Template-based Memory Access Engine for Accelerators in SoCs

Bin Li, Zhen Fang, and Ravi Iyer
Intel Labs, Hillsboro, Oregon, USA
{bin.li, zhen.fang, ravishankar.iyer}@intel.com

Abstract – With the rapid progress in semiconductor technologies, more and more accelerators can be integrated onto a single SoC chip. In SoCs, accelerators often require deterministic data access. However, as more and more applications are running simultaneous, latency can vary significantly due to contention. To address this problem, we propose a template-based memory access engine (MAE) for accelerators in SoCs. The proposed MAE can handle several common memory access patterns observed for near-future accelerators. Our evaluation results show that the proposed MAE can significantly reduce memory access latency and jitter, thus very effective for accelerators in SoCs.

I. Introduction

Nowadays, embedded handheld devices have become increasingly popular. To speed up time-to-market, Systems-on-a-chip (SoCs) have become the design platform for these handheld devices. Fig. 1 shows an example of today's SoC architectures. In this architecture, one general-purpose core (such as Intel's Atom™ [1] and ARM's Cortex A9[2]) along with several accelerators (video decoders, graphics, imaging, etc.) are connected to a memory controller through interconnect. In this environment, the requirements placed by cores and accelerators are typically quite different. For cores, the memory access is usually best-effort style and cores employ caches to capture the working set. For accelerators, they often require deterministic access, and use buffers to store data.

With the rapid growth of semiconductor technologies, more and more processor cores and accelerators can be integrated onto a single SoC chip. Furthermore, it is expected that several applications will be running simultaneously on such platform and need to share the memory system [20]. However, this results in memory bandwidth contention problems between cores and accelerators as well as among different accelerators. This contention problem leads to increased memory access latency and jitter for accelerators, making memory access latency to be unpredictable. The long memory access latency and large jitter places a heavy burden for accelerator design and greatly affects the performance for accelerators.

To hide the long memory access latency, researchers have proposed different techniques to prefetch data from off-chip memory to on-chip storage, such as software prefetching [4,5,6,7] and hardware prefetching [8,9,10,11,12]. Most of these prefetching techniques are speculative. They work well for general purpose programs. However, accelerators require non-speculative data access. Current accelerator design usually utilizes a local direct memory access (DMA) controller to coordinate data movement between off-chip memory and its local buffer. However, dedicated DMA logic

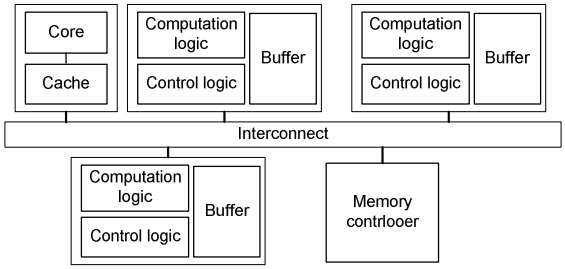


Fig. 1 Today's SoC architecture

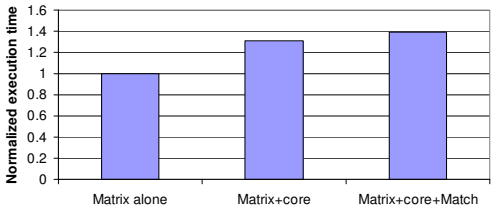


Fig. 2 Matrix accelerator performance degradation

at local accelerator does not have a global view of the contention. When the contention at memory increases, the latency will be increased accordingly. Accelerator design can also utilize the system provided central direct memory access (DMA) controller to fetch data from off-chip memory to on-chip storage, but current central DMA only supports simple access patterns such as streaming access and scatter/gather access. For complex memory access pattern, the accelerator has to issue multiple requests to the DMA. Furthermore, current central DMA employs round-robin or fixed priority arbitration for different requests. This works well when the memory bandwidth requirement from each request is low. However, it greatly increases the design complexity when multiple applications are contending for memory bandwidth simultaneously.

Fig. 2 shows an example of performance degradation for *Matrix accelerator* with local DMA design when we increase the number of applications running simultaneously (*Matrix* is an accelerator for matrix multiplication computation. *Match* is an accelerator for image processing. Cores are running SPEC [3] applications *art* and *mcf*. Detailed experimental setup is provided in Section V). From Fig. 2, we can see that the execution time for *Matrix accelerator* continues to increase as we increase the number of applications running on the platform. This is because the memory access latency is increased due to contention. As the number of processing elements on chip increases, the memory access latency issue will become more and more severe. It is very difficult for accelerator itself to solve the problem as it does not have a global view. On the other hand, many accelerators are usually working on one or two tasks and their memory access traffic

978-1-4244-7516-2/11/$26.00 ©2011 IEEE

patterns are known at design time. Moreover, many accelerators have similar memory access patterns. If we can utilize the common characteristics of the accelerators and coordinate the memory access requests at global, the performance for each accelerator can be improved.

In this paper, we propose a template-based global memory access engine (MAE) for accelerators. The MAE provides several common memory access templates for accelerators, which can deal with simple as well as complex memory access patterns. It consists of a template-based prefetcher and a prefetch buffer. The MAE is located next to the memory controller. The accelerator can utilize the MAE to prefetch data from off-chip memory to on-chip buffers. Because MAE has a global view about the memory system as well as the requirements from each accelerator, it decides when to send out requests to the memory controller based on the information it collected. The accelerator now only needs to go to MAE to read the data. As a result, the memory access latency perceived by the accelerator is much smaller, and jitter is reduced significantly.

The rest of the paper is organized as follows. Section II describes related work. Section III presents some common memory access patterns by accelerators. Section IV describes the templates we proposed for different memory access patterns and the detailed design of the MAE. Section V presents our experimental methodology and evaluation results and Section VI concludes the paper.

II. Related Work

Researchers have proposed different techniques to hide the long memory access latency, such as software prefetching [4, 5, 6, 7] and hardware prefetching [8, 9, 10, 11, 12]. In software prefetching, the compiler inserts prefetch instructions that bring the indicated block of data onto on-chip memory. In hardware prefetching, the hardware predicts prefetch addresses by observing a program's runtime behavior. Most of these techniques are speculative. They work well for general purpose programs where long memory access latencies are undesirable but tolerable. However, accelerators typically require non-speculative data access. As a result, speculative prefetching techniques are not suitable for accelerators.

Current accelerator design can use system provided central DMA controller to fetch data from off-chip memory to on-chip storage [13, 14, 15]. However, existing DMA only supports simple access patterns. Besides, current DMA only employs round-robin arbitration or fixed priority arbitration. It cannot prioritize requests based on the congestion situation. Our proposed template-based MAE is similar to DMA in concept. The difference is that we provide more complex memory access patterns, making the MAE more intelligent. The accelerator only needs to setup the MAE and sends out simple commands. The MAE then fetches the data from off-chip memory to the on-chip storage automatically. Furthermore, our MAE can handle the contention problem. It prioritizes the requests based on the feedback from buffers to ensure a balanced produce-consume rate. As a result, the memory access latency and jitter are reduced to a much smaller range as perceived by the accelerator, especially in the situation where multiple applications are running simultaneously. This in turn reduces the design complexity for accelerators.

III. Common Memory Access Patterns by Accelerators

We observe that emerging applications targeting accelerator systems have common data access patterns, which if used wisely, can yield significant application performance improvement. In this section, we summarize the common memory access patterns as below:

- **Streaming** Many accelerators, such as video post processors, display controllers, advanced encryption standard (AES) accelerators, fetch a continuous chunk of data from memory into a local buffer for processing. This is perhaps the most commonly used memory access pattern.
- **Strided** Some accelerators fetch data that have fixed strides. That is, the address of next block of data is usually a fixed distance from the current address. For example, the accelerator for matrix multiplication needs to fetch data from one column of the matrix in which the stride is the row size.
- **Complex** Some accelerators have regular but complex data access patterns. For example, Mobile Augmented Reality (MAR) is an emerging usage model on handheld devices [16]. One important step in MAR computation is Hessian computation. As shown in Fig. 3, to compute one point (the central point in Fig. 3) in the image, it requires many points in its neighbor. The distances from other points to this central point though all different, are all fixed distances that are known *a priori* for the application. This type of data has a feature that if one point's address is known, the other points' addresses can be calculated by the distance from this point. This kind of data access pattern is widely used in image processing.
- **Linked-list** Some accelerators access linked-lists in the memory. We take on-die network interface controller (NIC) as an example. In this architecture, the NIC moves packets from memory to the NIC buffer or vise versa. The NIC operation requires two data structure: a header descriptor and payload buffers. The header descriptors are used to contain information for each payload buffer and are usually maintained as a linked-list data structure.
- **Indirect array access** Some accelerators use an array to index a second array. For example, for array A[B[i]], the accelerator first fetches data for B[i]. It then uses the returned value from B[i] as the index for array A and fetches the corresponding data. One example of A[B[i]] access is histogram calculation, which is a common operation in many image processing applications.
- **Gather access** The gather access is usually used when accelerator reads (gathers) multiple data from non-contiguous locations in the memory. The accelerator provides the absolute address as well as the amount of data to fetch at each location. This kind of memory access pattern is very popular in accelerators, such as vector execution [17].

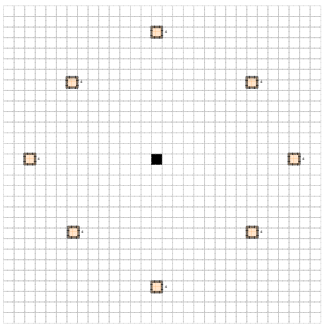


Fig. 3 Example of a complex but regular data access pattern (hessian matrix)

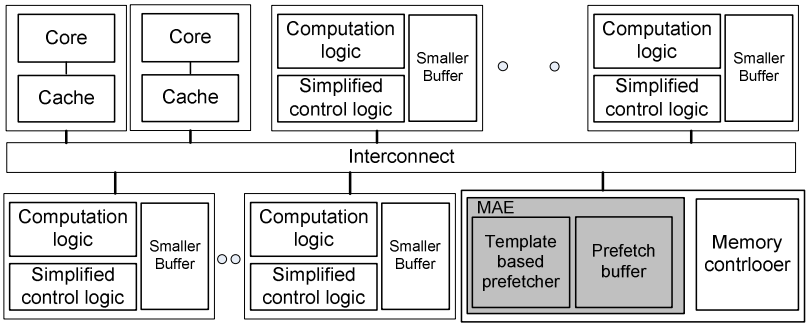


Fig. 4 Future SoC architecture with proposed MAE

Collectively, these memory access patterns cover a wide range for accelerators. Most accelerators that we have surveyed have at least one or two access patterns as introduced above. This indicates the potential for an on-chip MAE that can handle these memory access patterns simultaneously.

IV. Template-based MAE Architecture Design

After describing the common memory access patterns for accelerators, we now describe our proposed template-based MAE for accelerators in this section. Fig. 4 illustrates such an architecture with the MAE we proposed for future SoC architectures. In this architecture, the proposed MAE resides next to the on-chip memory controller. It has better knowledge about the congestion in memory systems and decides when to send requests to memory controller to satisfy accelerator requirements. It helps accelerators to prefetch the data from off-chip memory to on-chip prefetch buffers. The centralized prefetch buffer covers bulk of the memory access latency and jitter caused by off-chip memory such that the local buffers within each accelerator can be reduced. Each accelerator also has simplified control logic now because the MAE carries most of the work.

In this section, we first describe the memory access templates that our MAE supports. We then present the architecture design of the MAE.

A. *Prefetch templates*

Based on the memory access patterns discussed in Section II, we provide different prefetch templates in our MAE. Each accelerator first sets up its own templates before usage. It then sends out prefetch commands to the MAE during execution. The MAE automatically fetches the data based on the template the accelerator specified as well as the information provided in the prefetch command. Table 1 summarizes the prefetch templates we supported in our MAE.

Note that our templates are for data read accesses only. This is because the amounts of writes are much smaller than reads in accelerators. Hence, we only focus on data read access in our design. The proposed method works in physical address space. We expect device drivers for the accelerators use contiguous physical address regions. Though there have been proposals for accelerators to use virtual addresses, physical address space is still by far the dominant addressing domain that accelerators (and their drivers) use. On the other hand, our design will work well if the accelerators use virtual address, in which case the memory requests simply go through the input/output memory management unit (IOMMU) first before they are presented to the MAE. We will introduce our MAE design in detail in the next section.

B. *MAE architecture design*

Fig. 5 shows the architecture of our template-based MAE design. The accelerator can send either prefetch commands or read commands to MAE through interconnect. The prefetch command is the command that an accelerator sends to the MAE to prefetch data from off-chip memory to MAE. The read command is to move data from MAE to accelerator's

Table 1 Prefetch templates

PF template ID	Format					
ID 1 (Streaming)	Start address			Byte Count		
ID 2 (Strided)	Start address	Byte count	Stride		Repeating times	
ID 3 (Complex)	Start address	Offset	Offset	Offset	Offset	Offset
	Offset	Offset	Offset	Byte Count	Stride	Repeating times
ID 4 (Linked-list)	Head address	Tail address	Offset for next element address		Offset for payload start address	
	Offset for payload size		Current address		Payload byte count	
ID 5 (Indirect array)	Inner array start address		Outer array start address		Element offset	
ID 6 (Gather)	Address	Byte count	Address	Byte count	Address	Byte count
	Address	Byte count	Address	Byte count	Address	Byte count

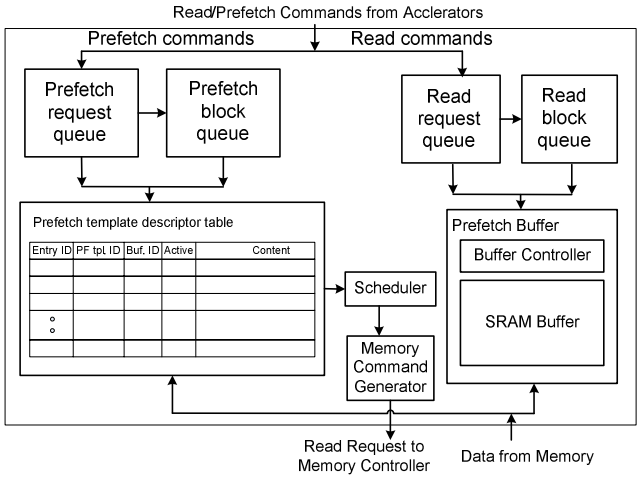


Fig. 5 MAE architecture overview

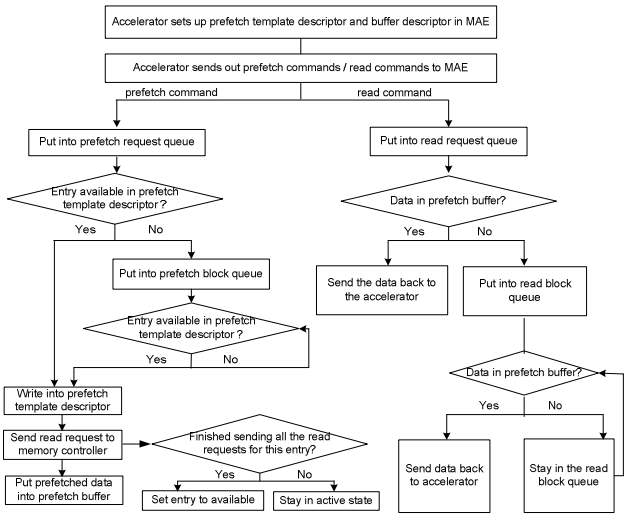


Fig. 6 MAE flow chart

local buffer. For read commands that cannot be prefetched as well as write commands, the accelerators send them directly to memory controller (bypass MAE). At the interface of the MAE, the prefetch commands are put into the prefetch request queue while the read commands are put into the read request queue. The prefetch template descriptor records the necessary information required for each prefetch command. The scheduler decides which entry in the prefetch template descriptor will be served first. The memory command generator generates the address and forms the read command which will be sent to the memory controller. The prefetch buffer is used to store prefetched data. Fig. 6 shows the flow chart of the MAE. We will explain each one in detail next.

Prefetch template descriptor table. Prefetch template descriptor table is the place where each accelerator sets up its memory access pattern. The accelerator can setup the prefetch template descriptor entry either statically at setup time or dynamically during execution. In our design, each accelerator can setup and use multiple descriptors. Each entry in the prefetch template descriptor table contains an *entry ID*, which specifies the entry number for this template descriptor; a prefetch template ID (*PF tpl. ID*), which specifies the type of memory access pattern for this entry; a buffer ID (*Buf. ID*) which indicates the buffer the prefetched data will be written into; an active bit which indicates if the entry is currently active (busy) or is available for new prefetch commands; and the content where all the required information is recorded for this memory access pattern as illustrated in Table 1. If there are more accelerators than can be handled by MAE (e.g. no empty prefetch descriptor template entry available), the MAE can reject the application for prefetching. The accelerator now has to go to memory directly to fetch the data at the cost of reduced performance. If the prefetcher rejects the application, the decision can be made based on the priority of the application.

Prefetch request queue/block queue. After MAE receives the prefetch command sent from accelerators, it first puts the command into the prefetch request queue. At the head of this prefetch request queue, MAE checks if the corresponding entry in the prefetch template descriptor that matches this *entry ID* is available. If so, the prefetch command is written into the prefetch template descriptor and removed from the prefetch request queue. Otherwise, the prefetch command is put into the prefetch block queue. Fig. 6 also shows the flow chart for prefetch requests.

Scheduler. The scheduler makes decisions about which entry from the prefetch template descriptor will be processed next based on buffer status at prefetch buffer and requirements from accelerators. If the data in prefetch buffer is consumed fast, the scheduler will service this prefetch request entry earlier than entries that still have data for accelerators to consume.

Memory command generator. The memory command generator reads in the information from prefetch template descriptor and generates the address for each READ request it will send to the memory controller.

Prefetch buffer. The prefetch buffer consists of a buffer controller and a SRAM storage. The buffer controller maintains a set of buffer descriptors. It also has an address translator based on *accelerator ID* to translate the address from accelerator to buffer address in SRAM. The SRAM storage holds the prefetched data. The accelerator initializes the buffer descriptor for which it is going to use at setup time. A buffer is also allocated for each buffer descriptor in the SRAM storage. The SRAM are shared by all the active accelerators. Each accelerator can setup multiple buffers in this SRAM storage.

Read request queue/block queue. The read command from the accelerator is first put into the read request queue. At the head of this read request queue, the MAE checks if the requested data is available. If so, MAE sends back the data from the prefetch buffer to the accelerator. If not, the read request is put into the read block queue. The read block queue continuously checks the prefetch buffer whenever a new data is received from the memory. If the new data matches the

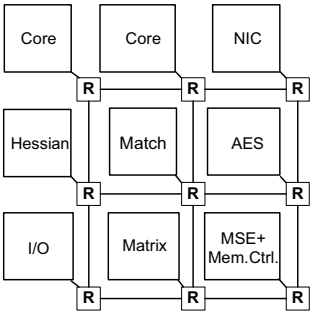


Fig. 7 SoC architecture evaluated

Table 2: Simulation parameters

Subsystem	Description
Core	1 GHz low power core (in-order, 2-issue)
Cache	256KB cache for each core, 64B cache line
NoC	3x3 mesh, 5x5 crossbar in each router, dimension order routing, 3 VCs, 5 buffers per VC, 4 router pipeline stages
Memory Bandwidth	3.2GB/s maximum sustained 1-channel low power DDR
Memory Latency	100 core clocks (100ns)
Prefetch Buffer	32KB (shared by all the active accelerators in the platform)
Prefetch Queue Size	Block queue: 8 Request queue:8
Read Queue Size	Block queue: 8 Request queue: 8
Prefetch template descriptor entries	10

request in the read block queue, the MAE sends the data back to the accelerator and removes the entry from the queue.

The flow chart of our proposed template-based MAE is summarized in Fig. 6. With our template-based MAE, the accelerator now only sends out very simple prefetch command to the MAE for the portion of data that can be prefetched. The MAE then prefetches the data from off-chip memory into the prefetch buffer based on the template specified by the accelerator as well as the status for each stream. The MAE greatly reduces the design complexity for accelerators. It also reduces the average memory access latency and jitter as perceived by the accelerators.

V. Evaluation

In this section, we evaluate the effectiveness of our proposed template-based MAE. We present comparison results of our MAE with the current design where local DMA is used. We show results when only one application is running at a time as well as results where several applications are running simultaneously.

A. Simulation methodology

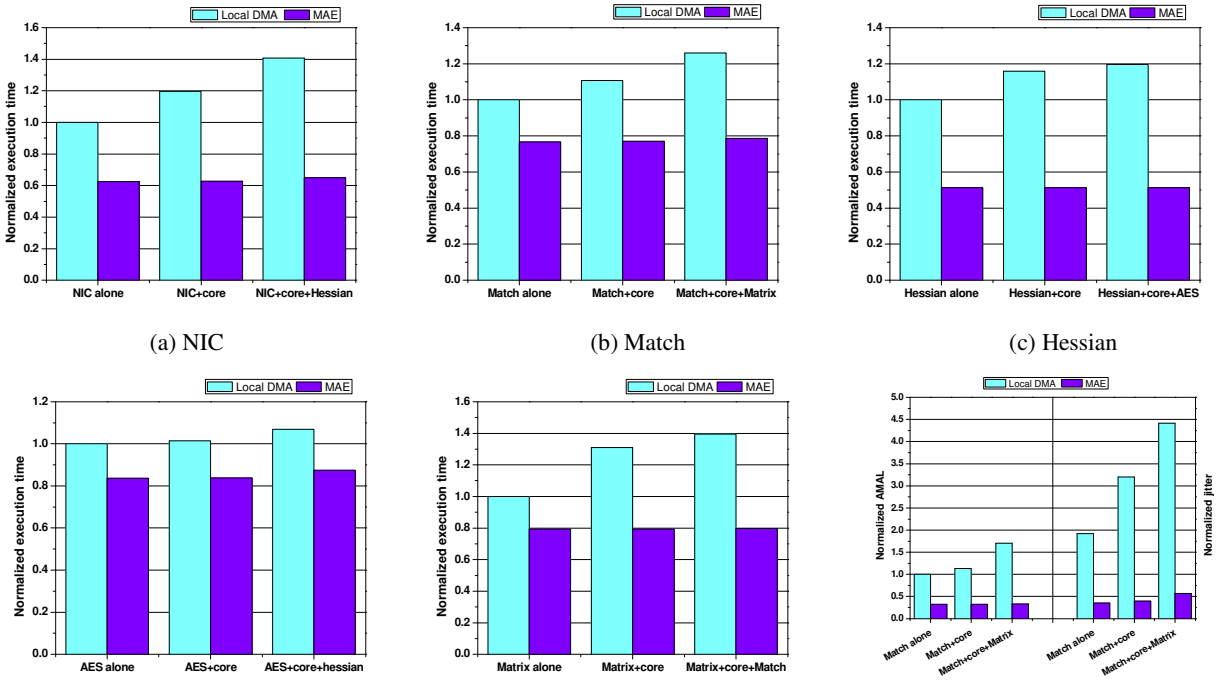
In order to evaluate our template-based MAE, we employ an evaluation methodology that combines real, validated implementation of accelerators and cycle-accurate software simulation. Accelerator traces along with detailed timing information are derived from our hardware implementation. The traces are fed into a SoC simulator based on ASPEN [18] and GARNET [19], which together models a detailed cache hierarchy, memory subsystem, and a NoC-based interconnect as NoC is becoming the communication fabric in future SoCs. We added our proposed MAE to this ASPEN+GARNET simulator. We use two evaluation metrics. The first is execution time. The second is average memory access latency and jitter as perceived by each accelerator.

Our experiments are conducted to simulate the SoC configuration as illustrated in Fig. 7. There are 2 cores along with 5 accelerators and a peripheral I/O in this platform. Accelerators utilize MAE to prefetch the data. Cores go to the memory controller directly to fetch the data. All these agents (totally 9) are interconnected through a 3x3 mesh NoC. The five accelerators are (1) Network Interface Controller (NIC), which handles an interface to a computer network with linked-list data access pattern; (2) Advanced Encryption Standard (AES) accelerator, which has streaming access pattern; (3) Hessian accelerator for Mobile Augmented Reality (MAR). MAR is an emerging usage model on handheld devices that automatically recognizes user input object and displays information regarding the object [16]. It has recently gained significant interest in industry. MAR consists of two accelerators: (a) hessian for interest-point detection, which identifies interest points in the query image and has complex access pattern, and (b) match accelerator which compares descriptor vectors of the query image against descriptor vectors of each image in the data base; (4) Match accelerator in MAR, which has streaming access pattern; (5) Matrix accelerator for matrix multiplication, which has strided access pattern. The two cores ran SPEC CPU2000 application [2] (*art* and *mcf* chosen for their compute/memory intensive nature). The Hessian and Match accelerators have been implemented in Altera Stratix II FPGA and functionally verified on an x86 platform [16]. At the time of writing this paper, integration of the accelerators onto a test chip is ongoing. We setup the simulation parameters to reflect our target platform as close as possible. Table 2 shows the rest of the simulation parameters.

B. Simulation results

In this section, we evaluate the effectiveness of our proposed template-based MAE. We compare simulation results for three experiments: (1) only one accelerator runs on the platform, (2) one accelerator runs with two SPEC applications on cores, (3) one accelerator runs with SPECs and another accelerator.

Fig. 8(a)-(e) shows the simulation results for each accelerator. For each experiment, there are two bars, where the left bar is the execution time when the accelerator runs with local DMA. The right bar is the execution time when MAE is applied. The results are normalized to the execution time when accelerator runs alone. From Fig. 8, we can see that as the number of applications running on the platform increases and each accelerator relies on its local DMA to fetch data (left bar in each group), the execution time for each accelerator increases rapidly due to resource contention. For example, NIC's execution time is increased to 1.4X when running with cores and Hessian simultaneously. When we use MAE to prefetch data from off-chip memory to on-chip storage (right bar in each group), the execution time is reduced dramatically (e.g., reduced to 63% of the baseline



(a) NIC (b) Match (c) Hessian

(d) AES (e) Matrix (f) AMAL and jitter comparison for Match

Fig. 8 Execution time and AMAL/jitter improvements for accelerators with MAE.

execution time for NIC). Furthermore, we can see that the execution time for each accelerator now increases slowly when running with other applications simultaneously. This is because our MAE can effectively prefetch data for accelerators and hide the long latency in memory. As a result, the data access latency perceived by each accelerator is much smaller now. This demonstrates that our proposed MAE is very effective in reducing the execution time by prefetching data from off-chip memory to on-chip storage.

Next, we look at the average memory access latency (AMAL) and jitter (in terms of maximum memory access latency) perceived by accelerators before and after applying our MAE. Due to limited space, we only show the results for Match accelerator. We see similar trends for other accelerators as well. Fig. 8(f) shows our comparison results, which are normalized to the AMAL when Match is running alone and with local DMA. From Fig. 8(f), we can see that as the number of applications Match runs together with increases, its AMAL and jitter are increased rapidly. This is because the contention at the memory increases, which causes long memory access latency. When we apply MAE in the system, the AMAL is reduced significantly (the left-hand side in Fig. 8(f)). This is because the latency spent in memory is hidden by our MAE and the latency perceived by accelerator is reduced significantly. The right-hand side part of Fig. 8(f) shows the jitter perceived by Match. Again, we find that without MAE, the jitter increases dramatically when we increase the number of active applications. When MAE is used, the jitter is reduced significantly.

In short, when more applications run simultaneously, the benefit of our proposed template-based MAE is more prudent. The results shown above demonstrate that our proposed template-based MAE can significantly reduce execution time as well AMAL and jitter for accelerators, especially in the situation where multiple applicants are running concurrently. This in turn reduces the design complexity for each accelerator.

VI. Conclusion

In this paper, we observe that many accelerators have common memory access patterns. The current approach to implement the accelerator leads to high memory access latency and jitter. In order to address this issue, we proposed a template-based MAE. Our templates cover several common memory access patterns we observe for near future accelerators. Instead of each accelerator sending out requests directly to the memory controller to move data from memory to its local buffer, the accelerator now sends out simple prefetch command to the MAE and the MAE automatically fetches data from off-chip memory to on-chip prefetch buffer for accelerators. The MAE has a global view of requests from different accelerators as well as their demands. It prioritizes requests based on the feedback from prefetch buffers. Our simulation results show that the proposed template-based MAE is very effective in reducing AMAL and jitter, especially in the situation where multiple applications are running concurrently. This leads to reduced execution time for accelerators. It also reduces the design complexity for individual accelerator, thus is very effective for future SoC architectures.

Acknowledgement

The authors would like to thank Yoonseok Yang of Texas A&M University for providing us with AES traces for our simulation. We also wish to thank Seung Eun Lee and Yong Zhang of Intel for their helps with accelerator traces generation.

References

[1] Intel Atom Processor: http://www.intel.com/technology/atom/.
[2] ARM Cortex-A9 MPCore: http://www.arm.com/products/CPUs/ARMCortex-A9_MPCore.html
[3] http://www.spec.org/.
[4] T. Mowry, et al., "Tolerating latency through software-controlled data prefetching", *JPDC*, vol. 12, pp. 87-106, June, 1991.
[5] C. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors", *ISCA*, 2001, pp. 40-51.
[6] A. C. Klaiber, et al., "An architecture for software-controlled data prefetching", *ISCA*, pp. 43-53, Toronto, Canada, May 1991.
[7] A. Roth, et al., "Effective jump-pointer prefetching for linked data structures", *ISCA*, pp. 111-121, Atlanta, GA, May 1999.
[8] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", ISCA, pp. 363-373, 1990.
[9] J.-L. Baer, et al., "Effective hardware-based data prefetching for high-performance processors", *IEEE TC*, 44(5):609-623, 1995.
[10] C. F. Chen, et al., "Accurate and complexity-effective spatial pattern prediction", *HPCA*, pp. 276-287, 2004
[11 K. J. Nesbit, et al., "Data cache prefetching using a global history buffer", *HPCA*, pp. 96-105, 2004.
[12] I. Hur., et al., "Memory prefetching using adaptive stream detection", *Microarchitecture*, Washington, DC, pp. 397-408, 2006
[13] Kim, D., et al., "Data cache and direct memory access in programming mediaprocessors", *IEEE Micro* 21, pp. 33-42, 2001.
[14] D. Tang, et al., "DMA cache: using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance", *HPCA*, February, 2010.
[15] M. Dasygenis, et al., "A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck", *IEEE Trans. VLSIS*, pp. 279-291, March, 2006.
[16] S. Lee, et al., "Accelerating Mobile Augmented Reality on a Handheld Platform", *ICCD, October, 2009.*
[17] S. Kumar, et al., "Atomic vector operations on chip multiprocessors*", ISCA*, 2008, pp. 441-452.
[18] J. Moses, et al., "ASPEN: towards effective simulation of threads and engines in evolving platforms," *MASCOTS*, 2004.
[19] N. Agarwal, et al., "GARNET: a detailed on-chip network model inside a full-system simulator, *ISPASS,* Boston, Massachusetts, 2009.
[20] F. Liu, et al., "Understanding How Off-chip Memory Bandwidth Partitioning in Chip-Multiprocessors Affects System Performance", *HPCA*, Bangalore, India, Jan 2010.