Improving Scalability of CMPs with Dense ACCs Coverage

Nasibeh Teimouri, Hamed Tabkhi and Gunar Schirner
Department of Electrical and Computer Engineering
Northeastern University, Boston (MA), USA
Email: {nteimouri, tabkhi, schirner}@ece.neu.edu

***Abstract*—Utilizing Hardware Accelerators (ACCs) is a promising solution to improve performance and power efficiency of Chip Multi-Processors (CMPs). However, new challenges arise with the trend of shifting from few ACCs (with sparse ACCs coverage) to many ACCs (denser ACCs coverage) on a chip. The primary challenges are a lack of clear semantics in ACC communication as well as a processor-centric view for orchestrating the entire system.**

This paper opens a path toward efficient integration of many ACCs on a single chip. To this end, the paper at first identifies 4 major semantic aspects when two ACCs communicate with each other: data access model, data granularity, marshalling, and synchronization. Based on the identified semantics, the paper then proposes an efficient architecture solution, Transparent Self-Synchronizing (TSS), to realize the identified semantics in the underlying architecture. In principle, TSS proposes a shift from the current processor-centric view to a more equal, peer view between ACCs and the host processors. TSS minimizes the interaction with the host processor and reduces the volume of ACC-to-ACC communication traffic exposed to the system fabric. Our results using 8 streaming applications with a varying ACC coverage density demonstrate significant benefits of TSS, including a 3x speedup over the current ACC-based architectures.

I. Introduction

Accelerator-based CMPs (ACMPs) have emerged as one of the primary platforms for high-performance low-power computing. ACCs offer an efficient but fixed data-path for compute-intensive kernels, and host processor cores provide flexibility to execute remaining parts of applications. Streaming applications are one of the major application domains suitable for ACMPs.

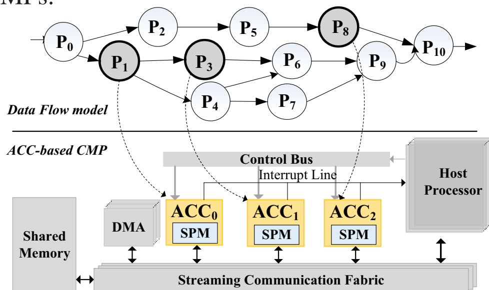


Figure 1: Example of SDF Mapped to ACC-Rich CMP

Fig. 1 exemplifies a streaming application (captured in a data flow model) mapped to an ACMP. Compute-intensive nodes, highlighted by dark color, are mapped to ACCs while remaining nodes execute on processor cores. In processor-centric ACMPs, the processor cores are responsible for orchestrating ACCs, even when two ACCs logically communicate (e.g. P1 to P3, Fig. 1). Major orchestration tasks are: (1) synchronizing, scheduling and configuration of ACCs, and (2) adjusting data type and granularity with respect to the specific functionality of ACCs.

With a trend toward specialization, the number of ACCs increases, as well as their coverage (i.e. an increasing number of adjacent ACCs). Major bottlenecks stem from a processor-centric view when the host processor is responsible for orchestrating many ACCs simultaneously. With a denser ACCs coverage, many ACCs communicate directly with each other (ACC-to-ACC communication) yet still require the processor for control and coordination. ACC-to-ACC communication is currently realized in two steps: ACC-to-processor and processor-to-ACC. Other major design bottlenecks include an immense communication demand for ACC communication, and very large on-chip memory for local Scratch-Pad Memories (SPMs). Overall, the limitations in shared resources (processor cycles, communication bandwidth, and memory capacity) significantly reduce ACCs utilization and overshadow the benefits of ACMPs. A study in [1] demonstrates that with 20 ACCs, the system performance drops to less than 10% of its peak.

Current studies that aim to mitigate the scalability limitations of ACMPs, [2], [3], [4], lack a holistic design view to identify and mitigate all major sources of bottleneck at the same time. There is a need to shift from the processor-centric view to an equal view between the host processor and ACCs with a clear semantic definition. Moreover, novel solutions are required to consider all aspects of design when it comes to integrating many customized ACCs into a system.

The goal of this paper is to open a path toward efficient integration of many ACCs on a chip. To this end, the contributions of this paper are: (1) a clear semantic definition for ACC communication, and (2) Transparent Self-Synchronizing architecture (TSS) as an extensible architecture template to efficiently integrate many ACCs. The paper defines 4 semantic aspects: data access model, data granularity, marshalling, and synchronization. TSS proposes an efficient architecture solution to realize the identified semantics in the underlying architecture with: (a) an autonomous control to synchronize the ACCs independently of the host processor, (b) fine-tuned local buffers per ACC to minimize the on-chip memory demand, and (c) an ACC-specific interconnect to hide the communication traffic of direct ACC-to-ACC from the system communication fabric. The TSS is mainly suitable for streaming applications where the kernels work in a producer-consumer fashion. Our results using 8 streaming applications demonstrate significant benefits

978-3-9815370-6-2/DATE16/©2016 EDAA

of TSS over the processor-centric ACMP. TSS results in up to 52x reduction in system fabric communication volume, reduces on-chip memory up to 12x and scheduling overhead by 5x.

The paper continues with Section II summarizing the related research work. Following that, Section III formulates the semantics of ACC communication. Section IV presents the details of the proposed TSS architecture. Section V evaluates the TSS benefits and compares it with Processor-Centric ACC-based CMPs, and finally Section VI concludes this paper.

II. Related Work

Some approaches aim to address scalability limitations of the processor-centric ACMPs [2], [5], [6], [7], [3], [4]. Among them, [8], [5] propose a two-level hierarchical interconnection to localize the ACC-to-ACC traffic. [6], [7], [3] propose run-time SPM optimization methods to reduce the on-chip memory demand for ACCs. [4], [9] propose Accelerator Block Composers (ABCs) to reduce synchronization overhead on the host processors.

In an orthogonal approach, streaming architectures such as MIT RAW processor [10], Complex dataflow machine [11], Coarse-Grain Reconfigurable Architectures (CGRAs) [12], and dynamically reconfigurable processors such as Modular SI in [13], provide programmable data-paths for efficient realization of streaming applications. However, these approaches mainly focus on computation rather than constructing/composing a system out of many heterogeneous ACCs.

Overall, we observe the lack of holistic view among the previous approaches. They either ignore the scalability issues or only focus on one design bottleneck. As an example, the ABC(s)s in CAMEL[4] can reduce the synchronization overhead on the host processor(s), but leaves behind the problem of large memory demands and immense ACC-to-ACC communication traffic. With the lack of holistic view, solving one design bottleneck may introduce additional overheads to other aspects. As one example, [6], [7], [3] save the memory at the cost of higher synchronization load on the host processor.

Recent studies ([14], [1]) hint about an equal non-discriminative view between processors and ACCs. [1] quantifies the scalability limitations of ACCs. PushPush in [14] suggests a mechanism enabling ACCs to directly call SW functions on host processors. Compared to previous approaches, this paper provides a holistic approach which includes both semantic definition as well as an efficient architecture template to mitigate the design bottlenecks.

This paper builds upon our discoveries [1], but differs in many aspects. While [1] mainly focused on quantifying the scalability limitations with less attention to architecture solution, this paper looks at the semantic definition and proposes an architecture solution to remove the limitations.

III. ACC Communication Semantic

To identify the semantic aspects, we investigate current processor-centric ACMPs. Fig. 2 illustrates the steps to start an ACC. In current ACMPs, each ACC has its own Scratch Pad Memory (SPM) to store input/output data of computation. The data is received/sent through the communication fabric to/from

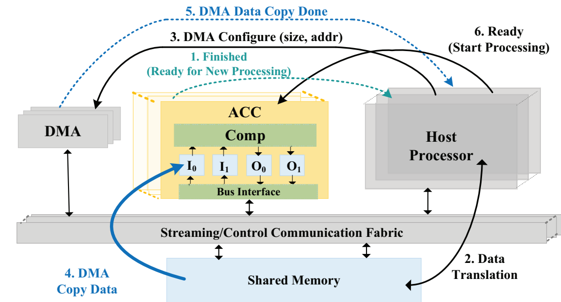


Figure 2: Steps for Communicating with ACCs in ACMP

memory via DMAs. To manage larger data sets, the streaming data is split into smaller chunks called jobs. At each point in time, processing inside an ACC is isolated to the current job. The job size is limited to the size of local SPMs.

Fig. 2 spells out 6 steps to start ACC processing: (1) The ACC notifies the host processor (signal *finished*) that it has finished the previous job. (2) The processor prepares the next job in shared memory (job size and granularity adjustment, and data type translation). (3) The host processor then configures a DMA to transfer the data into ACC's local SPM. (4) The DMA then transfers data through the communication fabric, and (5) notifies the processor about transfer completion. Finally, (6) the processor notifies the ACC (*ready* signal) to start processing. To overlap data transfer and processing, double buffering is employed – each ACC has separate input (I_0 and I_1) and output buffers (O_0 and O_1) illustrated in Fig. 2. While the ACC has exclusive access to one buffer set, the other buffer set is filled/emptied in parallel.

Overall, the formulation of ACC communication semantics reveals the inefficiency of Processor-Centric ACMPs. Looking at Fig. 2, actions (2) through (5) are purely dedicated for data preparation and transfer. The processor is always involved even when two ACCs logically communicate (ACC-to-ACC is split into ACC-to-processor and processor-to-ACC). Furthermore, shared memory and system communication fabric are occupied for ACC-to-ACC communication. The analysis highlights the importance of communication and synchronization. Overall, the paper defines 4 major aspects to communicate with ACCs:

Data Access Model defines how data can be accessed (i.e. to enable overlapping between processing and data transfer).

Synchronization defines when data needs to be accessed with respect to start and finish of computation.

Data Granularity defines the minimum amount of data required for processing. The granularity is dependent on the specific ACC's functionality.

Data Marshalling refers to input and output data representation. The communication format may differ from the processing format, which then requires a format conversion.

To translate these aspects to ACC-to-ACC communication, Fig. 3 shows the communication between a producer ACC (ACC_P) and a consumer ACC (ACC_C) with conceptual signals. The *Data Access Model* and *Synchronization* can be described as an extended FIFO with N elements. Each FIFO element holds the data for one job (*Data Granularity*). In

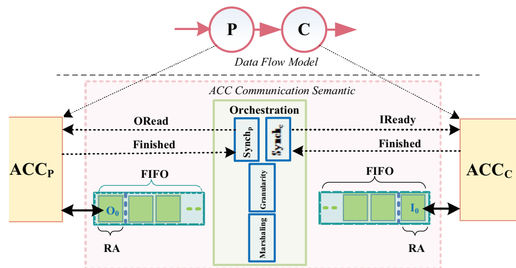


Figure 3: Producer Consumer ACC Communication

deviation from a standard FIFO, the FIFO's *head* allows random read and write access by enabling *IReady*. The ACC signals that it has finished consuming I_0 through *Finished*, upon which the FIFO can produce a new head element. Symmetrically, the same holds true for the output path. The *tail* element (O_0) has random access. *ORead* signals the availability of an empty tail, and *Finished* commits the tail element.

The output granularity of ACC_P may differ from the input granularity of ACC_C. In the result, granularity translation is required. Furthermore, if ACC_P's output data type differs from ACC_C's input data type, data type translation is needed. This can be considered similarly to marshalling in a networking protocol. ACC_P and ACC_C have to agree upon a common data format. Depending on the differences between data types, additional storage may be required to account for differently stridden accesses, or varying order of parameters in the data.

Any architecture has to address the semantic aspects above, but may choose a different implementation. Current processor-centric ACMPs realize these semantics in a very simple way. They often use a *Data Access Model* of double buffering, where the ACC has exclusive access to one set of local SPMs, while the other set is occupied for data transfer through DMAs. *Synchronization* is realized through the interrupt signals of DMAs, an MMR write to start ACC processing and another interrupt indicating processing finished. Less attention is placed on *Data Granularity* and *Data Marshalling* as they are implemented on the processor.

IV. TSS Architecture

This section introduces our proposed Transparent Self-Synchronizing (TSS) architecture. TSS focuses on the communication aspect of HW ACCs and constructing a system out of a heterogeneous set of HW ACCs.

A. ACC Semantic Support

The primary goal is to support all four semantic aspects independently of the host processor. Fig. 4 outlines the proposed architecture in the context of a single ACC. The ACC itself only realizes the processing. It expects exclusive read/write access to input/output buffers for processing (see *Data Access Model*, Section III). The input path is realized through an Input Control Management (ICM), and the output path through an Output Control Management (OCM). The internal structures of ICM and OCM are based on the identified semantics. ICM and OCM internally realize double buffers (for easier comparison with previous approaches), a marshalling unit, and a synchronization unit with data granularity management.

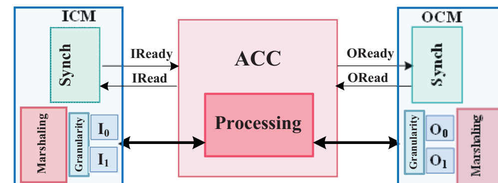


Figure 4: Single ACC Communication in TSS

The data marshalling unit is responsible for adjusting the data type with respect to the functionality of ACC. ACCs, depending on their functionality, may use a variety of data types. On the output side, the marshalling unit serializes a filled buffer in the output format of the ACC into a flat byte stream. The marshalling unit in the ICM receives the flat byte stream and creates a filled buffer in the ACC's desired input format. The marshalling unit splits/collects and reorders the data in bytes.

The granularity management controls the job size to minimize the memory demands for buffers. Job sizing is implemented in both ICM and OCM to fine tune job sizing. Compared to the Processor-Centric ACMPs which often have to operate with large job sizes to reduce the synchronization load on the host processor, ACCs in the TSS synchronize directly with each other. This allows much smaller job sizes, reducing memory size requirements, and synchronization load on the processor. Granularity management is implemented by a counter that matches the different stride accesses, or the varying order of data access between two ACCs.

To realize the data access model, ICM and OCM follow double buffering principles. They alternate exclusive access to the buffers to enable overlapping of data access and processing. However, in contrast to the processor-centric ACMPs, the size of buffer in ICM/OCM is much smaller – equal to the minimal streaming data (job) required for processing.

Furthermore, ICM and OCM contain local synchronization components to autonomously control the ACCs, eliminating unnecessary interactions with the host processor. It orders processing across the jobs with respect to double buffering semantic (for *IReady* and *ORead* signals). It receives the *finished* signal from the ACC and issues *IReady* and *ORead* signals to notify the ACC about the availability of data in local buffers.

One major benefit of ICM and OCM is the separation of communication and processing in ACCs. ICM and OCM principles are general enough to be applicable to all ACCs (assuming stream data processing). ACC-specific communication tasks are input/output data type (marshalling), and granularity adjustment. Separating processing from communication allows ACC designers to focus mainly on realizing the processing efficiently to eliminate the tedious and error-prone tasks of outside communication.

B. ACC-to-ACC Communication

Fig. 5 outlines our overall approach for direct ACC-to-ACC communication. The communication between two ACCs relies on the ICM and OCM introduced earlier. Between ACCs, a network/interconnect topology is required to provide

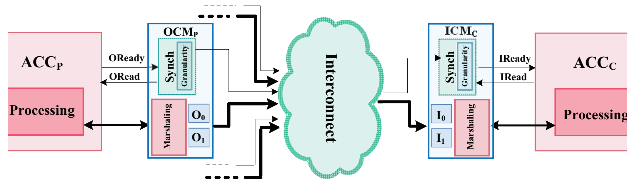


Figure 5: ACC-to-ACC Communication in TSS

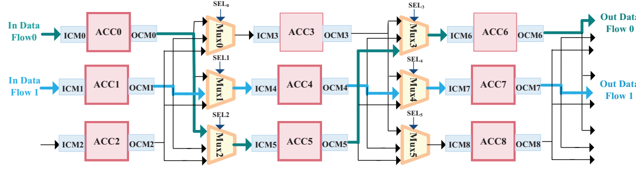


Figure 6: Parallel Sub flows

a communication path. Options range from direct links to multiplexers, and a bus-based communication to Network-on-Chip (NoC). The primary aim is to keep the traffic of direct ACC-to-ACC local, hidden from system communication fabric and shared memory.

In this paper, to separate between the architecture principles and topology of ACCs interconnection, a dedicated point-to-point connection between ACCs is desired. At the same time, some interconnect flexibility is needed to allow construction of varying data-paths of ACCs with support for self-feedback, backward, and forward connections. An NoC can be considered as the topology with full flexibility, but potentially has a high overhead. In contrast, a sparser connectivity that only allows connections between ACCs that result in meaningful compositions would be sufficient – not all combinations necessarily yield valid applications.

We choose a MUX-based topology to provide sparse configurable point-to-point connections between ACCs. Similar principles are applied in industrial products, e.g. Analog Devices Pipelined Vision Processor [15]. We consider a common data representation in the communication medium between ACCs. The data granularity management unit and marshalling unit, in either the producer side or consumer side, convert the output data type into the common format (size and type). The decentralized realization of data granularity management, data marshalling, and synchronization eliminate processor interaction in ACC-to-ACC communication. With this, ACCs become more independent of the host processor. Moreover, using the local synchronization unit per ACC, both producer and consumer ACCs are synchronized upon the availability of data; produced or being consumed. In this way, the processor-centric view shifts toward a peer-controlled, non-discriminatory setting.

The MUXed interconnect also enables spatial parallelism by concurrent execution of multiple chains of ACCs. Fig. 6 shows one TSS instance with two parallel chains (illustrated by blue and green colors). The distributed MUXed interconnect allows chaining/cascading of multiple ACCs inside TSS. Multiple ACCs directly communicate with each other and their traffic is hidden from the system fabric.

C. Gateway

The gateway interfaces the TSS to outside resources. Fig. 7 shows the gateway with SPMs, bus interfaces, an interrupt line to the processor, as well as a control/configuration unit with Memory-Mapped Registers (MMRs). The gateway also contains ICM and OCM units. ICMSs feed ACCs and OCMs collect results. The gateway architecture in Fig. 7 shows 3 ICMs and OCMs allowing 3 concurrent chains of ACCs.

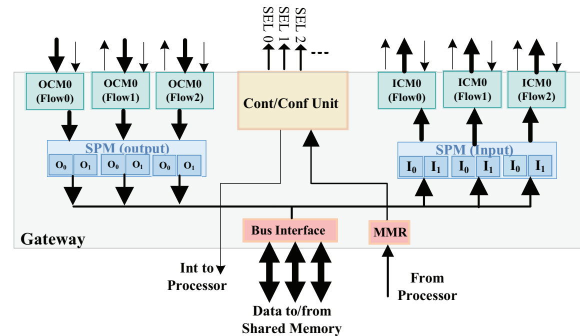


Figure 7: Gateway Architecture

The gateway's input SPM collects data (via DMA) from shared memory and then feeds it to ACCs. An output SPM collects the result of each flow to be then picked up by the DMA. The gateway performs granularity adjustment on input (through ICMs) and output (through OCMs). A larger outside job size can be broken up into many smaller internal jobs. The outside communication with the processor occurs very similarly as described for ACCs in current systems (see Fig. 2): DMA in, MMR to start, interrupt signaling done, and DMA out. Different from existing systems, the configuration MMRs also capture MUX configuration (for flow composition). Once at the beginning, the processor writes the configuration information into the MMRs and the gateway then configures the MUXs. For simplicity, we assume a constant configuration during application execution.

D. TSS Summary and System Integration

Fig. 8 shows the TSS integration to the host processor through the shared memory. The TSS is practically only visible through the gateway that receives data from shared memory and writes it back to the shared memory. With the abstraction through the gateway, each sequence of direct ACC-to-ACC communication, regardless of its length, appears as an individual ACC to the processor. The benefits of the TSS architecture template increase with denser ACC coverage containing longer stretches of contiguous processing. Conversely, without any ACC-to-ACC communication (no adjacent ACCs) it will behave very similar to existing architectures.

V. Experimental Results

This section evaluates the benefits of the proposed TSS compared to the processor-centric ACMP. For evaluation, we use Virtual Platforms (VPs) generated by the System-on-Chip Environment (SCE) [16] for both TSS and processor-centric ACMP. The general components of the VPs are: (1) an ARM9

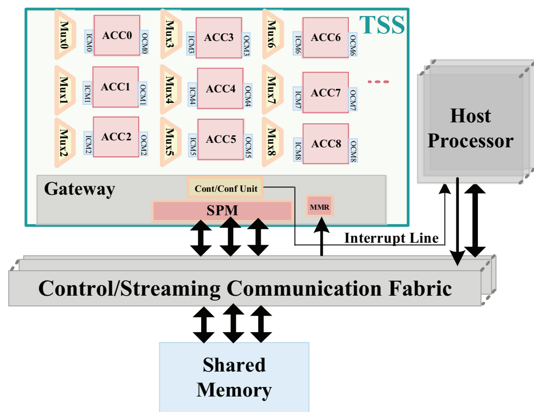


Figure 8: TSS System Integration

core (running uCOS/II) simulated by an OVP ISS working at 500 MHz clock frequency. (2) A multi-layer AMBA AHB (32 bit-width, 100MHz) with eight concurrent channels (4R and 4W). (3) four DMA modules. (4) a shared memory module with four access ports.

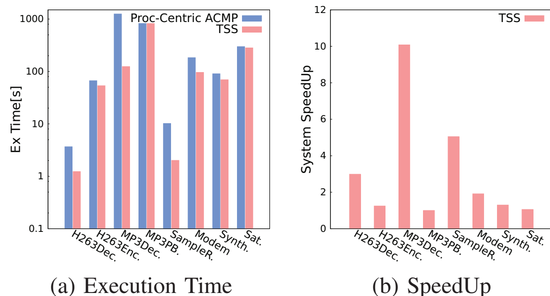
For the purpose of experiments, we use eight streaming applications captured in abstract data flow models (generated by SDF3 [17]). All are acyclic. Table I lists the properties of data flow models with the number of processing kernels and edges/links (*#of Nodes* and *#Edges*), range of operations per node (*Comp[Min:Max]*), bytes transferred per edge (*Comm[Min:Max]*), number of nodes mapped to ACCs (*#ACCs*) and logical direct ACC-to-ACC connections (*#ACC-to-ACC*). Please note that the decision for HW/SW mapping is beyond the scope of this paper. We follow general mapping strategies: Compute-intensive kernels are mapped to HW ACCs.

Table I: Applications Characteristics

App.	#Nodes	Comp[Min:Max]	#Edges	Comm[Min:Max]	#ACCs	#ACC-to-ACC
H263Dec.	4	[486:26018]	5	[64:38016]	4	3
H263Enc.	5	[6264:382419]	7	[384:38016]	4	4
MP3Dec.	14	[409:1865001]	17	[576:576]	8	8
MP3PB	4	[4:10000]	5	[4:4]	2	1
Sam.Rat.	5	[4:4]	6	[4:4]	5	4
Modem	16	[130:7000]	22	[4:4]	11	13
Synthetic	23	[100:89112]	24	[100:1000]	13	13
Satellite	22	[1000:1000]	29	[4:4]	11	10

A. Performance Evaluation

Fig. 9 presents the absolute execution time in logarithmic scale (Fig. 9a) as well as the relative speedup of TSS over the processor-centric ACMP across all applications (Fig. 9b). Overall, Fig. 9b shows 3x performance improvement in TSS



(a) Execution Time (b) SpeedUp

Figure 9: System Performance and Relative Speedup

over the processor-centric ACMPs on average. The major sources of performance improvement are minimizing the interaction with the host processor(s), removing the unnecessary data copies between shared memory and local SPMs, and hiding ACC-to-ACC traffic from system communication fabric. By increasing the number of direct ACC-to-ACC links/edges over the total number of links/edges, TSS delivers higher speedups.

To show more insights about TSS benefits, Fig. 10 compares TSS and the processor-centric ACMP to evaluate the load on the shared resources: memory, communication fabric, and processor. Fig. 10a compares the memory demand between TSS and the processor-centric ACMPs. We assume a fixed but fairly large maximum on-chip memory: 2 MB on-chip memory. On average, TSS only requires 14% of original memory. TSS self-synchronization efficiently allows processing of small jobs without incurring high synchronization overhead. This enables ACCs to work on their minimal job sizes resulting in smaller buffers per ACC. Fig. 10b compares communication volume exposed to the system communication fabric. TSS significantly reduces the system communication volume by hiding ACC-to-ACC communications (e.g. TSS implementation of *H263Encoder* results in more than 95% reduction in communication volume compared to the processor-centric ACMP).

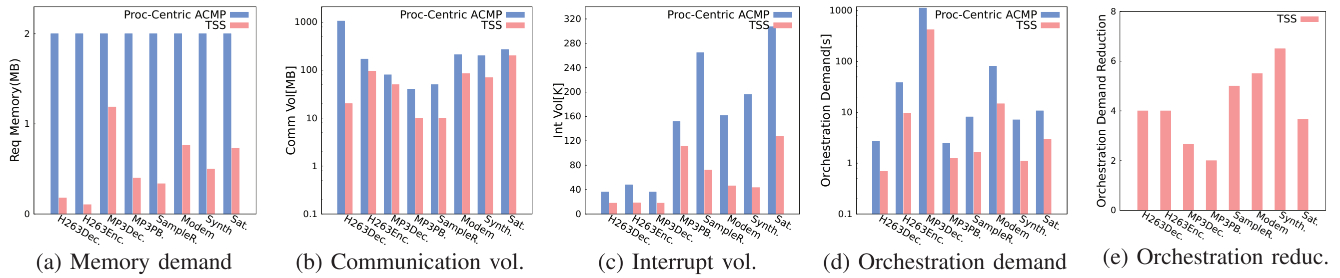
Fig. 10c shows the number of interrupts (i.e. synchronization requests) sent to the host processor. On average, interrupt volume on TSS is about 3x less than processor-centric ACMPs. As the number of direct ACC-to-ACC increases, the fewer synchronization requests will be sent to the host processor. Fig. 10d presents the execution time on the host processor dedicated for orchestration of ACCs (synchronization, granularity adjustment and marshaling). TSS significantly reduces orchestration time across all applications, 4.5x on average over the processor-centric ACMP (Fig. 10e illustrates the relative comparison).

B. Energy Consumption and Area Discussion

For energy estimation, we use reported numbers in the literature [18], [19] including: 14 pJ per 8 Bytes data transfer, 3.8 pJ for each Kilo operation in the ACCs (as well as ICMs/OCMs in TSS), 300mW power for ARM9 running at 500 MHz and 30mW static power per each 100KB of on-chip shared memory.

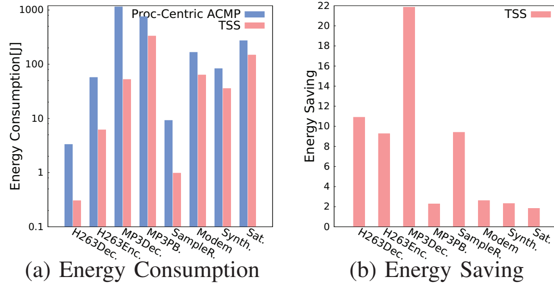
Fig. 11 shows both absolute energy consumption, and relative energy saving of TSS over the processor-centric ACMPs. Overall, the energy saving trend follows the speedup trend in Fig. 9b. As bottlenecks are relieved, less energy will be consumed. On average, TSS energy consumption is 8 times less than processor-centric ACMPs. The pronounced energy saving stems from the significant reduction of the load on host processor, volume of data transfer on the system communication fabric, and on-chip memory.

Since our work at this time is based on VP simulation, it is difficult to report and compare the area overhead. Nonetheless, the local buffers in ICMs and OCMs to store streaming data under processing are primary area consumers in TSS. With



(a) Memory demand (b) Communication vol. (c) Interrupt vol. (d) Orchestration demand (e) Orchestration reduc.

Figure 10: Demand on shared resources



(a) Energy Consumption (b) Energy Saving

Figure 11: Energy improvement

the TSS autonomous control, ACCs can work with smaller jobs (without creating excessive synchronization load on the processor) which allows for smaller local buffers compared to Processor-Centric ACMPs. Since the interconnect for direct ACC-to-ACC communication is MUX-based (and sparse), it also consumes little area only. The gateway contributes somewhat to the area (mainly due to its bus interface), but much less than a dedicated slave interface per ACC in processor-centric ACMPs. Overall, we do not anticipate an area overhead by TSS. Conversely, TSS leads to area saving due to reduction in on-chip memory (plotted in Fig. 10a).

VI. Conclusions

With increasing density in ACC coverage, formalizing ACC communication and defining supporting architectures becomes more critical. This paper defined 4 semantic aspects for communicating with an ACC: data granularity, marshalling, synchronization and data access model. This paper also proposed an architecture template, Transparent Self-Synchronizing (TSS), which realizes the semantic aspects and streamlines the integration of many ACCs. TSS shifts from the current processor-centric view toward a more equal peer view between ACCs and the processors. Our results using 8 streaming applications demonstrate that TSS improves performance by 3x, decreases communication volume on the communication fabric by 9x, reduces synchronization overhead by 3x and energy consumption by up to 22x.

Acknowledgment: This material is based upon work partially supported by the National Science Foundation under Award No. 1319501.

References

[1] N. Teimouri, H. Tabkhi, and G. Schirner, "Revisiting accelerator-rich CMPs: challenges and solutions," in *Design Automation Conference (DAC)*, 2015, pp. 84:1–84:6.

[2] P. Stillwell, V. Chadha, O. Tickoo, S. Zhang *et al.*, "Hippai: High performance portable accelerator interface for SoCs," in *International Conference on High Performance Computing (HiPC)*, 2009, pp. 109–118.

[3] M. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store framework for high-performance, low-power accelerator-based systems," *Computer Architecture Letters (CALs)*, vol. 9, no. 2, pp. 53–56, 2010.

[4] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 305–310.

[5] S. Tan, F. Qiao, B. Xia, H. Yang *et al.*, "A Functional Model of SystemC-Based MPEG-2 Decoder with Heterogeneous Multi-IP-Cores and Hybrid-Interconnections Architecture," in *Conference on Image and Signal Processing (CISP)*, 2009, pp. 1–5.

[6] E. Cota, P. Mantovani, M. Petracca, M. Casu *et al.*, "Accelerator Memory Reuse in the Dark Silicon Era," *Computer Architecture Letters (CALs)*, vol. 13, no. 1, pp. 9–12, 2014.

[7] J. Cong, M. A. Ghodrat, M. Gill, C. Liu *et al.*, "BiN: A buffer-in-NUCA Scheme for Accelerator-rich CMPs," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2012, pp. 225–230.

[8] C. Pham-Quoc, J. Heisswolf, S. Werner, Z. Al-Ars *et al.*, "Hybrid interconnect design for heterogeneous hardware accelerators," in *Design, Automation Test in Europe (DATE)*, 2013, pp. 843–846.

[9] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian *et al.*, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2012, pp. 379–384.

[10] M. Taylor, J. Kimand, J. Miller, D. Wentzlaff *et al.*, "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *Micro, IEEE*, vol. 22, no. 2, pp. 25–35, 2002.

[11] O. Pell and V. Averbukh, "Maximum Performance Computing with Dataflow Engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

[12] H. Park, Y. Park, and S. Mahlke, "Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications," in *Microarchitecture*. ACM, 2009, pp. 370–380.

[13] L. Bauer, M. Shafique, and J. Henkel, "A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor," in *Field Programmable Logic and Applications (FPL)*, 2008.

[14] D. Thomas, S. Fleming, G. Constantinides, and D. Ghica, "Transparent linking of compiled software and synthesized hardware," in *Design, Automation Test in Europe (DATE)*, 2015, pp. 1084–1089.

[15] R. Bushey, H. Tabkhi, and G. Schirner, "Flexible function-level acceleration of embedded vision applications using the Pipelined Vision Processor," in *Asilomar Conference on Signals, Systems and Computer (Asilomar SSC)*, 2013, pp. 1447–1452.

[16] R. Dömer, A. Gerstlauer, J. Peng, D. Shin *et al.*, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP J. Embedded Syst.*, 2008.

[17] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design (ACSD)*, 2006, pp. 276–278.

[18] S. Keckler, W. Dally, B. Khailany, M. Garland *et al.*, "GPUs and the Future of Parallel Computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, 2011.

[19] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis *et al.*, "Towards Energy-proportional Datacenter Memory with Mobile DRAM," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 37–48, 2012.