of the missing request will be replaced. The complete partitioning oriented replacement control is demonstrated in Algorithm 2, which returns the selected cacheline to be replaced.

A replacement block is in charge of the selection of replaced cacheline with the help of the partition table. As soon as a cache miss occurs and the bypassing decision is made, the replacement block will start selecting the cacheline to be replaced. The selection will execute in parallel with the missing cacheline being fetched when a miss occurs, so that the latency can be overlapped with the large cacheline fetch latency.

6.3. Partition Retirement Mechanism

If an $active$ partition is not accessed for a long time, we release it from cache to make space for other partitions. This process is called partition retirement. A counter based method is applied for partition retirement. In a set, each partition owns one counter. When a memory request of a partition comes to the set, the counter of this partition will be set to 0. At the same time, the counters

for other $active$ partitions are all incremented by 1. In this way, the $active$ partition which is not accessed for a long time will have a large counter number. If the number of a counter is larger than a threshold M, its associated partition should retire and be released from cache. A retired partition is set to $negative$, so that its occupied cachelines are free to be used by other $active$ partitions.

The partition retirement is handled by the bypassing control block. When a request comes, besides the bypassing decision making, partition replacement and partition insertion described in Section 6.1, the bypassing control block will also update the counters and modify the partition table according to the partition retirement decision.

6.4. Overhead

In the partition table, for each partition in each set, there are one $active$ bit and one register for the assigned size. In addition, a register is used to record the $free\ space$ for each set. All the Key values are set in the partition table to be used by the replacement block and bypassing control block. For a 4-associative cache with 16KB capacity, only 116B resource is needed for the partition table when there are 4 kernels, which only brings an overhead of 0.7%.

Moreover, in the cache, each cacheline will be associated with a partition ID and the occupied cacheline number of each partition needs to be recorded in each set, which requires only 128B resource for a 4-associative cache with 16KB capacity when there are 4 kernels. This overhead is only about 0.7%.

The latency overhead of the dynamic control is also negligible. When a request comes, the bypassing control block executes in parallel with the normal cache access and the bypassing decision will be referred to only when the cache access misses. Therefore, the cache hit latency will not be affected. The cache miss latency will also not be affected much, since the bypassing decision can be obtained after some simple bits checking and comparisons, whose latency overhead is not only small compared with the large latency of fetching a cacheline from the L2 cache, but also can be overlapped with the normal cache access process. The partition table update including partition retirement, partition insertion and partition replacement will also execute in parallel and not affect the normal cache access process. Furthermore, the cacheline selection for replacement can be completed in parallel with the process of fetching the missing cacheline from L2 cache, so that the corresponding latency overhead can be overlapped with the large cacheline fetch latency. In this way, our scheme only introduces negligible latency overhead, which guarantees the efficiency of our scheme. The related detailed analytical data will be shared in a future version of this work.

7. Experimental Results

To assess the effectiveness of our approach, we compare our cache management scheme against a typical baseline design where the FPGA cache adopts the LRU replacement policy without cache bypassing and partitioning, which is commonly used in most designs and denoted as LRU in this paper. The baseline LRU replacement policy is proved to be advanced and near-optimal in [16]. The detailed experiment setup is explained as follows. The dynamic control module is implemented on a CPU-FPGA simulator PAAS [9], which supports cycle-accurate simulation of typical cache coherent CPU-FPGA systems. The accelerated functions are based on Polybench [11]. 12 different combinations of the benchmarks from Polybench are evaluated, where each combination combines 3 benchmarks as the accelerated kernels as defined in Table 1. Each combination is evaluated with the cache associativity of 1, 2, 4, 8, respectively, for the FPGA L1 cache. The system is configured with one CPU core and an FPGA fabric which holds three kernels with different acceleration functions. The FPGA L1 cache is set to be 16KB with the cacheline size of 64B. The shared L2 cache is set to be 256KB with 8 associativity. The write-back policy is applied to the caches. These are typical settings for coherent cache based CPU-FPGA systems. In addition, since our strategy is only applied to the FPGA L1 cache, it will be scalable with increased number of CPUs, FPGAs and kernels.

We calculate the FPGA L1 cache hit rate for each evaluation case, which equals to the number of FPGA L1 cache hits over the number of all the memory requests from kernels. The average hit rate on different associativities is shown in Figure 6 for each benchmark combination. The hit rate improvement is defined as the difference between the hit rate under our scheme and the hit rate under the baseline case LRU. With our scheme, the hit rate improvement can be from 2% to 37% with an average of 22.90% for all the evaluated cases. The average hit rate improvement is annotated in Table 1 for each benchmark combination. All the evaluated cases show a large improvement on the FPGA L1 cache hit rate, which indicates our scheme can successfully increase the FPGA cache hit by a large degree and alleviate the cache contention for cache coherent CPU-FPGA systems.

As the FPGA cache hit rate increases, more requests can be satisfied by the FPGA L1 cache. The number of FPGA cache misses and the number of requests from FPGA to the shared L2 cache will be reduced significantly. Handling the FPGA cache misses and requesting data from FPGA to the L2 cache are time consuming and power consuming, especially for the cache coherent CPU-FPGA system where the L2 cache is usually located at the CPU side and with more distance from FPGA. Therefore, with our scheme, the performance and energy efficiency can be improved by a large degree.

The speedup is defined as the percentage of the execution time difference between our scheme and the baseline case LRU over the baseline case execution time. For all the evaluated cases, the speedup can be up to 12.52% with an average of 4.99%. We annotate the average speedup for each benchmark combination in Table 1. As we can see, our scheme can reduce the total execution time a lot by alleviating the FPGA cache contention. Moreover, if the number of kernels increases or more memory requests are sent from kernels, the cache contention will become severer with increased cache miss rate. In this case, our proposed strategy which can alleviate the cache contention and increase the cache hit rate will benefit more, thus can bring even more speedup.

8. Conclusion

In this work, a static and dynamic combined cache management strategy is proposed for emerging cache coherent CPU-FPGA platforms, which is aimed at a better utilization of the FPGA L1 cache. An automatic LLVM pass is first developed to analyze the accelerated functions with reuse distance theory and generate sets of Key values for describing the memory access patterns. With the generated Key values, a novel dynamic control mechanism is ap-

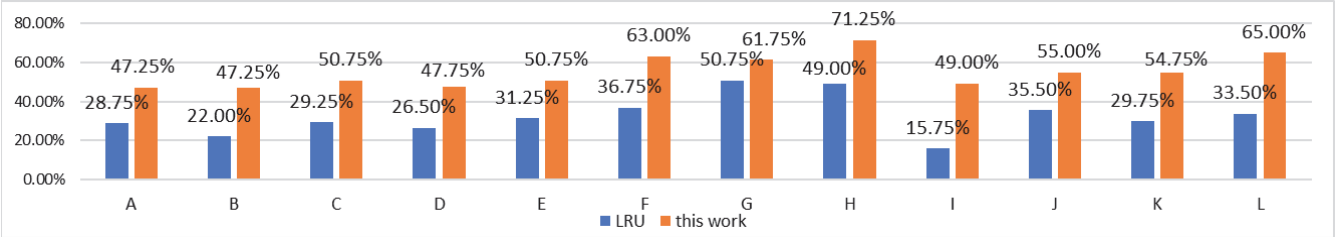

Figure 6: Average Hit Rate of the Evaluated Benchmark Combinations

Table 1: Average Hit Rate Improvement and Average Speedup for the Benchmark Combinations

combination	A	B	C	D	E	F	G	H	I	J	K	L
combined benchmarks	trmm floyd warshall gesummv	floyd warshall covariance trmm	gemm floyd warshall trmm	trmm gesummv covariance	gemm covariance trmm	seidel 2d floyd warshall covariance	gemm seidel 2d floyd warshall	covariance gemm seidel 2d	floyd warshall gesummv covariance	gemm floyd warshall gesummv	covariance gemm floyd warshall	gesummv covariance gemm
hit rate improvement	18.5%	25.25%	21.5%	21.25%	19.5%	26.25%	11%	22.25%	33.25%	19.5%	25%	31.5%
speedup	4.3%	5.9%	2.7%	3.1%	5.1%	7.0%	5.9%	3.7%	4.9%	5.1%	8.2%	4.0%

plied to flexibly control the cache bypassing and cache partitioning at runtime to alleviate the cache contention and increase the cache hit rate. To the best of our knowledge, this is the first work aimed at the cache management for cache coherent CPU-FPGA systems. As well, different from previous cache management methods, this work comprehensively integrates reuse distance theory, cache bypassing and cache partitioning through a static and dynamic combined strategy. Experiments on PAAS show that the proposed scheme can significantly increase the hit rate for the FPGA cache by 22.90% on average.

Acknowledgments

This research is supported by the funding GRF 16245716.

References

[1] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.*, 5(2):78–101, 1966.

[2] Kristof Beyls and Erik DHollander. Reuse distance as a metric for cache behavior. In *Proc. 13th IASTED Conf. Parallel and Distributed Computing and Systems (PDCS)*, volume 14, pages 350–360. ACM, 2001.

[3] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. 25th ACM Int'l Conf. Supercomputing (SC)*, pages 402–412. ACM, 2014.

[4] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proc. 45th Annual IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pages 389–400. IEEE, 2012.

[5] PK Gupta. Accelerating datacenter workloads. In *26th Int'l Conf. Field Programmable Logic and Applications (FPL)*. IEEE, 2016.

[6] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Computers*, 57(4):433–447, 2008.

[7] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. 13th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122. IEEE, 2004.

[8] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. Optimal bypass monitor for high performance last-level caches. In *Proc. 21st Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pages 315–324. IEEE, 2012.

[9] Tingyuan Liang, Liang Feng, Sinha Sharad, and Wei Zhang. Paas: A system level simulator for heterogeneous computing architectures. In *27th Int'l Conf. Field Programmable Logic and Applications (FPL)*. IEEE, 2017.

[10] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. 14th IEEE Int'l Symp. High Performance Computer Architecture (HPCA)*, pages 367–378. IEEE, 2008.

[11] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[12] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. 41st ACM/IEEE Int'l Symp. Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[13] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. 39th Annual IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pages 423–432. IEEE, 2006.

[14] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *Proc. 2017 ACM Int'l Conf. Management of Data (SIGMOD)*, pages 1659–1662. ACM, 2017.

[15] W. Snyder. Verilator: the fast free verilog simulator. *URL: http://www.veripool.org*, 2012.

[16] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Computers*, 41(9):1054–1068, 1992.

[17] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1):7–1, 2015.

[18] Youfeng Wu, Ryan Rakvic, Li-Ling Chen, Chyi-Chang Miao, George Chrysos, and Jesse Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *Proc. 35th Annual IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pages 134–145. IEEE, 2002.

[19] Chi Zhang, Ren Chen, and Viktor Prasanna. High throughput large scale sorting on a cpu-fpga heterogeneous platform. In *Proc. 2016 IEEE Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, pages 148–155. IEEE, 2016.