CASPHAr: Cache-Managed Accelerator Staging and Pipelining in Heterogeneous System Architectures

Mochamad Asri, Andreas Gerstlauer, *Senior Member, IEEE*

***Abstract*—Integrating accelerators onto the same chip with CPUs sharing the last level cache (LLC) is beneficial when CPUs and accelerators frequently exchange data. However, if shared data exceeds LLC capacity, expensive spills to and refetches from DRAM will be incurred, limiting the benefits of such integrated architectures. While this can be avoided through careful software optimizations, such as fine-grain data tiling and accelerator synchronization, this involves significant software changes and programmer effort.**

In this paper, we introduce CASPHAr, a LLC architecture that performs automatic, software- and hardware-transparent fine-grain data staging and synchronization between CPUs and accelerators in hardware. CASPHAr tracks and synchronizes producer and consumer accesses at cache line granularity. As soon as some fraction of shared data is produced and becomes ready in the LLC, the data will be delivered for processing in the waiting consumer. Furthermore, CASPHAr extends existing replacement policies to leverage synchronization information for making better eviction decisions. All combined, CASPHAr reduces data spills due to unnecessarily long lifetimes of shared data in the cache. In addition, fine-grain staging and synchronization inherently achieves system-level pipelining of interdependent kernels that can outperform software optimizations. Results show that CASPHAr can boost performance by up to 23% and achieve energy savings of up to 22% over baseline accelerations.

***Index Terms*—Accelerator-rich architectures, Last level cache**

I. Introduction

Accelerator-rich, heterogeneous system architectures have arisen as a promising approach to provide massive compute capabilities with high energy efficiency. A key challenge in such systems is the efficient movement and exchange of data shared between components. When accelerators are integrated as off-chip devices placed near or in DRAM, any data exchanged between host CPUs and accelerators has to travel between on- and off-chip memories. Resulting round-trip data movement overhead can account for up to 50% of total system energy [1].

Recent trends have proposed placing accelerators on the same die as CPUs sharing the last-level cache (LLC) [2]. Such integrated, on-chip heterogeneous architectures can minimize or completely avoid unnecessary off-chip DRAM transfers by allowing applications that frequently exchange data to share data through the LLC. However, if the amount of data exchanged between CPUs and accelerators exceeds LLC capacity, expensive spills to and refetches from DRAM will still take place. This can negate the performance and energy benefits of on-chip integration.

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2022 and appears as part of the ESWEEK-TCAD special issue. This work was supported by NSF grant CCF-1725743.

Mochamad Asri is with Meta, Inc. The work was performed while he was at UT Austin. Andreas Gerstlauer is with the Electrical and Computer Engineering Department at the University of Texas, Austin, TX 78712, USA.

Costly spills can be avoided by carefully optimizing the software to match hardware constraints. Problem sizes can be chunked into smaller pieces that fit into the LLC to reduce DRAM spills to and refetches. Data partitioning and locality-aware data placement in the memory hierarchy are widely exploited to maximize performance gains, commonly referred to as tiling or blocking optimizations supported by a high-performance software library or a compiler [3], [4], [5], [6], [7]. Using such optimizations, CPUs can stage data and invoke accelerators repeatedly at finer sub-block granularity matching LLC size [8]. For further performance gains, accelerator staging on CPUs and executions of accelerators themselves can be overlapped and pipelined across different sub-block iterations. However, such fine-grain, block-level data staging optimizations place a significant burden on the programmer, and software has to be potentially re-written and re-optimized for every new application or hardware generation.

Hardware-assisted accelerator synchronization and data staging schemes have been proposed in prior work instead. Full/empty bits associated with each main memory location [9], [10] were employed early on to provide efficient fine-grain synchronization between processors. More recently, automatic handling of such producer/consumer synchronization has been integrated into memory [11] and DMA [12] controllers shared between CPUs and accelerators to enable fine-grain staging and overlapping transparently in hardware without software or programmer involvement. However, existing memory-level data sharing schemes do not address the fundamental problems associated with integrating accelerators at the cache level. Expensive spills and refetches from/to DRAM still occur with such schemes since data can only be exchanged at the DRAM level. Providing an efficient hardware-assisted, fine-grain data staging and synchronization for on-chip integrated accelerators poses a set of uniquely different challenges. Efficient mechanisms to track the synchronization status of individual memory locations directly within the shared LLC structure across both resident and evicted cachelines are required. Furthermore, minimizing data spills within limited LLC capacities requires approaches for intelligently managing producer-consumer locality and data residency.

Towards those goals, we introduce CASPHAr[1], a LLC architecture for *Cache-Managed*, Fine-Grain Accelerator Staging and Pipelining in On-Chip Heterogeneous Architectures. CASPHAr supports transparent, efficient and seamless data sharing between CPUs and accelerators. CASPHAr tracks, synchronizes and coordinates production and consumption of data at cache-line granularity directly in the LLC. Using existing cache interfaces and stall mechanisms, CASPHAr provides drop-in

[1]Pronounced as "Casper", the friendly, transparent ghost.

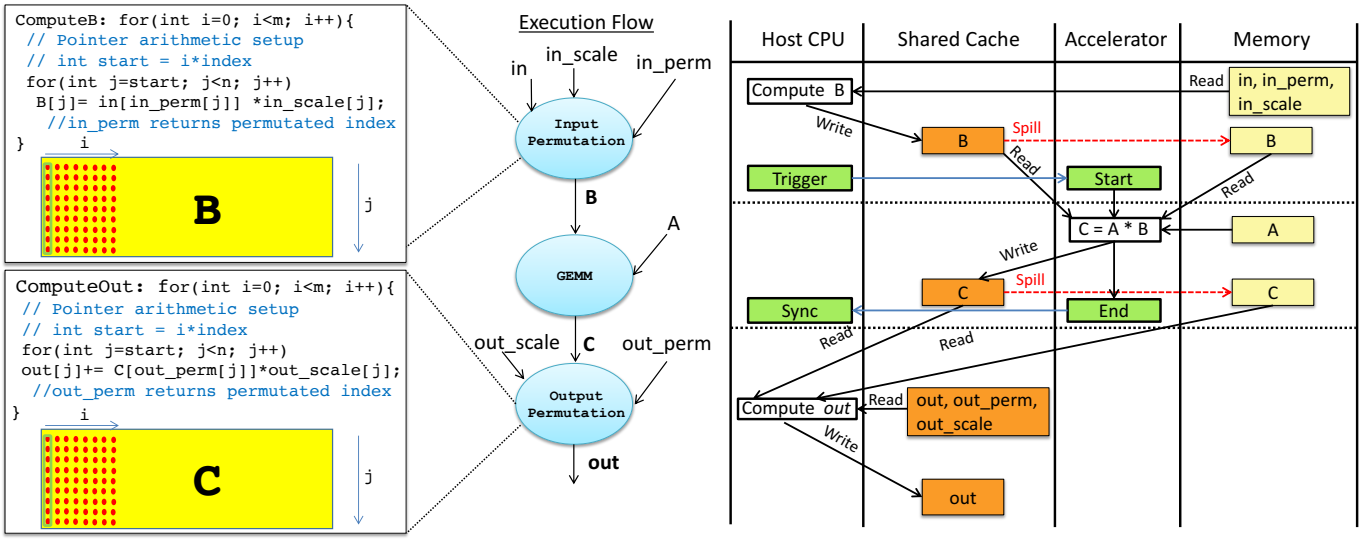

Fig. 1: Data flow graph (DFG) and data movement patterns of a typical accelerated application.

synchronization and staging support for on-chip integrated architectures without other hardware, accelerator and only minimal software changes. A shared cache line is marked as ready for any dependent consumer to access as soon as it is produced. In doing so, CASPHAr reduces data spills due to unnecessarily long lifetimes of shared data in the cache caused by accelerator staging at too coarse granularity. Moreover, fine-grain synchronization inherently enables overlapping and pipelining of producer and consumer executions.

CASPHAr achieves such capabilities specifically with three main techniques: 1) CASPHAr extends tag store bits in a conventional cache architecture to enable producer-consumer synchronization. 2) CASPHAr employs eviction range registers to keep track of ready/staged cachelines that might have gotten evicted without being consumed. In addition to tracking, eviction registers are used to avoid unnecessary refetching of evicted cachelines that are not ready/staged. 3) Finally, CASPHAr extends existing replacement policies in the LLC to synergistically and cooperatively leverage producer-consumer information in order to manage data residency and evict staged cache lines as soon as they are consumed.

In summary, our paper makes the following contributions:

- To the best of our knowledge, we are the first to address the problem of *cache-managed* data movement and coordination among heterogeneous system components in on-chip integrated architectures.
- We propose a novel last-level cache architecture for heterogeneous on-chip systems, CASPHAr, that provides automatic and programmer-transparent accelerator staging and pipelining optimizations in hardware. CASPHAr uses extended tag stores and eviction range registers to track and synchronize producer and consumer accesses at fine cache line granularity. This minimizes spilling and unnecessary refetching of shared data to DRAM while maximizing execution concurrency among components.
- We introduce extensions to existing cache replacement policies that are integrated with CASPHAr to improve data residency and eviction decisions. Leveraging synchronization information, existing policies can be extended to evict staged lines as soon as they are consumed, which minimizes spills due to capacity conflicts with other staged or non-staged lines.
- Using a cycle-accurate simulator [13], we demonstrate that CASPHAr can improve performance by up to 23% and reducing energy consumption by 22% compared to baseline acceleration while achieving results similar to or better than manual system-level data movement optimizations in software.

The rest of this paper is organized as follows: after a motivational example and a discussion of related work in Sections II and III, Sections IV and V present an overview of CASHPHAr and its main design, respectively, followed by evaluation results in Section VI. Finally, Section VII concludes the paper with a summary and outlook.

II. Motivation

To better understand the effects of system-level data movement overheads and potential optimization benefits, we evaluate a typical computation and data flow pattern in an accelerated system. We use the so-called U-List computation in a Finite Multipole Method (FMM) application [6], [7] as a driving example that is representative of a wide range of patterns in scientific and accelerated computing [8]. FMM belongs to the broad class of N-body methods used in computational physics and machine learning [14], [15], [7], [6], [16]. The U-List computation approximates a dense, $O(N^2)$ matrix-vector multiplication by a sparse, $O(N)$ matrix-vector multiplication.

Figure 1 shows the three main parts of the FMM U-List computation: pre-processing, the main kernel performing a General Matrix-Matrix Multiplication (GEMM), and post-processing. The preprocessing step applies a permutation to the input array (*"in"* in Figure 1) and assembles the result into matrix B. This step mainly consists of light-weight computation along with data shuffling and pointer chasing. As can be seen from the right side of the figure, pre-processing is normally performed on the CPU to produce B as input for the GEMM.

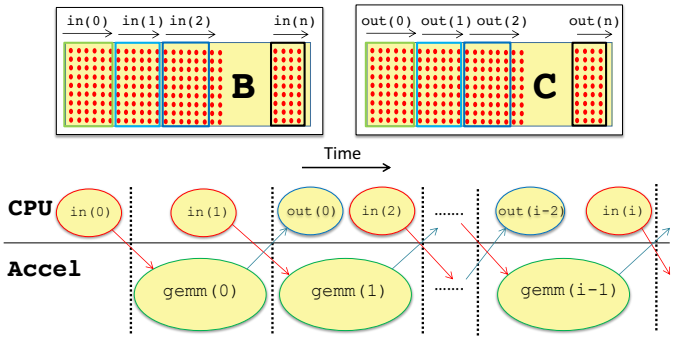

Fig. 2: Execution with software blocking & pipelining.

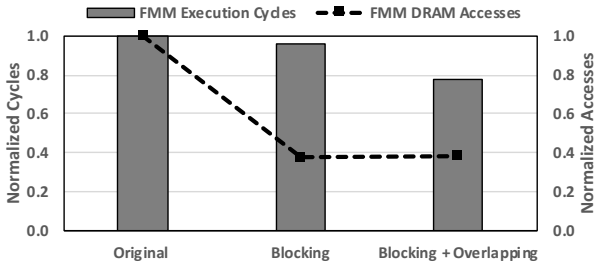

Fig. 3: Execution cycles and DRAM accesses of original vs. software optimizations.

The GEMM kernel is then performed on an accelerator, which will use matrix *B* as one of its inputs to produce matrix *C* as result. Note that, depending on the size of *B* and the size of the shared cache, some fraction of *B* might get evicted and spilled to the main memory before it can get used by the accelerator. In this case, the accelerator has to load some fraction of matrix *B* from main memory. Once the GEMM is complete, matrix *C* is sent to the host CPU, which performs a post-processing computation to assemble the final result in the array labeled *"out"*. Similarly, if the size of *C* is larger than the cache, a fraction of *C* might get evicted and spilled to the main memory before the host CPU can access it. In this case, the host CPU will experience cache misses that incur expensive main memory accesses to re-load the spilled data.

From a data movement perspective, all data exchanges of matrices *B* and *C* between CPU and accelerator should ideally happen exclusively through the LLC without expensive spills to main memory. One approach to avoid costly spills is to tile the data into smaller pieces and perform accelerator invocations on those blocked tiles. Moreover, blocking the problem size into sub-kernels with smaller granularity has the added benefit that software pipelining can be applied to exploit additional overlapping and parallelism between accelerator and CPU, as shown in Figure 2. Instead of processing the entire matrices *B* and *C* at once, the CPU and accelerator operate on smaller, blocked versions of matrix *B* and matrix *C* whose total size is equal or smaller than the LLC capacity. In doing so, expensive data spills to the main memory are avoided and all data remains resident in the LLC for as long as it is needed by either component. Furthermore, with the *i*-th sub-block iteration of input permutation, GEMM, and output permutation tasks denoted as *in(i)*, *gemm(i)*, and *out(i)*, respectively, the accelerator is operating on one sub-block of the GEMM while the CPU simultaneously performs output and input permutations for the sub-blocks of *B* and *C* from the previous and next iterations, respectively.

Figure 3 shows the performance gains and total number of DRAM accesses under software blocking and pipelining optimizations as compared to a naive acceleration. As can be observed, software blocking reduces DRAM accesses by more than 2x. By chunking the problem size into smaller pieces, such a blocking approach can keep the shared data resident in the LLC while it is alive, such that most of the data is exchanged through the LLC. In addition, software pipelining on top of blocking can exploit additional overlapping and parallelism between accelerators and CPUs, reducing execution cycles by up to 30% over a naive acceleration.

In conclusion, blocking optimizations can benefit in two ways: (i) reducing DRAM accesses, and (ii) opening software pipelining opportunities to exploit additional overlapping and parallelism between components. However, such a software-driven approach puts a significant burden on the programmer. The software effort of dealing with fine-grained data staging and synchronization can quickly outweigh the benefits and hinder widespread adoption of accelerator-rich on-chip heterogeneous architectures. In the next sections, we detail our cache-centric approach to exploit (i) and (ii) transparently with minimal software changes and to tackle associated design challenges in on-chip integrated architectures.

III. Related Work

Several prior works optimize data movement using software- and hardware-centric solutions.

Software-centric optimizations: Manual, compiler or library based support has been proposed to exploit parallelism and regular communication patterns in stream programs by formulating a set of cache-aware optimizations that improve instruction and data locality [3], [4], [5], [6], [7], [8]. All such approaches unlock system-level gains and data movement reductions at the expense of increased software burden.

Hardware-assisted approaches for dependent kernels: On the hardware/architecture side, several mechanisms have been proposed to achieve fine-grain data staging optimizations between interdependent kernels that can avoid caveats of software-based approaches. The work in [11] has employed full/empty bits in DRAM managed by a shared memory controller to overlap host-to-device memory copies with computation. They assume a discrete memory space between CPU and accelerator and thus are limited to the context of explicit data copy/transfer between CPU and accelerator private memories. The scheme in [12] instead extends a DMA controller with local full/empty information to synchronize data transfers at cacheline granularity. However, they require modifications on the accelerator side to comply with their design, and staging support is limited to one direction (i.e., CPU producer and accelerator consumer) in a DMA context with specific transfer patterns. In both schemes, data is shared through off-chip memories. Data exchanges still have to happen at the DRAM level and management of limited on-chip storage capacities is not addressed. The work in [17] has proposed throttling

of producers and consumers to avoid spills from last-level caches. Such an approach is orthogonal to our work. CASPHAr tackles cache-managed synchronization. By contrast, throttling still requires synchronization, but can be applied on top of CASPHAr to provide further gains. More recently, [18], [19], [20] proposed methods and architectures to optimize coherence interfaces for many-accelerator SoCs. However, coherency management among private/shared caches is orthogonal to data staging and synchronization issues in CASPHAr.

Hardware-assisted approaches for independent kernels: In the context of heterogeneous systems, Lee *et al.* demonstrate a cache management policy that is aware of differences in latency sensitivity and thread-level parallelism of CPU and GPU applications [21]. Li *et al.* propose a finer-grained cache management policy to cope with heterogeneous CPU-GPU accesses, leveraging behavior variations among various LLC sets [22]. In particular, CPU and GPU requests are prioritized disparately in each LLC set during cache block insertion and promotion, based on per-core utility behavior and per-set CPU-GPU miss counters. Such approaches have significant advantages in that they boost performance through microarchitecture techniques that are user-transparent. However, these works all assume that CPUs and GPUs are co-running independent tasks where no data sharing issues are involved. CASPHAr is conceptually different from these approaches. CASPHAr addresses data-sharing problems that arise when interdependent tasks frequently exchange data.

Overall, existing works fall into either software-centric optimizations or hardware optimizations with assumptions of no or little DRAM-level sharing. Software-centric approaches suffer from the cost of significant programmer effort in dealing with system-level optimizations such as synchronization, pipelining and overlapping. Existing hardware- or microarchitecture-driven optimizations have not considered system-level considerations of frequent data sharing between CPUs and accelerators, or are limited to the context of off-chip integration at the DRAM level. To the best of our knowledge, we are the first to address the problem of cache-managed system-level data movement coordination and orchestration in on-chip heterogeneous architectures.

IV. CASPHAr Overview

In this section, we provide an overview of the key CASPHAr concepts. We first explain the basic system architecture on which CASPHAr is based. We then introduce the CASPHAr paradigms on top of this base architecture. Finally, we demonstrate how CASPHAr interfaces with software.

A. Base System Architecture

Figure 4 shows the baseline system architecture we assume in this work. The accelerator shares the LLC with CPU cores. It is coupled at the interconnect network between the L2 cache and LLC in the same way as other cores are interconnected in multicore systems. CASPHAr is independent of the coherency management approach used. For the remainder of this paper we assume accelerators to be non-coherent, which is a system-architecture design commonly used in previous work [23], [24]. Since the accelerator itself explicitly manages all memory

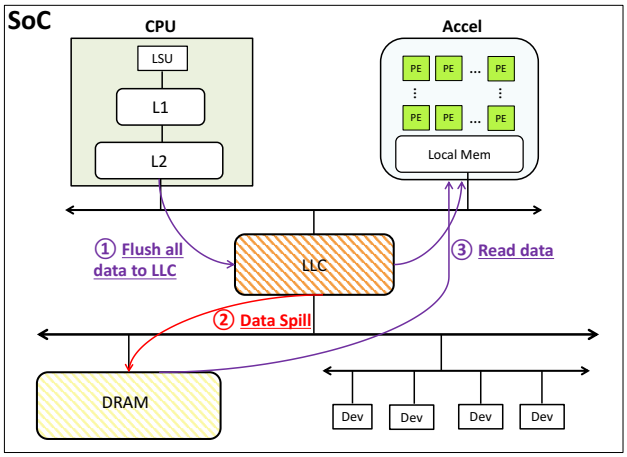

Fig. 4: Baseline system architecture.

and internal scratchpad accesses, it is typically desirable to avoid the overhead of having it participate in the coherency protocol. Previous work has shown that enforcing coherency between CPU and accelerators can have significant overhead, generating significant bus and coherency traffic that potentially limits acceleration benefits [25], [26].

Instead, we assume that the accelerator expects necessary data to be in the LLC or DRAM when it starts computing. Consequently, the programmer through the software ensures explicit flushes of any dependent data to the LLC before triggering the accelerator. This is a similar approach used in previous work [23], [27], [28], [29]. As can be seen from Figure 4, the CPU stages data to the accelerator by first flushing it to the LLC. The accelerator is synchronized to not start reading the data until the CPU completes flushing the complete staged data. If the data footprint is below the LLC capacity, all staged data will be resident in the LLC. However, if the data size exceeds LLC capacity, some data will get spilled to DRAM. Thus, when the accelerator subsequently reads the data, it may find a fraction of it in the LLC and some other part in DRAM due to the earlier spills.

Similarly, once the accelerator is finished computing, it writes the result back to the LLC for the CPU to access. If result size exceeds LLC capacity, as discussed previously, spills to DRAM will be incurred, and the CPU may experience LLC misses requiring some of the data to be fetched from memory when is starts accessing the results.

B. CASPHAr System Architecture

In contrast to the base architecture, CASPHAr frames the data staging and synchronization as a fine-grain in-cache producer and consumer transaction. CASPHAr allows a producer to stage data and synchronize with the consumer at cache line granularity. Therefore, instead of waiting for the full data footprint to be written to the LLC, CASPHAr enables CPUs and accelerators to exchange any dependent data as soon as it becomes ready.

Figure 5 illustrates the data movement flow in CASPHAr. The CPU stages data to the accelerator by storing individual items normally into a shared memory address (*Write*). Once the CPU has finished writing a location, i.e. after the last write

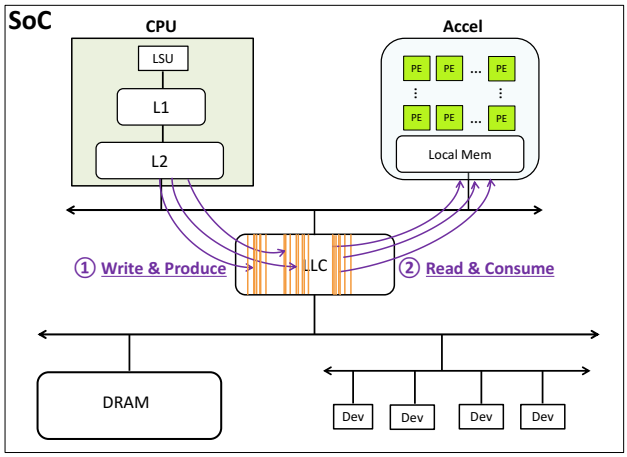

Fig. 5: CASPHAr system architecture.

to a shared memory address, the CPU marks the data line as ready in the LLC (*Produce*). In the process, the CPU needs to ensure that the produced data is available in the shared LLC. Consequently, the CPU needs a way to simultaneously flush data to the LLC and mark it as produced.

The x86 ISA has a software flush instruction (`clflush`) that evicts the targeted line all the way to DRAM, i.e. if the block exists in the LLC, it will be invalidated and further evicted. In our scenario, we wish to manage the eviction and keep or store the line in the LLC by not invalidating and potentially updating an already existing, or by allocating a new LLC line that potentially replaces and evicts others. At the same time, we need to provide a mechanism to indicate that a flushed line has been produced. There are two plausible options to support this: (1) adding a special instruction in the ISA that handles both eviction only to the LLC and flagging as produced, or (2) setting a configurable register in the LLC controller to identify a range of addresses that should not be invalidated but kept/stored and flagged when software eviction happens. We opted for the latter implementation due to its simplicity and transparent change from the software perspective. This approach will also not require any modifications of the accelerator design for cases when the accelerator is the producer.

CASPHAr then tracks the produced status of individual cache lines in the LLC. It tracks the status even if a cache line is spilled to and later re-fetched from main memory. The accelerator in turn accesses any shared memory location in a normal fashion (*Read*). If a shared location has not been produced yet, CASPHAr will indicate a standard miss condition and use regular miss handling mechanisms to block or delay the access until the data is ready. Finally, once the accelerator has finished accessing the shared location, it resets the line back to the default status (*Consume*). We assume that accelerators operate out of local scratchpads and read/write shared locations only once. Under this assumption, no modifications of accelerators are required to explicitly mark lines as produced or consumed.

Communication of accelerator results back to the CPU follows a similar flow, but in the opposite direction, i.e. with the accelerator being the producer and the CPU consuming shared data. In this case, the CPU needs a mechanism to indicate that a shared line has been finally consumed, i.e. that the last access has occurred. Since this is equivalent to marking the line as candidate for eviction, we utilize the same evict-to-LLC instruction to indicate both production and consumption of data. Note that this will also evict data from L1 and L2 caches to the LLC on consumption. However, since consumption is equivalent to indicating that data will not be accessed further, it is actually beneficial to evict it from and thus free up space in the higher-level caches.

All combined, unlike the base architecture, a consumer in CASPHAr does not need to wait until the full data region has been staged. Instead, it can access any cache line as soon as it is produced. Hence, data spills occurring in the base architecture due to long lifetime of shared data in the cache caused by coarse-grain data staging and synchronization can be avoided. Moreover, CASPHAr takes advantage of such fine-grain synchronization by inherently allowing accelerator executions to overlap with data staging on the CPU in an out-of-order/dataflow style. As a result, CASPHAr allows for system-level pipelining of interdependent kernels to be seamlessly achieved through its microarchitecture.

Overall, CASPHAr acts like a synchronized shared buffer of arbitrary size between producers and consumers that is realized transparently between LLC and DRAM. Note that CASPHAr supports irregular, random and non-conforming access patterns of producers and consumers. However, achievable benefits depend on the order and timing of shared memory accesses made by the involved components. CASPHAr will be able to avoid data spills as long as the maximum number of simultaneously alive (i.e. produced but not yet consumed) shared data items is smaller than the LLC size. It will remain beneficial as long as the peak shared buffer fill state is smaller than the complete shared data region size. In the worst-case, producers and consumers access items in opposite order, or consumers do not start processing data until producers are already finished. In such cases, CASPHAr will automatically fall back to a default base architecture execution.

C. Software Interface

To fully leverage the CASPHAr concept, software will require only minimal modifications. CASPHAr envisions physically contiguous memory regions dedicated to data exchanged between CPUs and accelerators. Any data structures that are shared between components are physically mapped to this region. Such an approach is widely used to simplify accelerator data exchange [30], [31]. The programmer can access the shared regions from conventional user-space virtual addresses, which are mapped to a physically contiguous memory region by the OS. CASPHAr in turn allows the user program or OS to configure the start and end address of the shared regions through a set of memory-mapped configurable register in the LLC controller. The CASPHAr cache controller will then use this information to identify if a cacheline access belongs to a shared region and hence requires tracking.

Such an approach does not require any modifications to the basic application code. However, to make use of fine-grain synchronization and overlapping opportunities, the accelerator

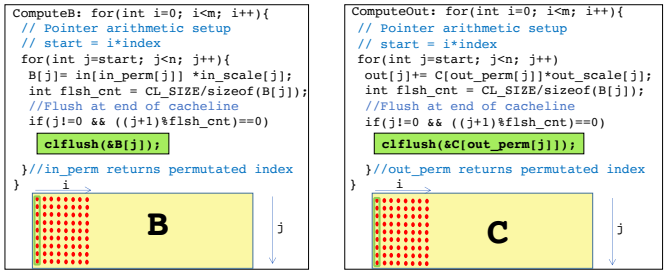

Fig. 6: CASPHAr programming model.

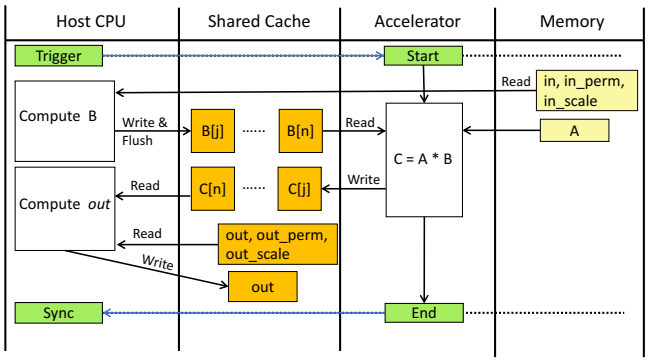

Fig. 7: Execution flow on a CASPHAr system.

must be invoked to execute in parallel to the CPU. As such, triggering of the accelerator must be hoisted to before the start of CPU pre-processing/data preparation, and synchronizing with its completion must be delayed until after post-processing of the results. In the process, the software will also set and reset the region registers in the LLC to enable and disable fine-grain tracking and synchronization of shared region accesses. Furthermore, the software must flag fine-grain production and consumption of data during pre-/post-processing.

Figure 6 and Figure 7 summarize the software modifications and resulting execution flow on a CASPHAr system. The host CPU triggers the accelerator, and the accelerator immediately begins its execution. The host CPU and the accelerator can then execute in parallel, with all producer-consumer transactions tracked and managed seamlessly by the LLC. Once the software has finished writing the full cacheline required by the accelerator, it uses a *clflush* instruction to flush the data/cacheline and mark it as produced in the LLC. This requires inserting corresponding primitives into the application code. Once the CPU finishes reading any data produced by the accelerator, *clflush* instructions are inserted to mark data as consumed and invalidate it in its local L1 and L2 caches.

D. Discussions and Possible Variations

In CASPHAr, the software has to explicitly mark if a cacheline is produced/consumed using `clflush` instructions. This can be automated using compiler support as demonstrated in prior work [32], [33], [34]. For accelerators that do not exhibit single-read/write behavior, an explicit clflush-like mechanism is needed to mark cache lines. For cases where producer and consumer data access patterns are irregular and/or data-dependent and thereby it is difficult for the compiler to statically determine the last write/read to/from a cacheline, an explicit synchronization hint from the programmer may still be required. Alternatively, marking a cacheline produced or consumed could be associated with an event such as the end of a kernel, instead of at single read/write granularity. Finally, for code where using producer-consumer patterns is particularly unsuitable, user can default to existing software-based synchronization mechanisms.

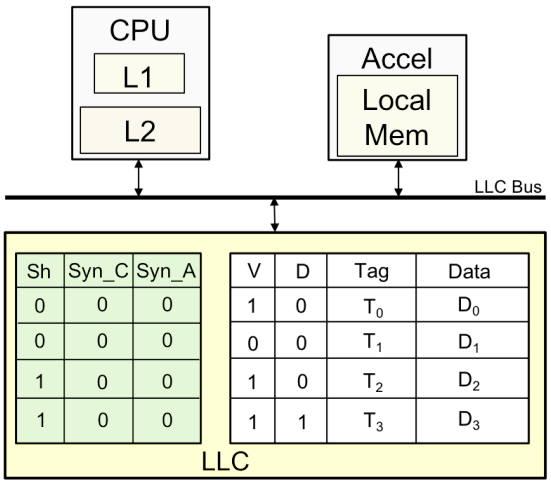

Fig. 8: CASPHAr LLC architecture.

V. CASPHAr Design

In this section, we describe the CASPHAr cache architecture that implements our fine-grained data staging and synchronization. We start by describing the microarchitecture of CASPHAr and its fine-grain staging and synchronization flow. We then detail techniques to tackle challenges in performing such synchronization in the LLC, such as a tracking mechanism of staged data during residence as well as eviction, and CASPHAr-aware extensions to existing replacement policies.

A. Base Cache Architecture

CASPHAr extends a conventional cache architecture to support CPU/accelerator synchronization at cacheline granularity. In order to do so, additional meta-data bits are added to the cache structure for every cacheline (Figure 8).

The *Sh* bit indicates a shared cache line between the CPU and accelerator. As mentioned before, CASPHAr requires that any data structures shared between the CPU and the accelerator to be physically mapped to contiguous memory regions as similarly adopted in related works [30], [31]. CASPHAr distinguishes between shared regions for data transfers from CPU to accelerator and for transfers from the accelerator back to the CPU. CASPHAr is equipped with a set of memory-mapped configuration registers in the LLC controller to mark the start and end address of the two shared regions from user or OS space. The cache controller uses this information to identify if a cache line access belongs to a shared region and which component is producer and consumer. If a line is in the shared region data, its *Sh* bit is set to 1.

Syn_C is a bit for synchronization when the CPU is the producer and the accelerator is the consumer. It is set to 1 when the CPU has produced the latest data ready to be consumed by the accelerator, i.e. when the CPU issues a *clflush* instruction. When the accelerator attempts to access the line, CASPHAr

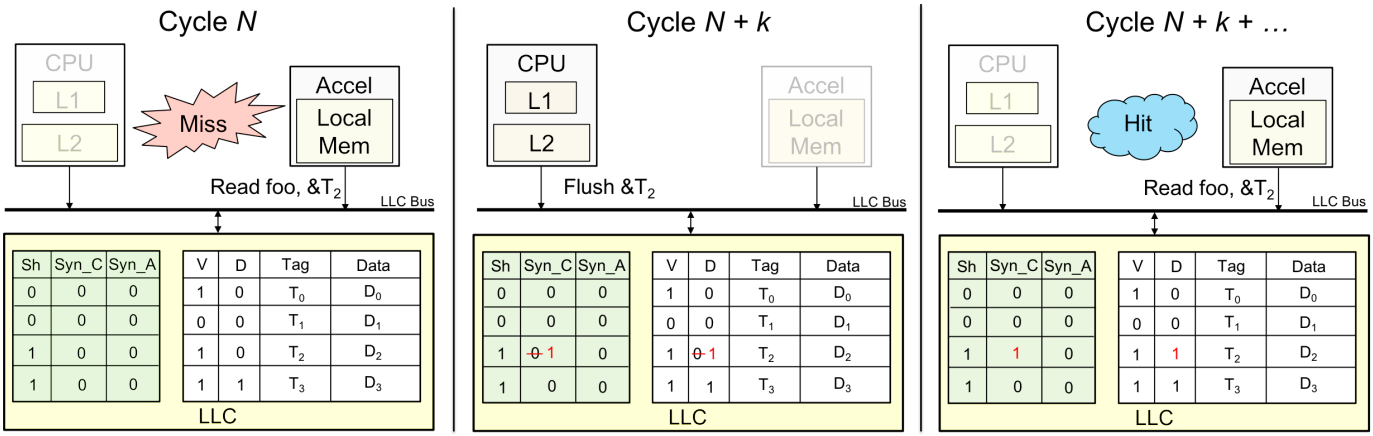

Fig. 9: CASPHAr synchronization on a synchronization miss but cache hit.

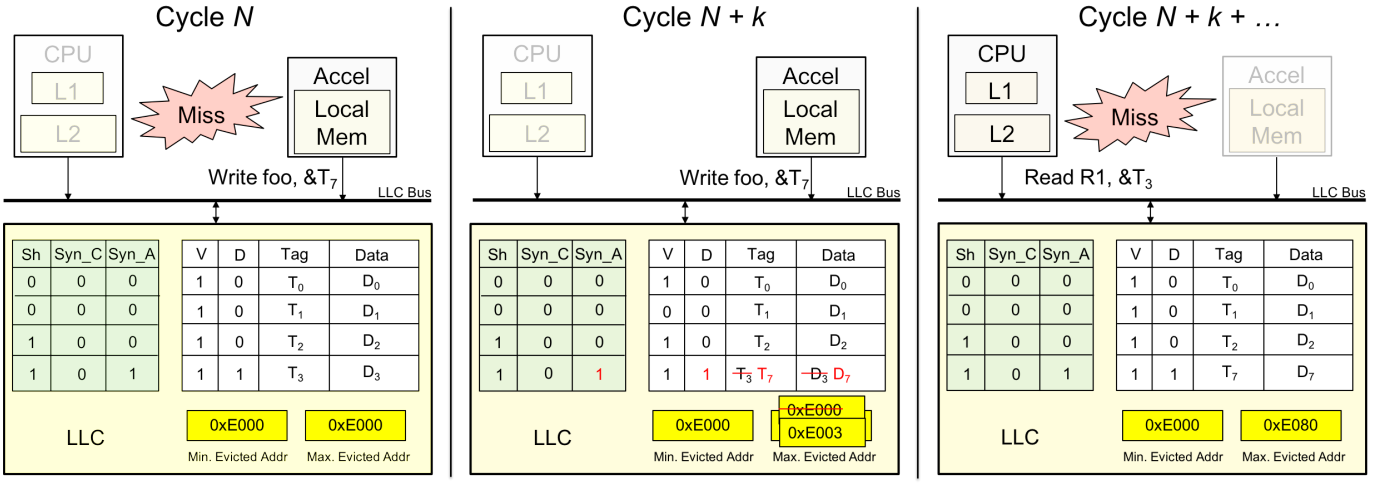

Fig. 10: CASPHAr synchronization on a cache miss and cache eviction.

first checks if the *Syn_C* bit is set to 1. If so, it means the cache line is ready to be consumed and the accelerator will be allowed to access the cache line and proceed accordingly much like a typical cache hit. Otherwise, the cache line is still being processed by the CPU and is not ready. In this case, the accelerator will experience a synchronization miss, and will be notified by the cache controller once the cache line becomes ready for consumption, i.e. when the *Syn_C* bit is set to 1. As with any other miss, the accelerator can issue other independent operations while waiting for that cache line to exploit memory-level parallelism. By contrast, *Syn_A* is the synchronization bit used when the accelerator produces data that later will be consumed by the CPU. *Syn_A* is set to 1 when the accelerator finishes producing data to be consumed by the CPU. The CPU will then similarly be synchronized and allowed to perform operations on that cache line as soon as the *Syn_A* bit is set to 1. *Sync_C* and *Syn_A* bits are reset whenever the accelerator or CPU, respectively, marks the data as consumed. On the CPU side, the *clflush* instruction thereby prioritizes resetting of *Syn_A* over setting of *Sync_C*.

Meta-data bits in the cache are initialized every time the configuration registers are set. On every region register (re-)configuration, the *Sh* bits are updated to reflect the tracking status of every line in the cache. Furthermore, they are updated every time a line is brought into the cache. By contrast, the *Syn* bits of all lines in the cache are reset to zero on configuration register updates (and otherwise tracked as described above). The synchronization status of evicted lines is tracked by storing their meta-data in off-chip memory on eviction and re-initializing *Syn* bits when the line is brought back into the cache. We assume full/empty (F/E) bits to be associated in the main memory with each shared region at cacheline granularity, either as part of extended memory arrays or in a separate memory region as in prior work [9], [10], [11].

In this paper, without loss in generality, we show a CASPHAr system for a one producer and one consumer scenario. CASPHAr inherently supports multiple producer-consumer scenarios including fork/join where each pair accesses unique addresses. Broadcasts can be supported by turning *Syn* bits into counters for a configured number of active consumers.

B. Synchronization Flow

In the following, we explain CASPHAr cacheline synchronization using a simple direct-mapped LLC as example.

Synchronization miss: Figure 9 shows the synchronization flow in CASPHAr for a synchronization miss when the requested line is in the cache but not yet ready. In this example, the accelerator is a consumer and the CPU is the producer of memory address T_2. When the accelerator attempts to read T_2, the *Syn_C* bit is 0, indicating that the data is not ready and

is still being processed by a consumer. A moment later, the CPU finishes processing the data and issues a software flush to memory address T_2. This will set the *Syn_C* and *D* bits to 1, marking the cache line ready and produced. The cache controller then wakes the accelerator up by signaling miss completion to let it retry a read for memory address T_2, in the same way that a cache miss wakes up a CPU instruction once the targeted cache line is ready. This time, the *Syn_C* bit is 1, i.e. the line is produced and ready. The accelerator then accesses the cache line as a hit and the process resumes. Note that deadlocks can occur if a line is never produced. CASPHAr is limited to regions with guaranteed producer-consumer relation.

Producer miss: Depending on the data access patterns and capacity conflicts, some cache lines might get evicted from the cache before they are produced. Figure 10 summarizes the synchronization flow in a case of pure cache miss for a case where the accelerator is the producer and the CPU the consumer. The accelerator attempts to write to a memory address T_7. However, cache line T_7 is not in the cache and consequently, the request is forwarded to the DRAM through the memory controller. Once the cache line T_7 is finally serviced, it is inserted into the cache and the synchronization process proceeds as explained above.

Consumer miss: In the scenario shown in Figure 10, line T_3 has to be evicted in order to make room for line T_7. However, T_3 itself is unconsumed. As in a conventional cache, such a line will be evicted and written back to memory. On a subsequent consumer access to such an evicted line, a miss will normally be serviced and the line will be brought back into the cache. However, if an evicted line has not yet been produced, it does not need to be brought back into the cache. It is sufficient to let the consumer experience a synchronization miss and be stalled until the producer updates the line.

C. Eviction and Replacement Optimizations

Eviction range registers: CASPHAr can be configured to additionally employ a set of eviction range registers to optimize unnecessary refetching of evicted lines. Two registers, *Min* and *Max*, can be employed to record the range of the smallest and largest address of produced but evicted cachelines. For every eviction of a produced cacheline that belongs to the shared memory region, *Min* and *Max* are updated when the evicted address is smaller or larger that the current *Min* or *Max*, respectively. When a consumer accesses the cache, CASPHAr will first check if the cache has the targeted line. On a miss, CASPHAr will consult the eviction range registers to check if the targetted cacheline has actually been produced but evicted. There exists two possible scenarios: 1) The cacheline's address is outside the range of *Min* and *Max*. This guarantees that the cacheline has not been previously produced. The cacheline is not fetched from memory and the consumer is stalled until the producer updates the line (and as a result, brings it back into the cache). 2) The targeted cacheline's address falls into the range of *Min* and *Max*. This in turn indicates two possibilities: the cacheline has been produced and evicted, or due to false positives in range tracking, the line has been evicted but not actually produced. In either case, the line will be brought from main memory to the LLC. CASPHAr will check the F/E information as a part of meta-data in main memory. If it finds the F/E bit is set, the consumer will subsequently be released. Otherwise, the consumer will be stalled and the synchronization will proceed normally.

In Figure 10, when the produced but unconsumed line T_3 is evicted, the *Max* eviction register is updated to match T_3's cacheline address $0xE003$. As more evictions occur, the *Max* eviction register is updated to $0xE080$. When the CPU later accesses T_3, it will experience a miss. CASPHAr sees the targeted cacheline's address $0xE003$ falling into the range defined by the *Min* and *Max* registers (i.e., the range of $0xE000$ - $0xE080$), and it assumes that the cache line can be accessed but has been evicted. Consequently, it will allow the access to proceed, issue the request to the memory controller and service the DRAM access. By contrast, if the targeted cacheline's address would be outside of the *Min* and *Max* range, it indicates that the evicted data requested by the consumer has not yet been produced. Thus, CASPHAr will not service the miss but instead continue and only wake up the corresponding consumer once the data is brought into the cache and marked by its producer.

As an alternative to eviction registers, evicted cachelines can be tracked by extending the LLC with an eviction bitvector, where each bit of the vector will mark whether a cache line that has been evicted has already been produced. This bitvector can be indexed relative to the start of the physical address of the shared region. A bitvector approach avoids any false positives and does not require backup F/E bits in main memory. However, it requires extra storage in the LLC with overhead that scales linearly with the size of the contiguous shared address space. By contrast, using two 64B eviction registers requires only constant LLC overhead independent of the shared memory region size. Eviction registers are optimized for the case of linear production patterns. Alternative structures like Bloom filters can be considered to provide different pattern-dependent false positive vs. overhead tradeoffs.

CASPHAr-aware cache replacement: CASPHAr enables fine-grain synchronization and data staging. However, if such capabilities are not managed carefully, they can skew the replacement policy's decision in efficiently managing cache space. A replacement policy can assign higher residence priority for some cache lines that have been consumed recently even though their lifetime in the cache has likely already expired. Similarly, it can unintentionally evict some cache lines that are ready and staged but not yet consumed. We describe how CASPHAr addresses these challenges by helping existing replacement policies make better decisions in the presence of fine-grain data-staging and synchronization.

CASPHAr allows any existing replacement policy to be extended to leverage fine-grain synchronization information. A replacement policy can use feedback from CASPHAr by prioritizing evictions for lines that have been consumed, i.e. whose lifetime has likely already ended. In doing so, CASPHAr can inherently and as quickly as possible make room for other staged lines that are ready for consumption in the LLC. CASPHAr achieves such a goal as follows: When the cache has to evict/replace a line to make room for a new one, it will

TABLE I: System configuration.

Parameter	Core Model
Issue Width	5
Cache Block Size	64 Bytes
L1-I$ Size	32kB, 8-way
L1-D$ Size	32kB, 8-way
L2 Size	256kB, 8-way
LLC Size	4MB, 16-way
LLC Inclusivity	Non-inclusive
LLC Repl. Policy	LRU, TRRIP [21], Hawkeye [35]
DRAM	
DDR Type	DDR3-1600
Specification	Micron MT41J256M4

first look for replacement candidates that have already been consumed. Consumed lines are identified using an extra bit that is set when a ready line is accessed. If such a consumed line exists, CASPHAr will evict that line. If there is no such cacheline, it will fall back to applying the conventional replacement policy. A further extension will thereby prioritize not-ready over ready lines for eviction, ensuring that ready lines remain resident as long as possible.

VI. Evaluation

In this section, we present our evaluation of CASPHAr across different applications and accelerator models.

Baseline: We use a modified version of MARSSx86 [13] as the cycle-accurate full-system simulator for architecture evaluation. This simulator has been calibrated to model the performance of a representative recent x86 CPU baseline architecture targeting HPC applications. We extend the baseline LLC model to implement CASPHAr. To show CASPHAr's compatibility with modern replacement policies, we integrate three replacement policies (LRU, TRRIP [21] and Hawkeye [35]) with CASPHAr. TRRIP extends RRIP [36] to be aware of different memory access behaviour among heterogeneous cores. Hawkeye was the winner of the most recent Cache Replacement Championship [37]. DRAMSim2 [38] is used to model DRAM delay and energy. Table I shows the detailed system configuration.

Overhead: Using Cacti [39], we estimate CASPHAr to add 0.5% area and 0.2%/0.5% dynamic power/static energy overhead for additional tag bits to the LLC while not affecting access or cycle times. Compared to tag and data arrays, which generally account for around 99% of cache area [40], the overhead for 128 eviction register bits and additional control logic for eviction register and synchronization bit setting and checking as part of existing hit/miss handling logic is assumed to be negligible. Overall, CASPHAr overheads are small, especially considering that LLCs typically contribute only up to 10% to energy at the full system level, which is dominated by CPUs and DRAMs [41], [8].

Comparison over prior art: We compare CASPHAr against a baseline acceleration (*Orig*) and prior art that employs a full/empty bit approach in main memory (*FE-DRAM*) [11]. Full/empty bits use meta-data to help with data exchange and synchronization. We compare a basic CASPHAr realization for cache-managed synchronization (*CASPHAr-Base*) to CASPHAr with eviction register and different replacement policy extensions (*CASPHAr-**). Base and *FE-DRAM* approaches use LRU policies by default.

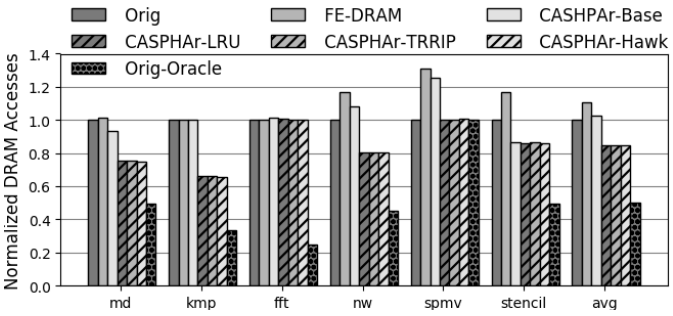

Fig. 11: DRAM accesses.

Accelerators and benchmarks: In addition to the FMM application introduced earlier, we use six accelerated applications from the MachSuite benchmark set [42] that cover a range of different producer-consumer patterns. Other than *spmv*, we included only benchmarks that were not already cache-contained. We also excluded benchmarks that required special dependencies not available on our simulator. All applications consist of data staging and kernel execution, where the CPU performs the data staging and invokes the accelerator to carry out the kernel execution. We build a cycle-accurate model for each accelerated kernel in the MachSuite applications using the Aladdin tool flow [43]. In all cases, accelerators are integrated into the full-system model at the shared on-chip LLC.

A. Performance and Energy Results

DRAM accesses: Figure 11 shows the normalized DRAM accesses across different MachSuite applications. We add a comparison to oracle execution with infinite LLC size (*Orig-Oracle*). Executions under an *FE-DRAM* approach generate an extra 10% of DRAM accesses overhead on average over original baseline acceleration, mainly due to not only the data exchanges but additional synchronization between CPU and accelerator having to occur in main memory. Such an overhead can be reduced with *CASPHAr-Base*, where a consumer by default inquires the LLC first for synchronization information and only accesses DRAM if the targeted cache line is not resident in the LLC. However, this approach still incurs 2% more DRAM accesses on average. By contrast, with the help of eviction range registers, unnecessary and expensive DRAM-level meta-data checking including associated cache pollution is avoided. *CASPHAr-** reduces DRAM accesses by up to 35% and 16% on average. False positives in range tracking are less than 0.01% for all benchmarks. In *spmv*, the problem size is small enough to already be cache-contained in baseline acceleration. Nevertheless, *FE-DRAM* and *CASPHAr-Base* still see an increase in DRAM accesses due to extra traffic generated by meta-data checking. In case of *fft*, execution with CASPHAr does not yield significant DRAM access savings despite significant saving opportunities existing as demonstrated by the oracle case. We observe that this is caused by the opposing order of shared data accesses made by the data staging and *fft* kernel execution in the application. As discussed in Section IV-B, when such a case occurs, producer-consumer transactions are restricted to a very minimal overlap. *Orig-Oracle* achieves the minimal DRAM accesses achievable among all optimizations. In *Orig-Oracle*, capacity conflicts are completely avoided, leaving only cold misses.

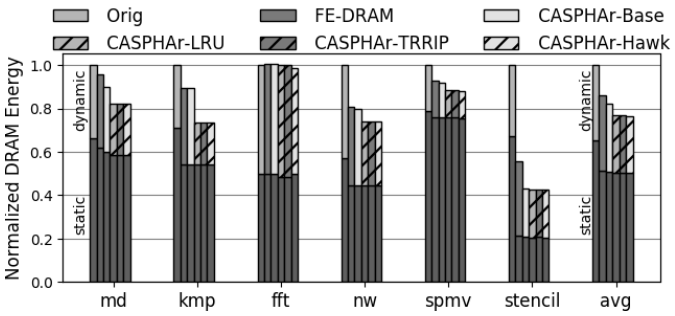

Fig. 12: DRAM energy.

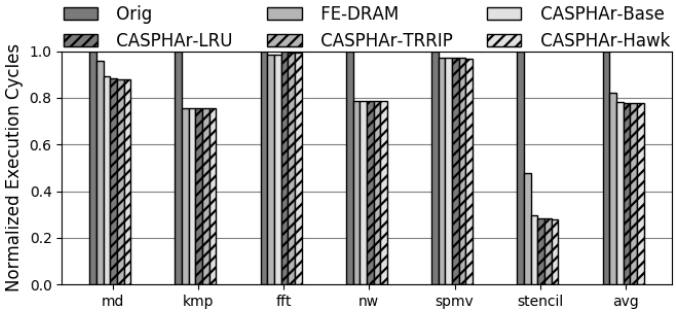

Fig. 13: Execution cycles.

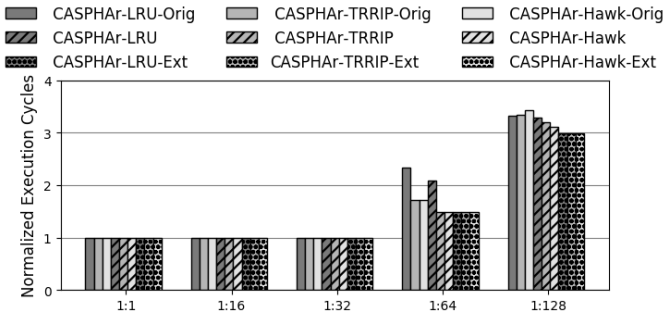

Fig. 14: Sensitivity to consumer slowdown rate (*stencil*).

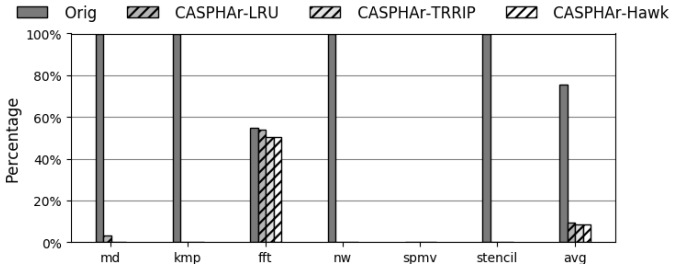

Fig. 15: Fraction of staged lines evicted before consumed.

DRAM energy: We further compare total (static + dynamic) DRAM energy consumption under original acceleration and CASPHAr in Figure 12. Energy savings of up to 65% and 22% on average can be observed with *CASPHAr-**. By contrast, *FE-DRAM* and *CASPHAr-Base* experience fewer savings averaging 16% and 18%, respectively, mainly driven by reductions in static energy due to shorter runtimes, where CPU energy savings will follow accordingly. CASPHAr achieves on average an additional 26% dynamic DRAM energy reduction over FE-DRAM due to fewer DRAM accesses. Among the applications, *fft* has limited energy savings. As discussed above, *fft* does not experience DRAM access reductions and this directly translates into minimal energy savings.

Performance: Figure 13 shows the normalized execution time across different MachSuite applications. Accelerations under *FE-DRAM* can improve performance by up to 52%, averaging an 18% performance boost. By contrast, CASPHAr reduces execution time by up to 71% and 23% on average using an extended *LRU* replacement policy, outperforming *FE-DRAM* by up to 40% and 6.7% on average. Despite a decrease in DRAM accesses, CASPHAr does not see performance gains for *kmp* and *nw* due to the latency-tolerant nature of these benchmarks. In *spmv*, even though the problem size already fits into the cache, CASPHAr still improves performance by 3% through fine-grain system-level pipelining and overlapping at cacheline granularity. By contrast, *fft* does not benefit from CASPHAr as the opposing order of data accesses in the applications also prevents pipelining and overlapping of dependent data.

Replacement policies: To further demonstrate the benefit of extended replacement policies, we study performance when varying the consumer access rate relative to the producer rate on a cache-sensitive *stencil* application. Figure 14 presents normalized runtime of CASPHAr using different replacement policies under varying relative consumer rates for the *stencil* application. We compare unmodified (*CASPHAr-*-Orig*) policies against default CASPHAr policies that prioritize eviction of consumed lines (*CASPHAr-**) and extended policies (*CASPHAr-*-Ext*) that further de-prioritize eviction of ready but unconsumed lines. We slow down the consumer access rate by decreasing its frequency. As the ratio of consumer to producer rate becomes smaller, CASPHAr experiences significant slowdown. A relatively lower consumer rate results in shared lines accumulating in the cache and at one point exceeding cache capacity. When such a case occurs, the replacement policy becomes critical in alleviating system-wide slowdown. For a medium slowdown of 1:64, *CASPHAr-** can reduce the performance degradation by to 10% for *LRU*, and 14% for *TRRIP* and *Hawkeye*. For a severe slowdown of 1:128 with more cache pressure, *CASPHAr-** can reduce performance degradation by 2%, 4.3% and 10% for *LRU*, *TRRIP* and *Hawkeye*. Further de-prioritization of ready lines (*CASPHAr-*-Ext*) can boost performance by an additional 8%, 5.7% and 3%, respectively. All in all, With extended policies and better eviction decisions, CASPHAr can moderate the performance penalty due to cache capacity conflicts as much as possible.

B. Why CASPHAr Works

In the following, we show additional details about CASPHAr and the fundamental reasons and limitations in improving performance and energy over a baseline acceleration.

DRAM spills: Figure 15 shows the normalized breakdown of evicted shared data across applications. We measure the fraction of data that was produced but evicted before consumed. As shown in Figure 15, the fraction of data that is not been consumed before it gets evicted, i.e. that has to be spilled to DRAM and eventually be brought back, is more than 50% and up to 100% in the baseline acceleration. In other words, coarse-grain data staging and synchronization experience significant data spills from LLC to DRAM. CASPHAr can avoid such unnecessary spills when coupled with any replacement policy. With CASPHAr, the cache inherently lets the consumer access

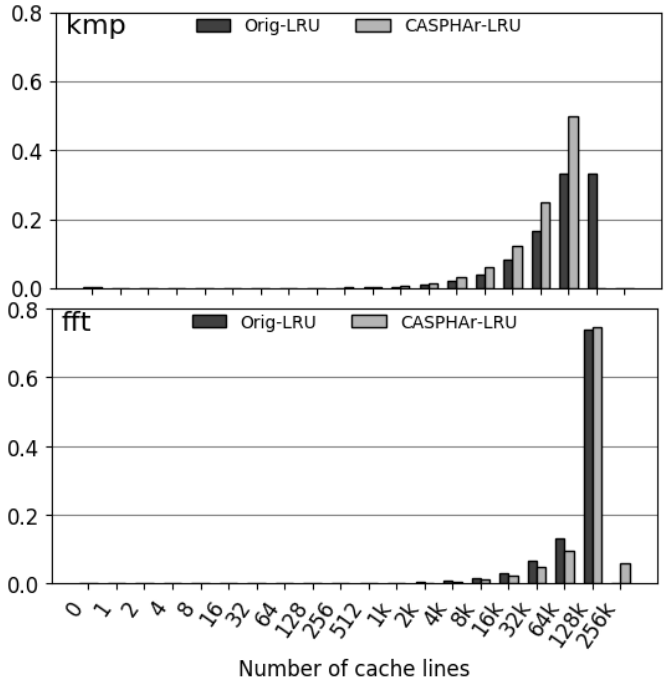

Fig. 16: Histograms of produced but unconsumed shared cache line occupancies.

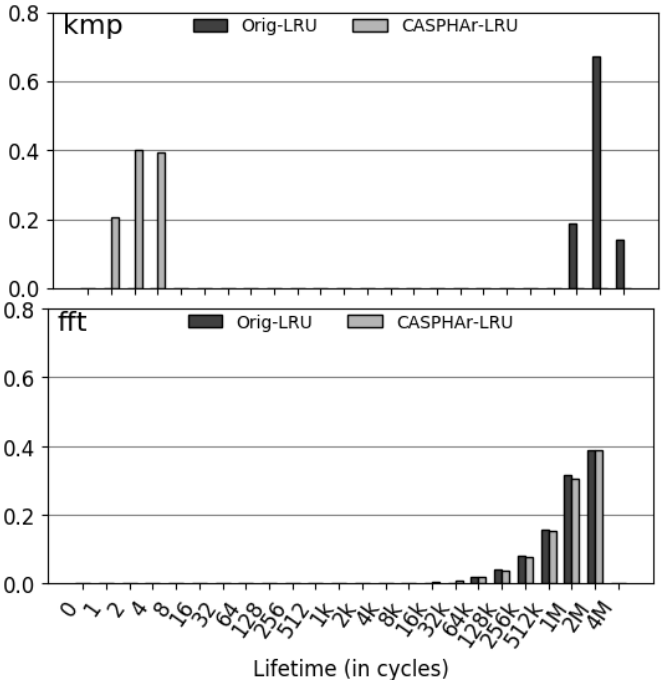

Fig. 17: Histograms of produced shared cache line lifetimes until consumed.

the staged data in a timely fashion before it gets evicted. CASPHAr does not show significant benefits in the case of *fft* due to opposing order that makes data spills unavoidable. In *spmv*, the problem size fits into the LLC capacity, i.e. only limited data spills are observed even in the baseline case. Note that while DRAM spills directly influence energy, depending on latency hiding in the application, savings in spills do not necessarily translate into performance benefits. At the same time, fine-grain pipelining and overlapping in CASPHAr can improve performance even when there are no DRAM gains.

Cache occupancy: CASPHAr avoids data spills by avoiding unnecessarily long occupancy of shared data in the cache through its fine-grain producer-consumer transactions. To demonstrate the actual benefit of such a technique, we profile the distribution of shared data occupancy in the LLC. We define shared data occupancy as the number of produced but unconsumed cache lines that are resident in the cache on every cache access. Figure 16 shows the shared data occupancy of *kmp* and *fft* applications. Here, *64K-128K* is the range where the number of produced cachelines waiting in the cache exceeds the LLC capacity. As can be seen from Figure 16, in the *kmp* case, more than 30% of the time, *Orig* execution falls into the range exceeding LLC capacity. As a result, capacity conflicts can not be avoided. By contrast, CASPHAr conceptually ”pushes” the distribution of shared data occupancy to the lower end of the range. Consequently, CASPHAr maintains a relatively smaller average and maximum shared data occupancy in the cache that does not cause capacity conflicts in the first place. By contrast, in the *fft* case, due to incompatible access order, CASPHAr does not significantly alter the occupancy distribution.

Shared data lifetime: We further demonstrate how CASPHAr unlocks inherent system-level pipelining and overlapping opportunities. We measure the distribution of the shared cache line lifetimes in the LLC. We define lifetime as the number of execution cycles from the moment a cache line is produced until it gets consumed. A shorter lifetime implies that fine-grain staging and synchronization are effectively achieved. Figure 17 illustrates that CASPHAr shortens the lifetime of cache lines by a significant margin over the baseline acceleration in the *kmp* application. CASPHAr effectively achieves a very short waiting time and delay between producer and consumer. This enables CASPHAr to free up cache space in a timely manner to make room for other cache lines. While CASPHAr is able to shift the lifetime distribution in *kmp*, in *fft*, average lifetime is reduced by less than 1%. Again, this is due to the nature of regular but opposing streaming order between data staging and kernel acceleration in *fft*. Both *kmp* and *md* have regular and matching producer and consumer patterns, where CASPHAr can reduce lifetime by 99% down to less than 500 cycles. Other benchmarks (*spmv, stencil, nw*) have regular producer but irregular consumer patterns with average lifetimes of 1,800-80,000 cycles and reductions of 30%-52% using CASPHAr.

Comparison to manual software optimizations: Finally, we discuss how CASPHAr compares against manually applied software blocking and pipelining optimizations. Figure 18 shows normalized execution cycles and DRAM accesses of CASPHAr compared to other software-centric approaches for the FMM and *spmv* applications. As mentioned before, FMM has a large data footprint while *spmv* has a relatively small footprint that natively fits into the cache. As can be seen from the figure, CASPHAr can unlock performance and off-chip access benefits similar to the manually tuned FMM. In case of the already cache-contained *spmv*, there is performance improvement from manual software optimizations. With a small data footprint and a coarse-grain accelerator, manual software optimizations are limited to only allowing accelerator execution

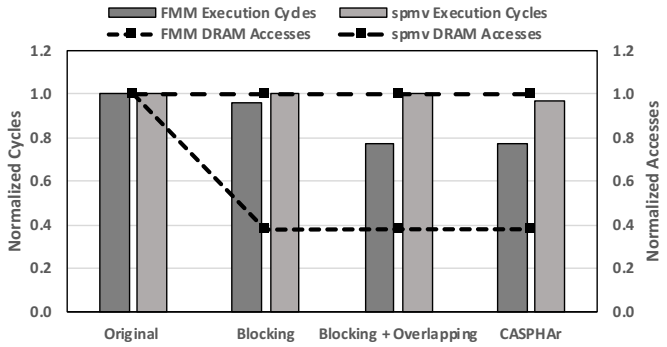

Fig. 18: Execution cycles and DRAM accesses of CASPHAr-LRU vs. manual optimizations (*FMM* and *spmv*).

once the CPU finishes preparing a minimum amount of data. A too fine-grained software blocking reduces accelerator efficiency and increase synchronization overhead. This forces the application to run sequentially even though a fraction of staged data may be ready earlier. CASPHAr, by contrast, is free from such restrictions. Since staged data synchronization is taken care of by the hardware, the accelerator can be allowed to start executing and to consume data as soon as it is ready without necessarily waiting for the host CPU to fully finish staging all data. As such, CASPHAr is able to improve *spmv* performance by 3%. Overall, CASPHAr can transparently achieve the same and in some cases better performance improvements than what is possible with complex manual software optimizations.

VII. Summary and Conclusions

In this paper, we presented CASPHAr, a novel last-level cache architecture that supports efficient accelerator staging and transparent fine-grain data movement orchestration by synchronizing producer and consumer accesses at cacheline granularity. CASPHAr tracks the synchronization status of individual cachelines across both resident and evicted cachelines, and it extends existing replacement policies to intelligently managing producer-consumer locality and data residency. In doing so, CASPHAr effectively reduces expensive spills and refetching to/from DRAM. Moreover, CASPHAr inherently realizes system-level pipelining between interdependent kernels that unlock additional performance gains. CASPHAr can achieve similar or better results than manual system-level data movement optimizations in software, improving performance by up to 23% and reducing energy consumption by 22% compared to baseline acceleration.

References

[1] J. Choi and R. W. Vuduc, "Analyzing the energy efficiency of the fast multipole method using a DVFS-aware energy model," in *IPDPSW*, 2016.

[2] N. Brookwood, "AMD Fusion Family of APUs: Enabling a Superior Immersive PC experience," 2012. AMD White Paper.

[3] J. Sermulins *et al.*, "Cache aware optimization of stream programs," *SIGPLAN Not.*, vol. 40, no. 7, 2005.

[4] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM TOMS*, vol. 35, no. 1, 2008.

[5] K. Fatahalian *et al.*, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," *SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2004.

[6] D. Lee *et al.*, "A distributed kernel summation framework for general-dimension machine learning," *Statistical Analysis and Data Mining*, vol. 7, no. 1, 2014.

[7] D. Malhotra and G. Biros, "A distributed-memory fast multipole method for volume potentials," *ACM TOMS*, vol. 43, no. 2, 2016.

[8] M. Asri *et al.*, "Hardware accelerator integration tradeoffs for high-performance computing: A case study of GEMM acceleration in N-body methods," *IEEE TPDS*, vol. 32, no. 8, p. 2035–2048, 2021.

[9] G. Alverson *et al.*, "Tera hardware software cooperation," in *SC*, 1997.

[10] A. Agarwal *et al.*, "The MIT Alewife machine: Architecture and performance," in *ISCA*, 1995.

[11] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *HPCA*, 2013.

[12] Y. S. Shao *et al.*, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *MICRO*, 2016.

[13] M. Asri *et al.*, "Simulator calibration for accelerator-rich architecture studies," in *SAMOS*, 2016.

[14] J. Dongarra and F. Sullivan, "The top 10 algorithms," *CiSE*, vol. 2, no. 1, pp. 22–79, 2000.

[15] J. Board and K. Schulten, "The fast multipole algorithm," *CiSE*, vol. 2, no. 1, pp. 76–79, 2000.

[16] A. Gray and A. Moore, "N-body Problems in Statistical Learning," in *NeurIPS*, 2000.

[17] J. Hestness, *Synchronization and Coordination in Heterogeneous Processors*. PhD thesis, University of Wisconsin, 2016.

[18] J. Alsop *et al.*, "Spandex: A flexible interface for efficient heterogeneous coherence," in *ISCA*, 2018.

[19] K. Bhardwaj *et al.*, "A comprehensive methodology to determine optimal coherence interfaces for many-accelerator SoCs," in *ISLPED*, 2020.

[20] J. Zuckerman *et al.*, "Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs," in *MICRO*, 2021.

[21] J. Lee and H. Kim, "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture," in *HPCA*, 2012.

[22] Z. Li *et al.*, "Set variation-aware shared LLC management for CPU-GPU heterogeneous architecture," in *DATE*, 2018.

[23] D. Giri *et al.*, "Accelerators and coherence: An SoC perspective," *IEEE Micro*, vol. 38, no. 6, 2018.

[24] E. G. Cota *et al.*, "An Analysis of Accelerator Coupling in Heterogeneous Architectures," in *DAC*, 2015.

[25] A. Boroumand *et al.*, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE CAL*, 2017.

[26] S. Xu *et al.*, "CuckooPIM: An efficient and less-blocking coherence mechanism for processing-in-memory systems," in *ASPDAC*, 2019.

[27] K. Hsieh *et al.*, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *ICCD*, 2016.

[28] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT*, 2015.

[29] S. Xu *et al.*, "PIMCH: Cooperative memory prefetching in processing-in-memory architecture," in *ASPDAC*, 2018.

[30] S. Haria *et al.*, "Devirtualizing Memory in Heterogeneous Systems," in *ASPLOS*, 2018.

[31] J. Gandhi *et al.*, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, 2016.

[32] L. Choi and P.-C. Yew, "Hardware and compiler-directed cache coherence in large-scale multiprocessors: Design considerations and performance study," *IEEE TPDS*, vol. 11, no. 4, pp. 375–394, 2000.

[33] A. Yarkhan *et al.*, "Porting the PLASMA numerical library to the OpenMP standard," *Int. J. Parallel Program.*, vol. 45, no. 3, 2017.

[34] D. E. Maydan *et al.*, "Array-data flow analysis and its use in array privatization," in *POPL*, 1993.

[35] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *ISCA*, 2016.

[36] A. Jaleel *et al.*, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ISCA*, 2010.

[37] A. Jain and C. Lin, "Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement," in *The 2nd Cache Replacement Championship*, 2017.

[38] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *IEEE CAL*, vol. 10, no. 1, 2011.

[39] R. Balasubramonian *et al.*, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," vol. 14, no. 2, 2017.

[40] A. Butko *et al.*, "Open2C: Open-source generator for exploration of coherent cache memory subsystems," in *MEMSYS*, 2018.

[41] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ASPLOS*, 2018.

[42] B. Reagen *et al.*, "MachSuite: Benchmarks for accelerator design and customized architectures," in *IISWC*, 2014.

[43] Y. S. Shao *et al.*, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA*, 2014.