Fusion : Design Tradeoffs in Coherent Cache Hierarchies for Accelerators *

Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula
School of Computing Sciences, Simon Fraser University
{ska124, ashriram, nvedula}@cs.sfu.ca

Abstract

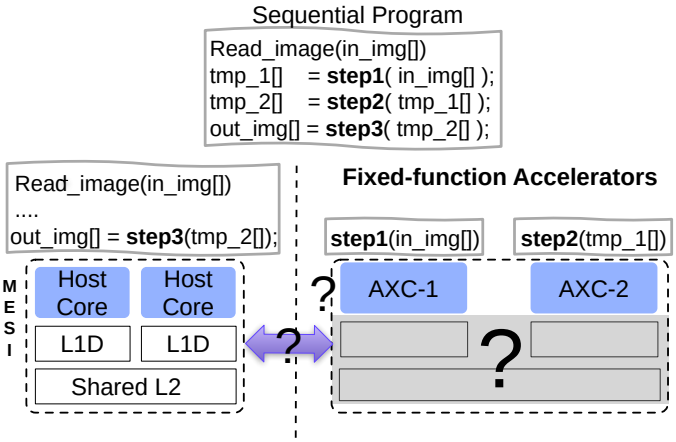
Chip designers have shown increasing interest in integrating specialized fixed-function coprocessors into multicore designs to improve energy efficiency. Recent work in academia [11, 37] and industry [16] has sought to enable more fine-grain offloading at the granularity of functions and loops. The sequential program now needs to migrate across the chip utilizing the appropriate accelerator for each program region. As the execution migrates, it has become increasingly challenging to retain the temporal and spatial locality of the original program as well as manage the data sharing.

We show that with the increasing energy cost of wires and caches relative to compute operations, it is imperative to optimize data movement to retain the energy benefits of accelerators. We develop FUSION, a lightweight coherent cache hierarchy for accelerators and study the tradeoffs compared to a scratchpad based architecture. We find that coherency, both between the accelerators and with the CPU, can help minimize data movement and save energy. FUSION leverages temporal coherence [32] to optimize data movement within the accelerator tile. The accelerator tile includes small per-accelerator L0 caches to minimize hit energy and a per-tile shared cache to improve localized-sharing between accelerators and minimize data exchanges with the host LLC. We find that overall FUSION improves performance by 4.3× compared to an oracle DMA that pushes data into the scratchpad. In workloads with inter-accelerator sharing we save up to 10× the dynamic energy of the cache hierarchy by minimizing the host-accelerator data ping-ponging.

1 Introduction

Current trends in microchip design indicate that to meet power budgets designers would have to power down an increasing fraction of the components on a chip [28]. Prior research in both industry and academia [24] have sought to address the power challenge through the use of hardware specialization (aka., accelerators) that target specific program regions. Prior work [11, 12, 36] has sought to extract accelerators from program regions such as functions and loops. We find that a key challenge to retaining the energy efficiency of accelerators is the data movement as the program moves around the chip seeking accelerators. This challenge is particularly relevant today as wire communication energy and cache access energy dominate compute energy in fixed function accelerators. Accelerators improve efficiency of the compute datapath to the level at which the overall energy consumption is primarily dominated by memory operations [12]. It is important to optimize data access energy to ensure that we retain the overall energy benefits from the use of accelerators. In this paper, our quantitative analysis focuses on fixed-function accelerators [12] to highlight the energy overhead of data migration between accelerators. However, we believe that the overall qualitative conclusions should extend to other accelerator types. Prior work has recognized the need to minimize short data movement between producer and consumer operations within an accelerator [26]; here, we focus on data sharing and data movement between accelerators.

We highlight the design challenges with an image processing application in Figure 1. The application reads an image and passes it through different step functions



Top: Image processing application that processes the input in 3 steps. An application representative of these steps would be "histogram" in our benchmark suite. Bottom: Multicore chip with fixed-function accelerators. AXC-1 and AXC-2 are accelerators collocated in a single tile. The ? indicate the design questions addressed in this paper: the cache hierarchy for accelerators, efficient data migration between accelerators, and the tradeoffs when transferring data to/from the host.

Figure 1: Offloading Sequential Program to Accelerators

*This work was supported in part by an NSERC Discovery Grant, NSERC SPG Grant, IBM Faculty Award, MARCO Gigascale Research Center, and Canadian Microelectronics Corporation.

ISCA'15, June 13-17, 2015, Portland, OR USA

DOI: http://dx.doi.org/10.1145/2749469.2750421

(`step1(), step2(), step3()`). In the contrived example, `step1()` and `step2()` are accelerated by the accelerators AXC-1 and AXC-2 respectively while `step3()` continues to run in software on the host processor. The key questions are i) how are data elements `in_img[]` and `out_img[]` transferred between the host and the accelerator and ii) how do the accelerators, AXC-1 and AXC-2, exchange the intermediate data (`tmp_1()`).

Industry vendors, spurred by the need to reduce the latency overhead of host-accelerator communication, have developed coherent direct memory access (DMA) engines [4, 10, 16] that transfer data directly from the host's LLC into the accelerator's explicitly managed local storage (scratchpad). While this approach is suitable for computationally intensive accelerators with few memory operations it is inefficient when fixed-function accelerators offload functions that share data with each other. The DMA-based approach requires multiple data transfers between the accelerator's scratchpads and the host, expending significant cache and interconnect energy. The DMA overhead is particularly notable compared to the "un-accelerated" system in which the inter-function data reuse would be captured by the cache hierarchy of the core running the sequential program. Also accelerators that are not compute dominated may experience performance overhead due to the DMA transfers on the critical path. Recent research [36, 41] has recognized the importance of collocating accelerators to minimize data transfer overhead. They integrate fixed-function accelerators at the host, and the L1 cache is shared between the host and the accelerators. Sharing the L1 cache helps minimize the overhead of data transfers between accelerators and enables participation in coherence. However, the shared L1 cache also introduces a challenge. We show that accelerators exhibit different memory level parallelism. To ensure that we retain the energy and performance benefits of the fixed function accelerators [12], the cache hierarchy needs to be optimized. We also show that the load-to-use latency and energy of the shared cache might minimize the benefits of fixed function accelerators. Additionally, the datapath of an in-core accelerator is limited by the bandwidth afforded by TLB, L1 cache, and register file ports of the core.

In this paper we focus on fine-grain offloading of multiple functions from a sequential program. We find that cache and coherence protocol optimizations are even more important today since with fixed-function accelerators the data movement constitutes the dominant overhead. We propose, *FUSION*, a multi-level coherent cache hierarchy for accelerators. *FUSION* adopts a split organization in which fixed-function accelerators are grouped in a physical "tile" and implement a localized lightweight memory hierarchy with private L0 caches per accelerator (L0X) and a shared L1 (L1X) cache per accelerator-tile. The private L0X caches data and act like a scratchpad to capture the locality within an offloaded function, and ensure low load-to-use latency and energy for the memory operations. The shared L1X captures the inter-function temporal and spatial locality between functions offloaded to the accelerators. Typically sequential programs tend to include multiple functions suitable for acceleration like the image processing example and a lightweight multi-level hierarchy is needed to capture this locality while minimizing energy for the access to the cached data elements.

Coherence is maintained locally within the accelerator tile between the L0Xs and L1X using time-stamp based coherence protocol [22, 31, 32]. Implementing lightweight coherence between the private L0Xs and the shared L1X eliminates the need for "DMA-ing" (programmed explicit copying) data between accelerator cores. *FUSION* removes the ping-pong effect of moving data out of an accelerator scratchpad into the host's coherence space (at the shared L2) and then into another accelerator scratchpad. Overall *FUSION* saves bandwidth between the L2 and the accelerator tile and hence energy in the on-chip interconnect. In this paper, we also propose *FUSION-Dx*, that further optimizes the inter-accelerator data sharing the accelerated functions similar to the example in Figure 1. In such cases *FUSION-Dx* leverages the accelerator tile's coherence protocol to proactively forward the dirty data from the producer accelerator's L0X to the consumer accelerator's L0X, thus saving write energy to the shared L1X and minimizing load-to-use latency for the shared data. Overall, we also discuss the energy benefits of the timestamp-based coherence which minimizes coherence messages, permits relocation of the TLBs to cache miss path, and enables proactive data movement optimizations.

FUSION is evaluated using a variety of applications drawn from the SD-VBS [35] and Machsuite [27] benchmark suites. *FUSION* eliminates the DMA transfers required to transfer data between accelerators and on average reduces energy by 2.5×. Like scratchpad based systems, *FUSION* exploits the temporal and spatial locality to minimize the accesses to the shared cache. Additionally, *FUSION-Dx* saves up to 17% of the dynamic link energy, for accelerators sharing data, by eliminating the shared L1X from the critical path.

The rest of the paper is organized as follows: Section 2 describes the baseline architecture and the challenges in designing a cache hierarchy for accelerator. We also characterize the behavior of functions offloaded to accelerators. Section 3.1 provides an overview of the *FUSION* architecture and highlights the benefits compared to the scratchpad approach and shared cache approach, and Section 3.2 provides a detailed description of the architecture and discusses the details of the protocol and cache layout. Section 4 details the evaluation toolchain and finally in Section 5 we present the quantitative tradeoffs between the different cache organizations.

Lessons Learned

- **Coherency mechanisms help optimize data movement.** We find that localized cache coherence within the accelerator tile help optimize data sharing between accelerators while minimizing interaction with the host's LLC to save energy.

- **DMA transfers increase energy overhead.** We have analyzed the traffic caused by repeated DMA transfers and find that significant energy is expended in applications which share data between functions.
- **Need to eliminate request messages.** Caches operate in a pull-based mode and potentially expend significant energy in the network links that may offset the gains obtained from eliminating host-accelerator DMA transfers for compute intensive accelerators.
- **Write forwarding can reduce cache energy.** We find that accesses to the shared L1X can expend significant energy in accelerators and proactive transfer of data between accelerator caches directly can save significant energy.

2 Background and Motivation

In this section, we characterize the two baseline architectures currently explored by recent accelerator studies [12, 41]: *SCRATCH* and *SHARED*. Later in the section we present the characteristics of accelerators derived from sequential programs. We using the image processing example shown in Figure 1 to highlight the overheads that may arise when running on *SCRATCH* and *SHARED* designs.

2.1 Baseline Architectures

Scratchpad per Accelerator (*SCRATCH*): Both ARM [10] and IBM [4] have enabled coherent access to the shared LLC from the accelerators. The coherence is limited to the DMA operations reading the most up-to-date data from the shared last level cache (LLC); writebacks from the scratchpad are managed using DMA. In Figure 2 the individual accelerators each include a scratchpad that connects with the shared L2 in the multicore through a coherent DMA controller. The scratchpad approach is well suited system which are either compute intensive (e.g., Cryptographic Unit [4]) or there is minimal interaction between accelerators (Cell SPE [34]). However, when functions or loops from sequential programs are offloaded to accelerators the shared data needs to be carefully managed as the program execution migrates between the different accelerators.

We illustrate the operation *SCRATCH* with the image processing example. To initialize the accelerator the host processor fills in the input data block at the shared L2 and DMAs the block to the accelerator (Steps ❶,❹). In the evaluation (see Section 5) we find that the proactive pushing of data into the scratchpad is one of the key benefits of DMA compared to a

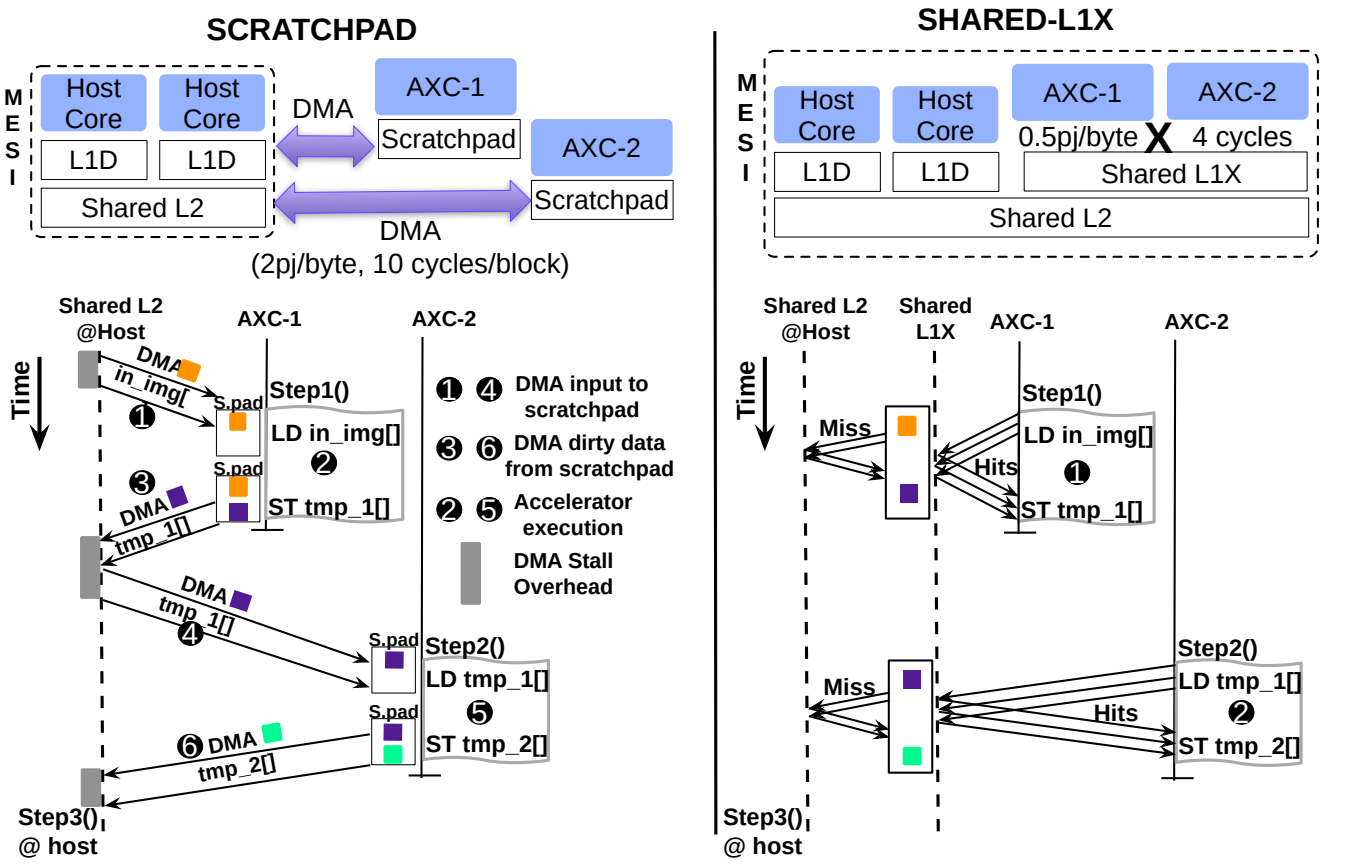


Figure 2: Left: *SCRATCH* Architecture. Per-accelerator scratchpads into which DMA transfers data. Switches to a different accelerator Right: *SHARED*. Shared L1 cache between the accelerators in a tile. The Shared L1 cache is kept coherent with the host multicore through MESI protocol. Host shared L2 maintains inclusion with the accelerators shared L1X.

cache hierarchy which operates in a pull-based mode. We find that coherence request messages expend significant energy in links. The scratchpad size per accelerator introduces a tradeoff. A large scratchpad reduces the number of DMA operations and amortizes the cost of DMA but increases the load-to-use latency and load-to-use energy during accelerator execution. With the data accesses dominating the energy consumption in fixed-function accelerators [30], scratchpads tend to be small requiring repeated DMA. Another challenge is the interaction between multiple accelerators. As shown in Figure 2, when the program switches from `step1()` to `step2()`, DMA is needed to transfer the temporary data (`tmp_1[]`) from AXC-1's scratchpad to the shared L2 and then into AXC-2's scratchpad (❸ and ❹). Our analysis of data access patterns reveals that functions drawn from the same program tend to have significant data sharing (see Table 1 below).

***SHARED* between Accelerators:** Current work in academia [11, 36] and industry [13] are exploring the benefits of "at-the-core" accelerators that share a host core's L1 cache which enables the accelerators to efficiently transfer data between the host and the accelerators and between the accelerators. Unfortunately, the shared L1 needs to be sized to accommodate both software threads running on the host processor and the various accelerators. Since fixed-function accelerators expend minimal energy on the operation itself the load-to-use latency and energy of the shared L1 cache constitutes a significant overhead [30] in today's wire limited era [7]. In this paper, we explore the benefit of multi-level caches for accelerators and demonstrate how to efficiently maintain coherence in the hierarchy.

Figure 2 : SHARED-L1X illustrates the *SHARED* architecture; we only show a single tile of accelerators. In *SHARED*, a tile of accelerators is connected to a multibanked L1 shared cache (Shared L1X) through a common switch. The accelerators incrementally load data into the shared cache as they run. As shown in steps ❶ and ❷, the accelerators are activated on a context switch once the basic register state is transferred. All accesses from the accelerators are issued to the shared L1X cache which appears as just another L1 agent to the coherence protocol and participates in the MESI operations. The host's shared L2 maintains inclusion with the L1X. Compared to *SCRATCH* all accesses from the accelerators are issued to the L1X which has a higher load-to-use latency and energy. Our *SHARED* architecture is similar to both Dyser [11, 29] in that accelerators share a common L1 cache that participates in coherence operations; in our model the host processor resides on a separate tile. Since caches tend to dominate the overall energy consumption in fixed function accelerators, earlier work [25] has recognized the need for customizing the cache size and organization for individual accelerators [12, 30]. However, there is a tradeoff between optimizing the shared L1X for data sharing between accelerators while supporting low load-to-use latency and energy per accelerator. Prior work [41] has managed the coherence between the host cache and the scratchpad using explicit instructions.

Table 1: Accelerator Characteristics

Function	% Time	%INT	%FP	%LD	%ST	MLP	%SHR
FFT							
step1	27.3	28	7.8	46.3	17.9	4.8	56.3
step2	8.8	52.1	0	29.9	18	4.0	99.5
step3	28.7	31.6	7.5	43.2	17.7	4.4	61.5
step4	8.5	49	0	31.8	19.2	2.9	50.8
step5	8.4	49	0	31.8	19.2	2.9	50.8
step6	18.4	20.3	3.3	53.8	22.6	4.3	19.4
Disparity							
padarray4	7.7	71	0	15.2	13.8	5.0	50
SAD	7.7	57.9	8.2	17.6	16.3	3.0	33.3
2D2D	15.4	62.8	0	24.9	12.3	3.5	49.7
finalSAD	30.8	22.8	0	71.3	5.9	5.7	47.9
findDisp.	23.1	32.7	32.3	30.7	4.3	2.2	31.4
Tracking							
imgBlur	14.3	52.8	15.1	24	8.1	2.0	58
imgResize	14.3	57.1	11.4	26.3	5.2	1.3	99.9
calcSobel	28.6	52.8	17.4	22.8	7.1	1.0	32.5
ADPCM							
coder	50	32.8	0.0	56.0	11.2	1.6	99.0
decoder	50	40.8	0.0	48.0	11.2	1.7	98.9
Susan							
bright	1.0	22.5	48.9	20.3	8.4	2.2	59.4
smooth	66.2	24.3	0.0	67.6	8.1	2.0	36.2
corn.	13.2	33.1	1.3	61.0	4.6	2.1	7.6
edges	20.6	32.6	1.6	60.3	5.5	1.9	12.3
Filter							
medfilt	74.4	48.2	0.0	49.1	2.7	1.6	14.2
edgefilt	25.5	41.3	23.9	28.1	6.7	4.1	23.1
Histogram							
rgb2hsl	48.2	22.1	51.8	20.7	5.4	3.5	8.3
histogram	3.6	40	0	53.3	6.7	1.0	100
equaliz.	3.6	36	0.1	59.9	4	1.0	66.2
hsl2rgb	15.7	26.3	40.8	22.1	10.8	3.1	75.0

See Section 4 for detailed description of our toolchain

Fixed-Function accelerators from Sequential Applications: Table 1 lists the characteristics of the specific functions we accelerated from our benchmarks drawn from Machsuite [27] and SD-VBS [35]. We focus on the workloads in which multiple functions can be offloaded to accelerators and share data between the accelerated functions and the host processor. Please refer to Section 4 for a detailed description of our toolchain. The term "accelerator" is used in different contexts so we have listed the function names and their features to clarify to the reader the granularity; in this paper we extract accelerators from loops and functions of sequential programs.

To choose the appropriate accelerators for this study we profiled the sequential programs on a Intel Core i5 processor. Table 1:%TIME lists the fraction of total time spent in the individual functions. In many of our applications (other than histogram) the critical path includes multiple functions and the functions are invoked repeatedly (possibly from different sites in the program). The breakdown of the operations within each accelerated function (%INT, %FP, %LD, %ST) is also presented in Table 1. The operation breakdown is obtained from the fixed-function hardware extracted from the dataflow graph

of the accelerator (see Section 4). Given the dominance of interconnect energy (≃1pJ/mm/byte [7]) and cache energy [12] compared to operation energy (0.5pJ/integer add [2]) in fixed-function accelerators it is imperative to capture the spatial and temporal locality and thus reduce the energy for load and store operation. The sharing degree (%SHR) characterizes inter-accelerator communication. We define the sharing degree (%SHR) to be the fraction of cache blocks accessed by the accelerator that are also accessed by at least another accelerator. The %SHR here is reflective of the temporal locality in the original application between functions and manifests itself as data transfers between individual accelerators. In our applications, apart from the initialization functions (e.g., `rgb2hsl` in Histogram) in our accelerated functions the average %SHR is ≃ 50%. Even in cases where only a small fraction of application time is spent (e.g., `paddaray4` in Disparity, `Equaliz.` in Histogram) the %SHR degree can be significant (50%+). In such cases, if we do not ensure low overhead data transfers into the functions, the overheads could potentially dominate the overall energy consumption of the accelerator. The final columns, LT, presents the Lease Time (used in the ACC protocol, see Section 3.1) assigned to each cache block in the L0X per function per benchmark.

Summary

- When multiple functions from a sequential program are offloaded to accelerators the data sharing needs to be carefully managed.
- Fixed-function accelerators access both private and shared data and we need a multi-level cache hierarchy to effectively capture the locality of the accelerators and reduce data access energy.
- To effectively manage sharing between accelerators and maintain data across a multi-level cache hierarchy scattered across accelerators we need to employ low-overhead cache coherence that minimizes control messages.

3 *FUSION* : A Coherent Accelerator Cache Hierarchy

3.1 Design Overview

FUSION is a multi-level coherent cache hierarchy for the accelerator tile that supports i) low load-to-use latency and low load-to-use energy for each fixed-function accelerator, ii) low overhead data sharing and data migration between accelerators within the tile, and iii) efficient data migration between the host and the accelerator. *FUSION* collocates fixed-function accelerators in a separate tile and implements a separate coherence protocol and cache hierarchy within the accelerator tile. The system can support multiple accelerator tiles, though in Figure 3 only a single accelerator tile is shown.

Figure 3 depicts the architectural details and we focus on the overall hierarchy herein to illustrate the tradeoffs between *FUSION* and existing designs, *SCRATCH* and *SHARED*. Private L0X caches, which can be sized independently, are provisioned for each accelerator within a tile to optimize load-to-use latency and energy [25]. All the private L0Xs are connected to a banked, shared L1X cache. The L0X supports write caching (unlike private caches in GPUs) to minimize write bandwidth and link energy. *FUSION* maintains coherence between the private L0X and shared L1X using the ACC (ACcelerator Coherence) protocol (details of ACC are presented in Section 3.2). The shared L1X is the ordering point and is a participant in the CPU's MESI protocol actions.

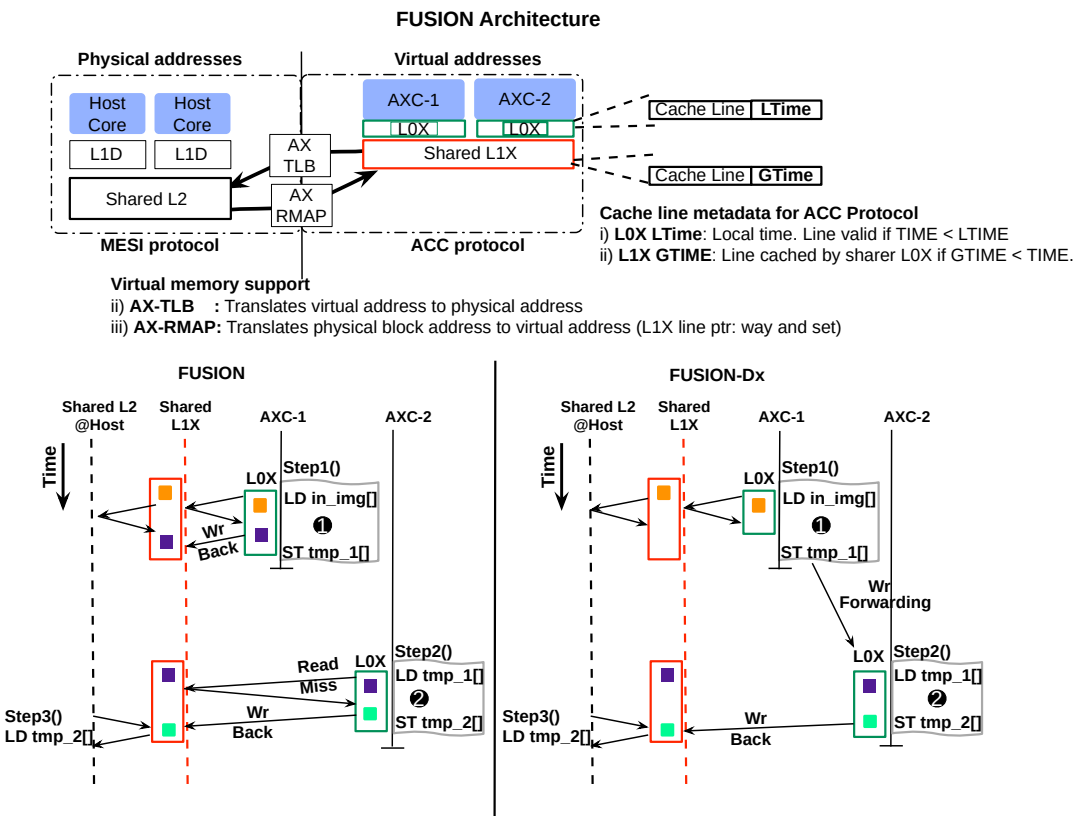
As shown in the timing diagram (Figure 3), like *SCRATCH*, *FUSION* caches the private data for the functions `step1()` and `step2()` in the L0X for energy efficiency. Like *SHARED*, *FUSION* enables the shared data (`tmp_1[]`) produced by `step1()` to be communicated efficiently to `step2()` without requiring intervention of the host processor (unlike *SCRATCH* which uses DMA). Finally, AXC-2 writes back the `tmp_2[]` which the host processor incrementally fetches as it runs `step3()`. Thus *FUSION* eliminates DMA transfers from the critical path, allowing fixed-function accelerators to incrementally fetch data when needed. More importantly, the extraneous DMA operations *between* accelerators are also eliminated.

FUSION-Dx further optimizes the data migration between accelerators to save energy. Note that while *FUSION* eliminates the DMA required when execution switches to a different accelerator, (e.g. `step2()`) it requires subsequent read misses to transfer data from the shared L1X to the consumer L0X. We find that the control messages for requests fetching intermediate data (e.g. `tmp_1[]`) expends significant energy (see Section 5.2 : Lesson 4) compared to *SCRATCH*. *FUSION-Dx* (see Figure 3, bottom right) optimizes producer-consumer sharing found between the accelerators by proactively pushing the data from the producer's (AXC-1) cache into the consumer's (AXC-2) cache and eliminates cold misses. Overall, *FUSION-Dx* eliminates the writeback from AXC-1's L0X to the L1X, AXC-2's read miss and the L1X access.

3.2 *FUSION* Architecture

In this section, we describe the architecture of *FUSION* and illustrate how individual accelerator and host memory operations, as well as their interaction, are handled. As depicted in Figure 3 the architecture is segmented into separate host and accelerator tiles. Only the operations within a single accelerator tile are described for brevity. The accelerator tile operates on virtual addresses and maintains coherence between L0Xs and the shared L1X using a time-stamp based coherence [22, 31, 32] protocol.

Virtual Memory: In *FUSION*, the accelerators operate with virtual addresses while the host processor operates with physical addresses. This design eliminates TLB's from the critical path of accelerator memory operations and minimizes the energy consumption per memory access. Figure 3 (top) shows the point of virtual-to-physical translation in *FUSION*; we address the issue of synonyms in the Appendix. Process id (PID) tags are added to the L0Xs and L1Xs to ensure that



FUSION: Baseline *FUSION* system that uses per-accelerator private caches (L0X) and a shared L1X to implement a multi-level hierarchy for accelerators. *FUSION-Dx*: Optimizes the accelerator coherence protocol (ACC) for direct Wr-forwarding between L0X of accelerators.

Figure 3: Top:*FUSION* Architecture. Bottom: Timeline for image processing example on *FUSION* and *FUSION-Dx*.

accelerators executing functions from different processes can co-exist on the same tile. The private L0Xs and shared L1Xs are indexed using virtual addresses on AXC memory operations. Since ACC is a self-invalidation protocol there are no internally forwarded coherence requests that need to look up the L0X caches. We add a TLB (see AX-TLB in Figure 3, top) on the miss path of the shared L1X when transiting from the accelerator tile to the host tile; the translation is needed to index into the shared L2 and participate in MESI actions.

Since the host uses physical addresses, a reverse translation (physical to virtual address) is needed to handle forwarded requests from the shared L2 to the L1X. A naive solution is to include the virtual address for the memory access in the coherence control message. Unfortunately, this doubles the size of the control message for all host memory requests since it is not known beforehand which ones may need to be forwarded to the accelerator tile. In the current wire energy dominated era, this is not an energy efficient solution. Instead, we chose to expend area and dedicate a separate accelerator reverse map (AX-RMAP) per accelerator tile. The AX-RMAP maintains the physical address of the lines in the shared L1X; it is indexed using physical block address and stores a pointer to the shared L1X line. Since the shared L2's directory acts as a filter (sharer list indicates if an accelerator tile has cached the line), only few requests are forwarded to the accelerator and require and AX-RMAP look up. Section 5.6 evaluates the proposed address translation scheme for the accelerator tiles.

Accelerator Coherence (ACC) Protocol: ACC is a time-stamp based self-invalidation protocol similar to protocols proposed for multiprocessors and GPUs [22, 31, 32]. ACC adds two important write optimizations for accelerators compared to the earlier approaches: Write caching and Write forwarding. We describe the baseline writeback caching here and implement write forwarding as part of the *FUSION-Dx* system described below. ACC is a self-invalidation protocol and is a strict 2-hop protocol that requires no extra coherence messages over the baseline scratchpad systems. The reduced need for coherence makes it attractive for the purpose of accelerators from an energy perspective. Note that the primary purpose of ACC is to *enable data migration* between accelerators without host intervention (DMA) as the program context migrates and *not concurrent sharing* between different accelerators. The ACC protocol supports sequential consistency semantics for accelerator execution.

Figure 3, *FUSION* Architecture (top), shows the compo-

nents of ACC. Only the accelerator cores within a single tile need to have a synchronized time-stamp register since ACC implements coherence only within a tile. A small time-stamp field (32 bits) is added to each cache line in the private L0X and shared L1X caches, as shown in Figure 3 (top). The local timestamp value (LTIME) in the L1 cache line indicates the lease time, essentially the time until which the particular cache line is valid. An L0X cache line with a local time-stamp *less* than the current system time is invalid. The global time-stamp (GTIME) in the L1X indicates a time by when all L0X caches will have self-invalidated the particular cache line.

Figure 4 (left) shows how ACC handles accelerator load and store misses. When AXC-1 issues a load request to the shared L1X, it requests a read-only epoch for the address (A) ending at time T=10 – ❶. The shared L1X receives AXC-1's load request (including the epoch request), records it and forwards it to the host-side along with a pointer to the L1X location (way and set). When transiting into the host tile, the AXC's load request is translated to a physical address. When the request crosses over to the host tile, it appears as a MESI load request from an L1 to the host's L2.

We have implemented a directory based 3-hop MESI protocol that takes the requisite actions to supply the requested data to the L1X. The data response includes a pointer to the L1X location so that on transitioning into the accelerator tile (which uses virtual addresses), the data response can update the he appropriate L1X entry. The shared L1X then replies to AXC-1 with the data and time-stamp of T=10 – ❷. The time-stamp indicates to AXC-1 that it cannot use this location beyond time T=10 – ❸. Subsequently, AXC-1 requests a write-epoch that expires at T=15 – ❹. To satisfy this write request the L1X implicitly locks the line and updates the L1X time-stamp to T=15. Subsequent readers or writers detect the locked line and simply stall at the L1X until the write lease expires and writeback completes. AXC-1 triggers a self downgrades and issues a writeback to the L1X – ❺ at T=15. When AXC-2 issues a read request – ❻ – the L1X finds the global time-stamp (T=20) to be less than the current time (T=25) and checks if the writeback has completed and waits if necessary. Once the writeback is complete, the L0X responds with a read lease.

A key implementation decision is how to implement self downgrade. This requires checking for dirty lines in the cache. We implement the downgrade checks without sweeping the entire cache by using the time-stamps as a filter. Each L0X cache set includes a writeback time-stamp (the earliest write lease in the set); each accelerator includes a writeback time-stamp for the entire L0X cache (the shortest lease time-stamp amongst all the sets). These timestamps are used to filter the checks for the dirty lines. The writeback timestamps are updated whenever the dirty bits in the cache are updated.

Overall, accelerators can acquire both read and write epoch's on the cache line and the shared L1X distinguishes such cases; subsequent accesses stall on write epochs, while reads are permitted in conjunction with other read epochs. The epoch requests are fixed based on the expected latency of the accelerator invocation since it experiences minimal non-determinism (only due to memory hierarchy).

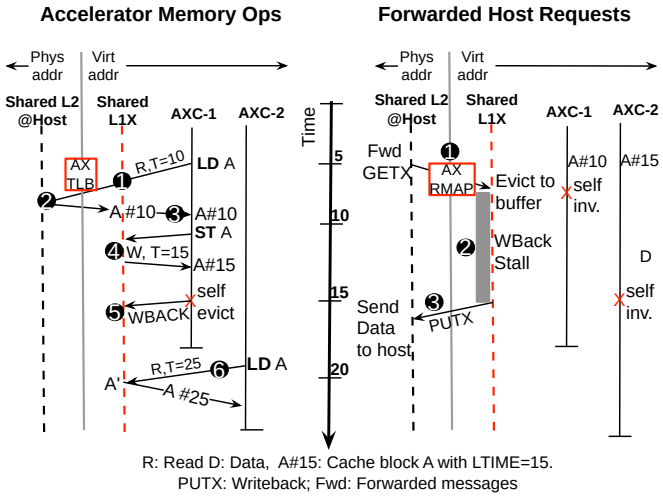


Figure 4: Left: ACC protocol servicing requests from accelerator and interaction with MESI. Right: ACC Protocol servicing forwarded requests from MESI.

Integrating ACC with MESI: Herein we describe how ACC participates in MESI operations. Exclusivity is maintained between the host processor tile and accelerator tile. The shared L1X always caches a block in exclusive state irrespective of the accelerator operation (read or write). When participating in the MESI protocol the shared L1X states map to a 3-state MEI protocol (M: modified, E: exclusive and clean, I:invalid). The ACC protocol responds in the same manner to all forwarded host coherence requests; i.e. relinquish ownership when the GTIME time-stamp expires and send an eviction notice to the shared L2. The shared L2 has perfect information on whether the accelerator tile is caching the block and eliminates any extraneous coherence messages that a cache would need to deal with as a result of silent drops from S→I. When a block is in I state in the accelerator tile, the L1X will not receive any messages from the shared L2 for that block.

Figure 4 (right) illustrates a forwarded request from the host processor. The host processor performs a store operation which results in a forwarded request to the accelerator tile. The forwarded request translates the physical address to the shared L1X pointer via the Accelerator Reverse Map (AX-RMAP) – ❶. A key benefit of the time-stamp based approach of the ACC protocol is that shared L1X will filter out the MESI *Fwd* messages and not forward them to the L0X. In the illustration, the forwarded request is received at time T=5 while the block is cached in the private L0X until time T=15. The *Fwd* message triggers an eviction from the shared L1X to a writeback buffer – ❷ – but the response eviction notice is stalled until time T=15. At T=15, a PUTX (eviction notice) is sent back to the shared L2 – ❸. The shared L1X uses the GTIME (see Figure 3,

top) to ascertain when it is safe to respond to MESI protocol action and does so without involving the private L0Xs. In our workloads, we observed between up to ≃800 forwarded requests from the CPU to the accelerator tile for the whole workload. (*TRACK*:817, *ADPCM,DISP.*:≃500, others < 50).

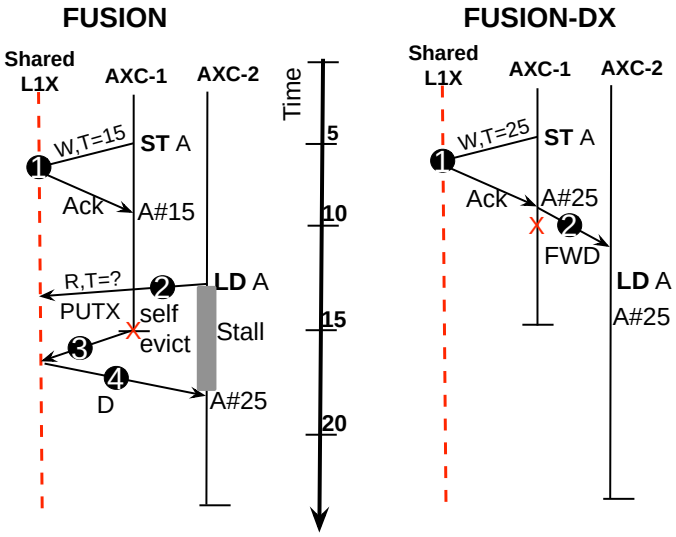


Figure 5: Left: *FUSION* without write forwarding. Right: *FUSION-Dx*. ACC protocol with write forwarding.

***FUSION-Dx* – Extending ACC to support Forwarding:** A key overhead present in the cache based coherence model (pull-based) versus the scratchpad based DMA model (push based) is the extraneous control messages issued per cache miss. While the control message overhead is minimal in a multicore [31] or GPGPU [32] context, for fixed-function accelerators, they add notable overhead to the overall energy consumption. One particular source of inefficiency is the store-load forwarding shown in Figure 5:*FUSION*. AXC-1 writes to A – ❶ – but does not complete processing until later – ❸. In the meantime, AXC-2 wishes to read the data but has to stall until AXC-1 self-evicts the line. There are two inefficiencies in the contrived example: i) the coherence messages (write back, read request and data response) needed over the L0X-L1X link wire and ii) the stalled read on AXC-2. With accelerators exploiting all available operation parallelism and reducing compute energy consumption, a significant challenge in our workloads is the energy cost of the coherence messages. *FUSION-Dx* optimizes by providing a mechanism to directly forward data from AXC-1 to AXC-2. For MESI protocols such "proactive" forwarding requires significant complexity and involvement of the coherence directory [17]. With the ACC protocol, forwarding simply involves self-eviction and forwarding the data with the already requested lease lifetime (see Figure 5:*FUSION-Dx*). Forwarding without informing the shared L1X is feasible with ACC since the L1X only tracks the lease epoch and is not concerned with the owner of the lease. The only challenge is to identify the stores that may benefit for forwarding and the producer-consumer cores. In this work, where the simulation infrastructure is trace driven (see Section 4), we post process the trace to identify the stores to be forwarded from the producer to the consumer accelerator.

4 Toolchain and Benchmarks

We have developed a detailed cycle accurate simulator that models the host cores, fixed-function accelerators and memory system faithfully. The host OOO core pipeline is modelled in detail using macsim [1] and the memory hierarchy using GEMS [21]. Table 1 characterizes the accelerators that we extracted; we assume that all accelerators derived from an application are collocated on the same accelerator tile.

Modelling accelerator cores: To identify and model the fixed-function datapath of the accelerator we adopt a technique similar to Aladdin [12]. The applications are profiled using `gprof` which identifies the critical functions and the function call hierarchy. Based on the `gprof` profile, we identify top level functions for acceleration and ensure that accelerated functions are free of external library calls such as `malloc`. A dynamic trace of these functions is used to generate a constrained dynamic data dependence graph that includes program order constraints (control and memory dependencies). To model the fixed function accelerator we traverse the activity of the constrained data dependence graph on a cycle-by-cycle, generating any requisite memory operations in a cycle and stalling the appropriate operations as necessary based on the availability of allocated resources. We assume an aggressive non-blocking interface to memory.

Table 2: System parameters

Host Core	2 GHz, 4-way OOO, 96 entry ROB, 6 ALU, 2 FPU, INT RF (64 entries), FP RF (64 entries) 32 entry load queue, 32 entry store queue
L1	64K 4-way D-Cache, 3 cycles
LLC	4M shared 16 way, 8 tile NUCA, ring, avg. 20 cycles. Directory MESI coherence
Main Memory	4ch,open-page, 16GB 32 entry cmd queue, 200 cycle latency
Accelerator Cache Hierarchy	
Scratchpad Shared-L1X Private L0X	4 or 8KB RAM (ITRS HP) 64KB or 256KB, 16 banks. 8 way. (ITRS, HP) 4 or 8KB Cache (ITRS HP)
# of AXCs	2 (FILT) — 6 (FFT)
Link Energy Parameters	
	Accelerator-L1X (0.4pJ/byte), L1X-Host L2 (6pJ/byte)

Systems compared: We evaluate the following systems:
i) Oracle-*SCRATCH*: For the *SCRATCH* system we model individual scratchpads per-accelerator and generate DMA code to move data into and out of the accelerator. We assume a particularly aggressive oracle DMA implementation, and auto generate the DMA operations based on the memory accesses we observe in the dynamic trace of the application. We *only* DMA into the accelerator scratchpad read data and DMA out dirty data. All the benchmarks have working set sizes larger

Table 3: Accelerator Execution Metrics

Function	KCyc.	LT	%En.	Function	KCyc.	LT	%En.
FFT (Cache/Compute Energy = 0.8)							
step1	25.3	500	34	step4	9.9	700	6
step2	7.1	700	4	step5	9.9	700	6
step3	23.5	200	35	step6	17.8	500	15
Disparity (1.6)							
padarray4	11.2	500	5	finalSAD	25.9	500	23
SAD	27.7	500	25	finalDisp	71.4	500	33
2D2D	34.2	500	14				
Tracking (0.5)							
imgBlur	9587	700	48	calcSobel	7358	720	34
imgResize	3837	770	18				
Histogram (2.7)							
rgb2hsl	38007	500	47	equaliz.	3250	500	1
histogram	3244	500	2	hsl2rgb	69671	500	50
ADPCM (9.7)							
coder	3453	1400	55	decoder	3364	1400	45
Filter (4.9)							
medfilt	48403	400	49	edgefilt	5663	400	51
Susan (3.1)							
bright	18.6	1000	1	corn	6328	1200	5
smooth	61496	1700	86	edges	18858	1700	8

KCyc.: Execution time (K Cycles), LT: lease time assigned to blocks, %En. : % of total accelerator energy. Cache / Core energy ratio shown in brackets beside benchmark name.

than the scratchpad (4096 bytes) and thus are segmented into "windows" of execution with DMA operations required for each window. We faithfully model the complete state machine of the DMA controller and assume that it resides at the host's LLC i.e., no overhead for issuing DMA requests.

ii) *SHARED*: This system models a single shared L1 cache for all the accelerators in a tile. We collocate the functions from the same application in a tile and ensure that there is no inter-tile communication between offloaded functions from the same application.

iii) Finally, *FUSION* and *FUSION-Dx* include the full cache hierarchy with private L0Xs and shared L1X. GEMS was modified to add support for interfacing with the fixed-function accelerator cores and to model the ACC and MESI protocols along with their interaction in detail.

Energy Model: To model the energy of fixed function accelerators we use an activity count based power model from Aladdin [12]. We assume a 45nm ITRS HP technology. Cache energy is modeled using CACTI [23]; Table 2 lists the transistor types we assume for each cache. The L0X tag accesses include a 32 bit time stamp field check which is accounted for as an 15% energy overhead. In the benchmarks studied, we find that provisioning for 24 bits accounts for 98% of accelerator invocations; (*HIST.*, *FILT.* and *SUSAN* have functions which run longer) and using 3 additional bits accounts for all invocations. We estimate link energy based on published figures [7] (1pj/mm/byte) and calculate wire length based on the area of the components where $Wire\ Length = 2 \times \Sigma_{i=1}^{n} \sqrt{Component\_Area_i}$, for each i $\in$ dataflow path). Table 2 lists our simulation parameters and Table 3 lists the % of energy spent in each accelerator and the ratio of energy spent in caches relative to compute.

5 Evaluation

In this section, we evaluate the proposed *FUSION* architecture and present our results as a set of "Lessons Learned" in the design and implementation of a lightweight coherent cache hierarchy for accelerators. The *FUSION* architecture is compared to *SCRATCH* and *SHARED* designs and we present results for overall performance and energy before discussing the evaluation of i) Write-Back vs Write-Through at the L0X ii) accelerator cache sizes iii) address translation iv) the *FUSION-Dx* design, optimized for producer consumer data sharing.

5.1 Performance

Lesson 1 : Small shared cache (L1X) improves performance. The *SHARED* system, as described in [41], employs a shared 64KB cache to filter out accesses to the L2. Figure 6b, shows the cycle time of each system normalized to the SCRATCH system. *FFT, DISP., TRACK., HIST.* spend a significant amount of time (82%) in DMA transfers and the *SHARED* system outperforms the *SCRATCH* system (average 5.71×). For *ADPCM, SUSAN* and *FILT.*, where the DMA cycle time is less than 40% of the total for the *SCRATCH* system, the *SHARED* system degrades performance by 14%. For these 3 benchmarks the working set size is less than 30kB. The high spatial locality is captured in the *SCRATCH* system and offers low latency access to the data. Thus the higher penalty per access to the shared L1X in the *SHARED* design causes a performance degradation.

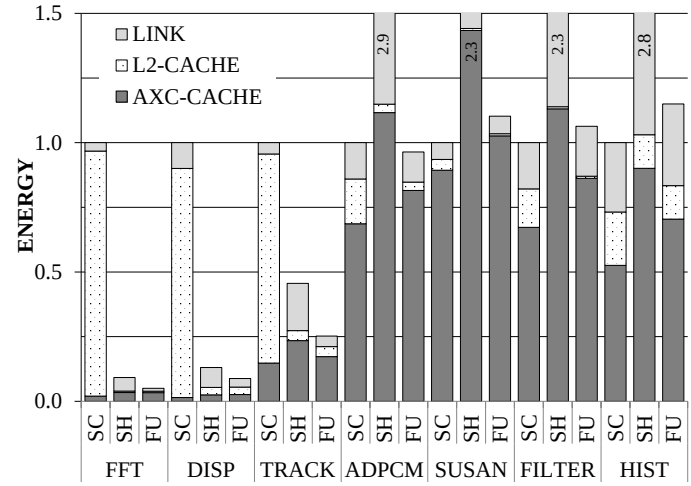
Lesson 2 : Small private caches (L0X) are needed to optimize for load-to-use latency. The *FUSION* system builds upon the *SHARED* system with the addition of coherent private caches which are of the same size as the scratchpad in *SCRATCH* system. The *FUSION* system is able to capture the spatial locality for *SUSAN, FILT.* and *ADPCM* which is the cause of degradation in the *SHARED* system. The *FUSION* system improves performance over *SCRATCH* by 2.8×.

5.2 Energy

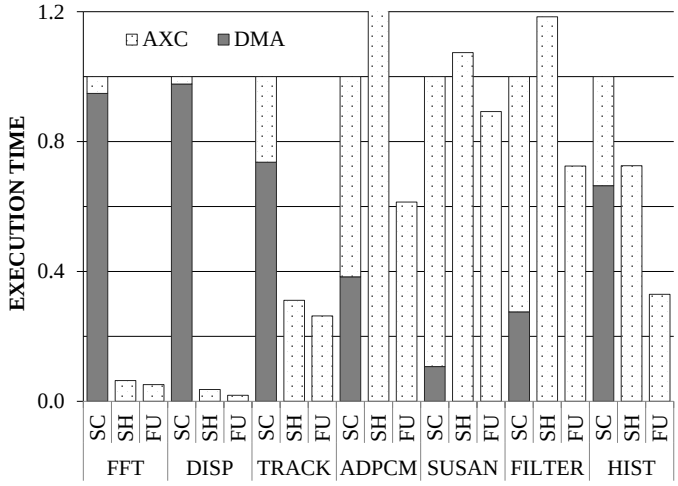
The energy breakdown of the benchmarks are presented in Figure 6a. We observe that the energy tradeoffs of pull-based cache architectures are different from that of push-based DMA execution models. The results in this subsection indicate that when optimizing for energy, a single architectural paradigms does not fit all applications.

For the *SCRATCH* system, *FFT* and *DISP.* are dominated by the L2 access energy due to repeated inter-AXC DMA transfers (963 and 640 respectively, see Table 6d). The large ratio of data transferred via DMA, column DMA(kB), compared to working set size, column WSet(kB), is an indicator of such pathological behaviour (165 for *FFT*). The *SHARED* system caches the AXC shared working set, eliminates spurious L2 accesses and reduces energy consumption by 10.6× and 7.6× for *FFT* and *DISP*.

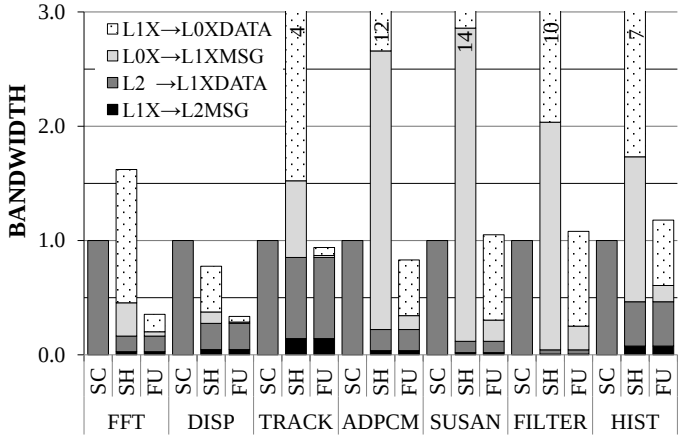
Lesson 3 : Small private caches (L0X) also improve energy. The *FUSION* system further reduces energy by intro-



(a) Energy



(b) Execution Time



(c) Bandwidth Breakdown

	Energy	WSet(kB)	# DMA	DMA #lines	DMA(kB)	DMA / WSet
FFT	✓	15	640	79550	2486	165.0
DISP.	✓	163	963	130164	4067	25.0
TRACK.	✓	371	524	67450	2107	5.7
ADPCM	✓	28	7508	7508	234	8.3
SUSAN	–	21	3466	3466	108	5.1
FILT.	–	21	15034	15034	470	22.0
HIST.	–	1191	95613	95613	2988	2.5

(d) DMA Breakdown

Figure 6: Design tradeoffs in the accelerator cache hierarchy. X-Axis SC : *SCRATCH*, SH : *SHARED*, FU : *FUSION*. Y-Axis: All plots, lower is better and values are normalized to *SCRATCH* system. Note for the *SHARED* design, the L1X→L0XDATA represents response from shared L1X to AXC and L0X→L1XMSG represents requests from AXC to the shared L1X. For the *SCRATCH* design, there is only one link for data from L2 to the local scratchpad.

ducing a 4K L0X which is 1.5× more energy efficient than even a heavily banked L1X and filters out 83% and 80% of the accesses (effect seen in Figure 6c) to the L1X for *FFT* and *DISP.* respectively significantly reducing the L0X-L1X link energy. *TRACK.* also spends a large fraction of energy in L2 accesses due to a large working set (371kB). Function *imgResize*, shares 99% of its data access (173 kB), triggering inter-AXC DMA transfers. The *SHARED* system and the *FUSION* system do not incur this overhead.

Lesson 4 : The L1X filters accesses to the L2 but L0X→L1X coherence message overhead is significant. *FILT.* has a large ratio of DMA data transferred with respect to working set size and *SHARED* and *FUSION* designs save energy by eliminating L2 accesses (filtered by shared L1X). This can be seen as the diminished L2 stack of Figure 6a.

However these gains are lost to repeated thrashing behaviour of the L0X as the benchmark iterates over each pixel in the image. This increases coherence request messages between L0X and L1X (see Figure 6c), expending significant energy. Similar behaviour is also observed in *SUSAN* and *HIST*. *HIST.* incurs additional penalty of coherence request messages (L1X→L2) for the *SHARED* and *FUSION* designs; large working set (1191kB) does not fit in L1X. The *FUSION* design mitigates some of the degradation observed in the *SHARED* system (see Figure 6c), but not enough to provide an overall energy benefit for *HIST*.

ADPCM sees a modest improvement of 4% as most of what is gained from the reduction in L2 accesses is lost in repeated L1X accesses. Overall *FUSION* reduces energy consumption by 2.4×, however for *HIST, SUSAN* and *FILT.*, *FUSION* in-

creases energy consumption by 10% (improves performance by 67%). The *SHARED* system performs poorly in general due to the higher penalty of link energy for a) messages from AXC→L1X (see Figure 6c) b) data from L1X→AXC and c) access energy for the shared L1X cache.

5.3 Writeback vs Write-Through at L0X

Lesson 5 : Write-through caches are expensive in terms of energy. Recent work [14, 32] have studied the effect of writes in data parallel accelerators and highlights their "bursty" behavior in GPGPU applications. For such applications, the contention introduced by writeback operations may evict freshly read data. We find that in fixed-function accelerators, write caching at the L0X is an important requirement to exploit the inherent data locality of the offloaded functions. Write-through adds energy overhead due to data transfers on the L0X→L1X link and L1X data access energy. In Table 4 we list the bandwidth consumption of both write-through and writeback models. We find that fixed-function accelerators that offload functions from existing programs reflect the locality behavior of the original program although the memory level parallelism may increase.

Table 4: Bandwidth in Flits (8bytes/flit)

	Write-Through	Writeback	% Dirty Blocks
FFT	230232	6642	39.5
DISP.	142656	40896	37.7
TRACK.	2266764	19428	45.3
ADPCM	3188262	14100	46.9
SUSAN	13973187	81600	35.1
FILT.	5728776	12096	51.3
HIST.	18575484	194316	46.4

5.4 *FUSION-Dx*: Write Forwarding

	# FWD Blocks	AXC Cache	AXC Link
FFT	4309	6.4%	16.9%
TRACK.	4582	1.5%	5.7%

Table 5: Inter-AXC forwarded blocks and percentage reduction in energy consumption per component

Lesson 6 : Protocol extensions can exploit inter-AXC producer consumer relationships. Optimized hardware realizations of large functions are often split into smaller blocks [12]. Using a shared cache introduces writebacks for data which is written by an accelerator and read by the next, effectively creating a producer-consumer relationship. *FUSION-Dx* optimizes for such relationships as described in Section 3.2. For each block forwarded from an L0X, the *FUSION-Dx* system saves energy spent by the *FUSION* system in 1 WriteBack to L1X, 1 Read from L1X and 1 request from L0X→L1X, while incurring the significantly lower cost of a L0X→L0X transfer (0.1pJ/byte). Table 5 enumerates the total number of blocks forwarded between AXC's of *FFT* and *TRACK.* and the corresponding savings in energy for the AXC component.

5.5 Larger AXC caches

Lesson 7 : Larger may not be better. We experimented with a larger AXC cache configuration (AXC-Large) where the L0X was 8KB (2×) in size while the L1X was 256KB (4×) in size. The working set sizes of the workloads were such that only 1 benchmark (*DISP.*) fit into the Large-L1X (163kB footprint, see Table 6d) amongst the ones which did not fit into the Small-L1X (*TRACK., HIST.* and *DISP.*). For *ADPCM, SUSAN* and *FILT.* (working set sizes smaller than 30kB), the severe degradation seen in Figure 7 is due to the higher L1X access energy (2× as much as L1X-Small). There were negligible drops in miss rate for *DISP.* and *TRACK.* (less than 10 blocks) at the L0X. *DISP.* experienced 22% drop in L1X misses, which translated to a 3% reduction in cycle time (mostly obviated by the increased L1X access latency; 2 cycles more than L1X-Small).

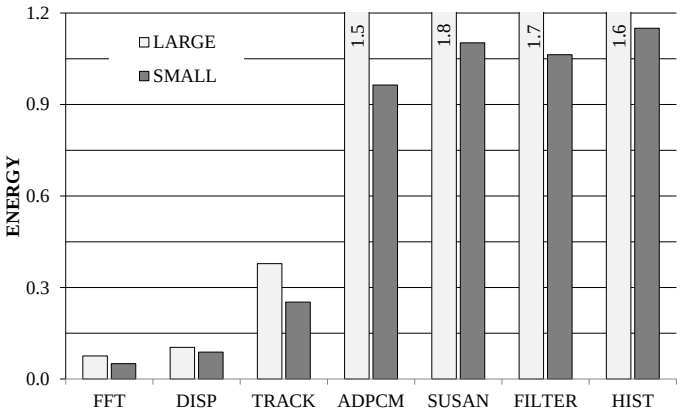


Figure 7: Comparing the benefits of LARGE (L0X:8KB,L1X:256KB) vs SMALL (L0X:4KB,L1X:64KB)

5.6 Address Translation

Bench	AX-TLB	AX-RMAP
FFT	514	41
DISP.	4243	589
TRACK.	3237	831
ADPCM	1447	548
SUSAN	671	6
FILT.	668	18
HIST.	60K	20

Table 6: Virtual memory table look up count

Lesson 8 : Address translation overheads need to be mitigated. The address translation energy is an important consideration in coherent cache hierarchies [18]. *FUSION* places the TLB off the critical path and on the shared L1X's miss path where the request has to transition into the physical address space. Here we illustrate the benefits by listing the number of look ups in the TLB for the baseline (64KB shared L1X). *FUSION* also needs reverse map look ups on forwarded requests from the host's shared L2. The host's directory tracks the

accelerator tile in the sharer list and only forwards requests to lines cached in the accelerator tile. Table 6 lists the number of look ups to the AX-RMAP and AX-TLB. Overall, we expend less than 1% of the energy on the AX-RMAP and AX-TLB; workloads that overflow the shared L1X and generate more misses, could possibly expend more energy.

6 Related Work

System-On-Chips: Chip designers have recognized the need for reducing the overhead of communication between the accelerator and the host processor. Current ARM multicores include an AXI bus [10] that snoops the shared L2 on the multicore; similarly IBM's Power processor [4] includes a Powerbus that ensures the most-up-to-date data is read from the processor. Both systems are similar to the *SCRATCH* configuration we study. The host processor and DMA is required to move in and out of the scratchpad and between the accelerators. The DMA overheads are minimal when accelerators are computationally intensive and read few data elements (e.g., cryptographic units) or have minimal interaction with each other (e.g., XML and cryptographic accelerators in PowerEN [5]). Past work [15, 20] have also studied the benefits of integrating network cards with the last level cache of the chip. The type of accelerators we study in this paper are functions extracted from a sequential program that have plenty of read-write sharing. In such cases, involving the DMA controller for moving shared data expends energy in the memory hierarchy and adds latency to the critical path of the accelerator. The *FUSION* designs we study in this paper implicitly move data between the accelerators directly and minimizes the overheads of the locality lost as a result of the execution migration between the different accelerators.

In-core Accelerators: Recent research from academia [11,26, 36] has extracted accelerators from sequential programs and have proposed to integrate these accelerators at- the-core and leverage the host processor's L1 cache. Integrating tightly with the L1 cache enables the accelerators to maintain coherence. Unfortunately, the load-to-use latency of the L1 cache shared between the accelerators may introduce a performance overhead. A pertinent example is the *demosaic* benchmark [26] which does not see as much of a performance gains as the other workloads due to the abundance of loads and stores. The *SHARED* configuration is a representative design and our analysis reveals that using a 64K shared cache to supply the accelerators results in a 2.7× energy overhead compared to using a cache hierarchy for *HIST., FILT., SUSAN* and *ADPCM* while saving 83% of energy for *FFT, DISP.* and *TRACK*. [8] proposes an interesting design where the accelerator-cache interface is configurable. The accelerator may choose to share the L1 data cache of the core or use it's own private data cache. The coprocessor dominated architecture proposal [41], ensures that accelerators with higher memory traffic are placed closer to the shared L1 data cache of the host to reduce energy consumption. However typical L1 caches are designed to meet the strict cycle-time constraints of the host processor and supplying data to accelerators with much higher memory level parallelism (up to 6×) is challenging. *FUSION* manages the cache hierarchy between un-core accelerators using a lightweight hardware coherence protocol and we have also explored the benefits of actively forwarding data between the accelerators.

Coherence Forwarding: Past work in academia [33,39] have extensively studied the benefits of proactively forwarding and streaming data in multiprocessors to reduce cache misses and improve performance. Current multicores [9] have also included coherence states to help with forwarding data between caches directly. Herein we have studied the benefits of proactively pushing data to save energy. While past work studied the addition of forwarding support to lazy coherence protocols [17] we have shown the benefit of adding forwarding to a time-stamp based protocol [31, 32].

Un-core Accelerators: Vuletić et al. [38] propose a hardware memory management unit (WMU) to allow accelerators to operate in the virtual address space of the invoking process. DASX (data structure accelerator) [19] leverages LLC TLBs present in modern multicores to issue accesses to memory and maintains coherence at kernel boundaries. Recent research [6, 40] incorporates application specific streaming frameworks independent of the host processor's cache hierarchy. *FUSION* allows accelerators to issue virtual memory addresses while removing address translation from the critical path; it also supports a flat coherence model.

7 Summary

The design tradeoffs for a coherent cache hierarchy for fixed-function accelerators have been evaluated. With the increasing energy cost of interconnects and caches relative to compute, it is imperative to optimize data movement to retain the energy benefits of accelerators. We develop *FUSION*, a lightweight multi-level cache hierarchy for accelerators and study the tradeoffs compared to a scratchpad-based architecture. *FUSION* leverages proposed time-stamp based coherence [22,31,32] to maintain coherency efficiently amongst the accelerator caches as well as integrating them with the MESI protocol. We find that i) small L0 private caches are essential to retain the energy benefit of accelerators and ii) shared L1 caches help optimize data movement between the functions offloaded from the same program and minimize host-accelerator data transfers. A comprehensive toolchain was developed for modelling fixed-function accelerators in a cycle accurate manner and used to study the tradeoffs compared to optimized DMA code. We study multiple facets of the cache hierarchy including write optimizations and evaluate the tradeoffs between the pull-based model of the cache hierarchy and push-based model of the DMA.

References

[1] Macsim : Simulator for heterogeneous architecture - https://code.google.com/p/macsim/.

[2] J. Balfour. EFFICIENT EMBEDDED COMPUTING.

[3] A. Basu, M. D. Hill, and M. M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[4] B. Blaner, B. Abali, B. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. Reilly, and P. Sandon. Ibm power7+ processor on-chip accelerators for cryptography and active memory expansion. *IBM Journal of Research and Development*, 57(6):3:1–3:16, Nov 2013.

[5] J. Brown, S. Woodward, B. Bass, and C. Johnson. IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *IEEE Micro*, 31(2):76–85, 2011.

[6] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance, 2013.

[7] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 878–878. IEEE, 2011.

[8] A. Farmahini-Farahani, N. S. Kim, and K. Morrow. Energy-efficient reconfigurable cache architectures for accelerator-enabled embedded systems. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 211–220. IEEE, 2014.

[9] J. Goodman. Source Snooping Cache Coherence Protocols. *Science*, 2009.

[10] Goodridge. The effect and technique of system coherence in arm multicore technology. 2008.

[11] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):0038–51, 2012.

[12] Y. S. S. B. R. Gu and Y. W. D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures.

[13] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *PROC of the 37th ISCA*, 2010.

[14] J. Hestness, S. W. Keckler, and D. A. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior,. In *IEEE International Symposium on Workload Characterization*, 2014.

[15] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *PROC of the 32nd ISCA*, 2005.

[16] Intel. Xeon chip with integrated fpga. 2014.

[17] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *Micro, IEEE*, 30(5):54–65, Sept 2010.

[18] S. Kaxiras and A. Ros. A new perspective for efficient virtual-cache coherence. In *PROC of the 40th ISCA*, pages 1–12, Apr. 2013.

[19] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan. DASX : Hardware accelerator for software data structures. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS 2015, june 2015.

[20] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *PROC of the 40th ISCA*, 2013.

[21] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.

[22] S. L. Min and J. L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Trans. Parallel Distrib. Syst.*, 3(1), 1992.

[23] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *PROC of the 40th MICRO*, 2007.

[24] E. Peter Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. In *DARPA IPTO*, September 2008.

[25] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. In *PROC of the 36th ISCA*, 2009.

[26] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *PROC of the 40th ISCA*, pages 1–12, Apr. 2013.

[27] B. Reagen, R. Adolf, S. Y. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[28] G. Research. Arm cto warns of dark silicon. 2010.

[29] S. Ricketts. Efficient Cache-Coherent Migration for Heterogeneous Coprocessors in Dark Silicon Limited Technology.

[30] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor. Efficient complex operators for irregular codes. In *PROC of the 17th HPCA*, 2011.

[31] K. S. Shim *et al.* Library Cache Coherence. Csail technical report mit-csail-tr-2011-027, May 2011.

[32] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for gpu architectures. In *HPCA*, pages 578–590, 2013.

[33] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *PROC of the 36th ISCA*, 2009.

[34] D. Stasiak, R. Chaudhry, D. Cox, S. Posluszny, J. Warnock, S. Weitzel, D. Wendel, and M. Wang. Cell processor low-power design methodology. In *Micro, ieee*, 2005.

[35] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. *SD-VBS: The San Diego Vision Benchmark Suite*. IEEE, Oct. 2009.

[36] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[37] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 163–174. ACM, 2011.

[38] M. Vuletic, P. Ienne, C. Claus, and W. Stechele. Multithreaded virtual-memory-enabled reconfigurable hardware accelerators. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 197–204. IEEE, 2006.

[39] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal Streaming of Shared Memory. In *PROC of the 32nd ISCA*, 2005.

[40] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *PROC of the 40th ISCA*, 2013.

[41] Q. Zheng, N. Goulding-Hotta, S. Ricketts, S. Swanson, M. B. Taylor, and J. Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *ACM Trans. Embed. Comput. Syst.*, 13(4s):130:1–130:24, Apr. 2014.

Appendix: Synonyms

As others have noted [3] synonyms (particularly) read-write synonyms are not prevalent in applications; we did not find any in our applications. To support synonyms within the same program i.e., multiple virtual addresses accessing the same physical address, we check the AX-RMAP on accelerator tile requests. If any are found in the accelerator tile caches, the duplicate is evicted i.e., only one synonym is permitted in the accelerator tile. *FUSION* does not support sharing of data between functions offloaded from different processes.