# AXR-CMP: Architecture Support in Accelerator-Rich CMPs


Jason Cong
UCLA, CS Department
cong@cs.ucla.edu

Mohammad Ali Ghodrat
UCLA, CS Department
ghodrat@cs.ucla.edu

Michael Gill
UCLA, CS Department
mgill@cs.ucla.edu

Chunyue Liu
UCLA, CS Department
liucy@cs.ucla.edu

Glenn Reinman
UCLA, CS Department
reinman@cs.ucla.edu

Yi Zou
UCLA, CS Department
zouyi@cs.ucla.edu



## ABSTRACT

To improve performance/power efficiency, we expect that future CMPs may use special-purpose accelerators extensively. This work discusses hardware architectural support for accelerator-rich CMPs. First, we introduce an efficient cache management scheme for accelerators to mitigate memory latency by overlapping data transfer with computation. Second, we present a hardware resource management scheme for accelerator sharing. This scheme supports sharing and arbitration of multiple cores for a common set of accelerators, and it uses a software-based priority mechanism to provide feedback to cores that indicates the wait time before acquiring a particular resource. Finally we propose architectural support that allows us to compose a larger *virtual* accelerator out of multiple smaller accelerators, and chain multiple accelerators together with minimal intervention of the requesting core. Experimental results show significant performance and energy improvement compared to approaches that use OS-based accelerator management, and achieve on the average 9X in performance (up to 40.17X) and 32X in energy efficiency (up to 90X) over a software implementation, with minimal hardware overhead.


## 1. INTRODUCTION

Power-efficiency has become one of the primary design goals in the many-core era. While ASIC/FPGA designs can provide orders of magnitude improvement in power efficiency over general-purpose processors, they lack reusability across different application domains, and significantly increase the overall design time and cost [19]. On the other hand, general-purpose designs can amortize the cost of their designs over many application domains, but can be 1,000 to 1,000,000 times less efficient in terms of performance/power ratio in some cases [19]. A recent industry trend to address this is the use of on-chip accelerators in many-core designs [12][20][13]. According to an ITRS prediction [2], this trend is expected to continue as accelerators become more common and present in greater numbers (close to 1500 by 2022). On-chip accelerators are application-specific or domain-specific implementations that provide power-efficient implementations of a particular functionality, and can range from simple tasks (i.e., a multiply accumulate operation) to tasks of more moderate complexity (i.e., an FFT or DCT) to even more complex tasks (i.e., complex encryption/decryption or video encoding/decoding algorithms). On-chip accelerators are combined with general-purpose cores in an effort to amortize the cost of the design across many application domains. Accelerators can capture the most commonly executed kernels of one or more application domains. They are relatively simple to design/optimize (compared to the entire application), and the general-purpose cores can be used to handle the rest of the application. A key issue in such accelerator-rich architectures is efficient management for sharing of accelerators among different cores and across different applications.

Accelerator-rich architectures also offer a good solution for overcoming the *utilization wall* as articulated in the recent study reported in [22]. It demonstrated that a 45nm chip filled with 64-bit operators would only have around 6.5% utilization (assuming a power budget of 80W). The remaining *un-utilizable* transistors are ideal candidates for accelerator implementations, as we do not expect all the accelerators to be used all the time. Moreover, once an accelerator is used, it provides a much higher performance/power efficiency due to its customized implementation, compared to the general-purpose cores.

We classify on-chip accelerators into two classes: 1) tightly coupled accelerators where the accelerator is a functional unit that is attached to a particular core (e.g., [13][11]) or implements extended instructions for that core [6][7]; and 2) loosely coupled accelerators (e.g., [3]) where the accelerator is a distinct entity attached to the network-on-chip (NoC), which can be shared among multiple cores. The key benefits of tightly coupling the accelerator are reduced latency to communicate to the accelerator and the ability to share certain blocks/functionalities with the processor core, such as a port to the NoC, TLB, and page table register. This paper focuses on the efficient use of loosely coupled accelerators, which have been much less studied. These accelerators are not tied to any particular core, and can potentially be shared among all cores on-chip – but this does require some form of arbitration and scheduling. Also, these accelerators need some efficient mechanism to handle address mapping, data streaming, and accelerator customization (e.g. via composition or chaining). We want to design an efficient architectural framework to support loosely coupled accelerators with low overhead in terms of both the latency to communicate with the accelerator and the mechanisms to handle arbitration, scheduling, address mapping, accelerator invocation, etc.

With these goals in mind, we propose an accelerator-rich CMP architecture framework, named AXR-CMP, with a low-overhead resource management scheme that (i) allows accelerators to be shared in flexible ways, (ii) is minimally invasive to core designs, and (iii) is friendly for application programs to use. Our paper provides the following contributions:

- An accelerator allocation protocol to avoid OS overhead in scheduling tasks to shared loosely coupled accelerators
- A DMA engine that facilitates data communication between the memory hierarchy and the accelerators, and between accelerators themselves to aid accelerator chaining and accelerator virtualization
- An approach to accelerator composability that allows accelerators to be pipelined together to overlap task latency and be composed together into virtual accelerators to collaboratively work on larger tasks
- A detailed analysis with RTL implementation of various hardware components introduced in AXR-CMP for efficient accelerator support

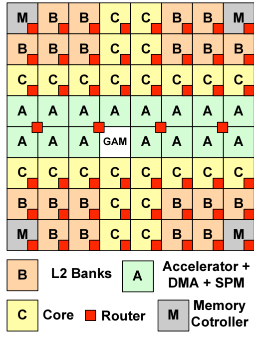


Figure 1: Overall architecture of AXR-CMP

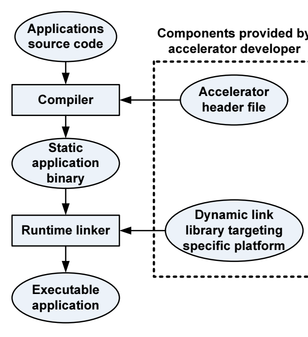


Figure 2: AXR-CMP development flow

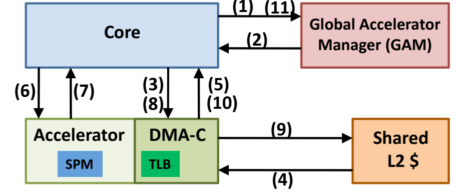


Figure 3: Detail of communication between core, accelerator, its DMA-C and the GAM in AXR-CMP

To the best of our knowledge, this is the first work with a detailed design for on-chip HW support for accelerator management. The rest of the paper is organized as follows: The architectural support for our proposed method is reviewed in Section 2. Section 2 also discusses microarchitecture design and provides the details of our architectural novelties. Sections 3 and 4 discuss our evaluation methodology and experimental results which support our proposed methods. Section 5 reviews some related work, and finally we conclude in Section 6.

## 2. ARCHITECTURE SUPPORT OF AXR-CMP

Figure 1 shows the overall architecture of AXR-CMP which is composed of cores, accelerators, the Global Accelerator Manager (GAM), shared L2 cache banks and shared NoC routers between multiple accelerators. All of the mentioned components are connected by the NoC. Accelerator nodes include a dedicated DMA-controller (DMA-C) and scratch-pad memory (SPM) for local storage and a small translation look-aside buffer (TLB) for virtual to physical translation. The newly added instructions to the ISA, together with their brief descriptions, are listed in Table 1. The development flow involved in using accelerators is presented in Figure 2. For each type of accelerator, one dynamic linked library (DLL) is provided. This DLL is specific to a target platform, and provides a mapping from accelerator calls to actual invocations of physical accelerators. Calls to accelerators have their implementations dynamically linked to application code. In this way, the application programmer is just calling the accelerators and the instructions listed in Table 1 would be called transparently through the DLLs.

**Table 1: AXR-CMP ISR**

| Instruction | Description |
|---|---|
| lcacc-req t | Request an accelerator of type *t* |
| lcacc-rsrv t, e | Reserve an accelerator of type *t* for an estimated *e* cycles of execution |
| lcacc-free id | Free accelerator *id* |
| lcacc-cmd id, f, addr | Execute a command of type *f* on accelerator *id* using parameters addressed by *addr* |

Figure 3 shows the communications among a core, the GAM, an accelerator, its DMA-C and the shared memory. The numbers on the arrows in Figure 3 show the steps taken when a core *C* uses an accelerator of type *X*. They are described below.

1. Core *C* requests an accelerator of type *X* from the GAM (*lcacc-req*).
2. GAM acknowledges (available now or later) or rejects the request (e.g., due to permission violation).
3. (a) If an accelerator *A* of type *X* is available now, core *C* asks DMA-C of *A* to transfer input data to SPM (*lcacc-cmd*).
   (b) If *A* is available later, core *C* waits and reserves it (using *lcacc-rsrv*) or continues on software path.
   (c) If it's rejected, core *C* continues on software path.
4. DMA-C starts data transfer from memory to SPM.
5. When data transfer finishes, DMA-C of *A* interrupts core *C*.
6. Core *C* starts accelerator *A* (*lcacc-cmd*).
7. When accelerator *A* finishes its job, it interrupts core *C*.
8. Core *C* asks DMA-C of *A* to transfer the output data from its SPM to memory (*lcacc-cmd*).
9. DMA-C of *A* starts data transfer from SPM to memory.
10. When DMA-C of *A* finishes it interrupts core *C*.
11. Core *C* frees the accelerator *A* by sending a free message to the GAM (*lcacc-free*).

The sample code shown in Figure 4 shows how to configure a DMA-C to copy a block of data with *size* bytes from the memory location with address *buffer* (lines 5-6) to SPM (step 3(a) above). Lines 2-4 initialize the *lpb_data_t* data structure (shown on the right-hand side of the code) for passing input parameters and lines 10-13 embed a SPARC assembly code which shows how to pass all this information to one of our newly introduced instructions *lcacc-cmd*. This code will be part of the DLL provided for an accelerator.

The next two subsections presents more details on how a core uses an accelerator and how the accelerators are being shared.

### 2.1 Using accelerators

#### 2.1.1 Core, accelerator and memory communication

For each accelerator, we add a DMA-C for transferring data between the shared L2 cache and the dedicated SPM. To overlap the data transfer with accelerator computation, the application divides the working data into a number of blocks, whose size is the minimum data size required by the accelerator to run. For different accelerators, this block size is different. The memory streaming scheme is demonstrated in Figure 5. Assume the blocks of the working set are B1, B2,..., Bn. The core will first ask the accelerator's DMA-C to fetch B1 (Figure 5 Step 1). When B1 is fetched, the core will start the accelerator. In the meantime, when the accelerator is working on B1, the core asks the DMA-C to transfer the subsequent block B2 (Figure 5 Step 2). When both the computation on B1 and communication of B2 are finished, the core asks the accelerator to work on B2 and asks the DMA-C to transfer the subsequent block B3 from the shared L2 cache to SPM and to transfer the output data O1 for block B1 back to the shared L2 cache (Figure 5 Step 3). It can be seen that, in this way, the necessary SPM size for the accelerator does not need to be the whole working data size. It needs a minimum of two times of the aforementioned block size plus some space to store the output data, so that the SPM is utilized as a streaming buffer (i.e., circular FIFO with two entries). The size of the SPM can also be larger than two blocks to achieve a deeper buffer (i.e., a circular FIFO with more than two entries).

```
0:void copy_from_cache_to_spm(void *buffer, int size, int accelerator_id) {
1:  #define CACHE_2_SP 1
2:  lpb_data_t prefetch_data;
3:  prefetch_data.nof_args = 2;
4:  prefetch_data->args = (void *)malloc
     (sizeof(void *)*prefetch_data.nof_args);
5:  prefetch_data->args[0]=(void *)buffer; // pointer in memory
6:  prefetch_data->args[1]=(void *)&size;  // size in memory
7:  accelerator_id = 0;
8:  function_id = CACHE_2_SP;
9:  asm {
10:     lduw g2, accelerator_id
11:     lduw g3, function_id
12:     lduw g4, prefetch_data
13:     lcacc-cmd  g2, g3, g4
14: }
15:}
```

```
typedef struct
{
    int nof_args;
    void **args;
    unsigned int status;
} lpb_data_t;
```

**Figure 4: DLL sample**

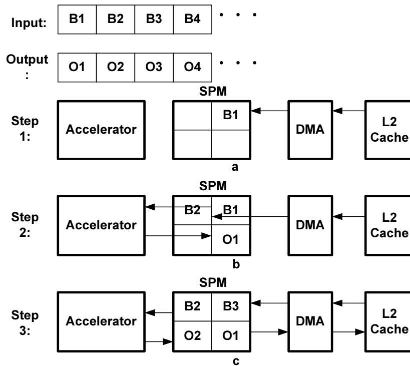


**Figure 5: Memory streaming**

### 2.1.2 Virtual address handling

Since the accelerator's DMA-C accesses the shared L2 cache with physical address while it is configured by the core with virtual address, a TLB is necessary to translate the virtual address to a physical address. We have considered two options for virtual address translation: 1) TLB on core only 2) a TLB on core and a TLB on accelerator. We have selected the second approach, since it gives simplicity to accelerator design, also makes the programmer interface between accelerator and the core simpler.

When a core is granted an accelerator, it first sends the starting address of the working data together with that address's TLB entry to the accelerator's DMA-C (using *lcacc-cmd*). Thus when the DMA-C begins to work, it only has one TLB entry. The DMA holds two pointers to carry out the following actions simultaneously:

- Pointer 1 is used to walk through the virtual addresses (in advance of actually executing anything) and to check the TLB for hits/misses. If there is a miss, it interrupts the core to get a translation by sending a message to the core with the accelerator ID and the missing page number. The core's interrupt handler searches the core's TLB table and returns the TLB entry corresponding to the missing virtual address. If the core's TLB misses too, OS intervention happens during the execution of this instruction to fetch the missing page number from low-level memory. After resuming from the page miss exception, this instruction is re-executed and returns the required table entry.

- Pointer 2 is used to walk through the virtual addresses to perform the data transfer with the help of the TLB. At the time of a TLB miss, it will stop and wait for Pointer 1.

The reason to hold two pointers is to perform the TLB filling and the data transfer in parallel. Since the TLB check is always faster than data transfer, we expect that the TLB miss handling latency will be overlapped with the data transfer latency. All the aforementioned messages sent from the core to accelerator, including the starting address, initial TLB entry and the following TLB entry based on the accelerator request, are issued by the *lcacc-cmd*.

## 2.2 Sharing accelerators

When the accelerators are shared among all the on-chip cores, it is possible that there are several cores competing for the same type of accelerator. In this situation, some of these cores may continue with executing the non-accelerated version if the wait time is too long for any potential gains. In this paper we propose a sharing and management scheme which can dynamically determine whether the core should wait to use an accelerator or choose a software path, based on an estimated waiting time. This proposed sharing and management strategy is performed by the GAM. The GAM tracks: 1) the types of available accelerators; 2) the number of accelerators of each type; 3) the jobs currently running or waiting to run on accelerators and their starting time and estimated execution time; 4) the waiting list for each type of accelerators and the estimated run time for each job in the waiting list. All of this data can dynamically change at run-time.

### 2.2.1 Accelerator execution-time estimation by core

The execution time of a certain job on an accelerator is data-dependent and is estimated by the requesting core. For most of our examples, we found that a linear regression was sufficient to model execution time. The regression model is provided by the accelerator DLL mentioned in Figure 2. For instance, Figure 6 shows the regression analysis for estimating the execution time of 3DES as a function of its input size. Figure 6 shows two graphs: *the measured time* which is the real simulation data, and the *linear* line ($time = 454.11 \times Size + 4e6$) which is the linear regression graph for the given measurements. This line is used in the DLL to estimate the execution time when reserving the accelerator, which is passed as the *e* parameter in the *"lcacc-rsrv t, e"* instruction.

### 2.2.2 Wait-time estimation by GAM

The estimation of waiting time can be done by dividing the total execution time of the jobs in the waiting list of an accelerator by the number of accelerators of that type. This provides a less accurate waiting time, but is practical for hardware implementation.

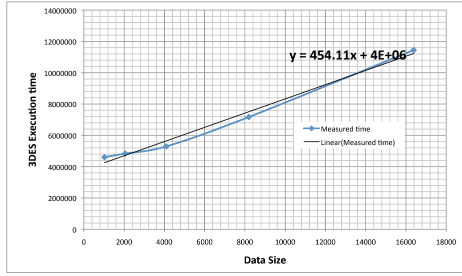


**Figure 6: 3DES execution time regression analysis**

To provide a much more accurate estimation, we develop an on-line first-come-first-serve (FCFS) scheduling estimation method shown in Algorithm 1. This algorithm finds the time at which each accelerator will be available, and then it returns the minimum available time. It does this by first initializing the availability time of each accelerator to the remaining time for the current job (lines 8-10) and then repeatedly updating the accelerator availability time by distributing the waiting jobs to the accelerator that will be available sooner (lines 11-14).

**Algorithm 1** FCFS scheduling based waiting time estimation

```
1:  T(j) → estimated run time of a job j on an accelerator
2:  RT(i) → the remaining time of the current job on an accelerator i
3:  M → number of jobs in the waiting list
4:  N → number of accelerators
5:  if M < N then
6:      waiting_time = (M+1)th minimum RT(i) (i ∈ [1, N])
7:  else
8:      for all i from 1 to N do
9:          Available_time_Acc(i) = RT(i)
10:     end for
11:     for all j from 1 to M do
12:         Let Available_time_Acc(i) be the minimum one for i ∈ [1, N]
13:         Available_time_Acc(i) += T(j)
14:     end for
15:     Waiting_time = the min of Available_time_Acc(i) (i ∈ [1, N])
16: end if
```

Figure 7 shows an example of applying Algorithm 1. It shows three jobs running on the accelerators, with the remaining times of 2, 3 and 4 and four jobs in the queue with their estimated execution time shown in the queue. If a new accelerator request arrives, the simple solution gives us an estimated waiting time of 14 and the FCFS-based method gives us an estimated waiting time of 8 (for *Acc2*).

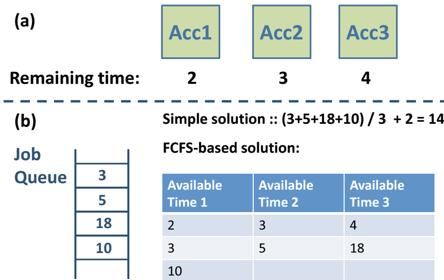


**Figure 7: Example of wait time estimation in the GAM**

## 2.3 Accelerator chaining and composition

In an accelerator-rich platform, there are many cases when the output of one accelerator feeds the input of another accelerator (like many streaming applications). In a traditional system, these two accelerators communicate through system memory; i.e., the controlling core reads the output of the first accelerator from its SPM to L2 and then writes it back to the second accelerator's SPM. To remove this inefficiency, two DMA-Cs can communicate and the source DMA-C can send the content of its SPM to another DMA-C to be written in its SPM. Figure 8 shows this scheme, in which *DMA-C1* is programmed as the source and *DMA-C2* is programmed as the destination by the core.

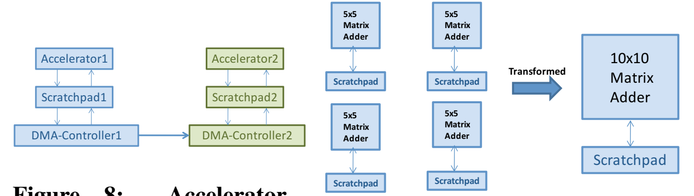


**Figure 8: Accelerator chaining using DMA-C**

**Figure 9: Accelerator composition technique**

By using the accelerator chaining technique we can create the virtual feeling of having a larger accelerator for one application (accelerator composition). Figure 9 is an example of how to do this for composing four smaller matrix engines to create a double-size matrix engine.

# 3. EVALUATION METHODOLOGY

We use Simics+GEMS for our baseline simulation infrastructure. Simics is a whole-system functional simulator [16]. Wisconsin's GEMS [17] is a collection of Simics modules which add the timing information for CPU (Opal module), memory hierarchy (Ruby module), and interconnection network (Garnet). We have made significant modifications to SIMICS/GEMS and have added several Simics modules to model our design. Table 2 shows the Simics/GEMS configuration that we used in our simulations.

**Table 2: Simics/GEMS configuration**

| | |
|---|---|
| CPU | Ultra-SPARC III-i |
| Number of cores | 1, 2, 4, 8, 16 |
| Coherence protocol | MESI_SCMP_bankdirectory |
| L1 cache | 32 KB, 4 way set-associative |
| L2 cache | 512 KB, 8-way set-associative |
| Network topology | Mesh |

The modules that were added are DMA-C and the GAM plus the computation engines for the targeted accelerators. Both DMA-C and the GAM are independent of the accelerator type and can be reused for any type of accelerator. We modeled the DMA-C and the GAM in Verilog to get an estimate of the area and clock speed. Table 3 shows the synthesis results for both of these modules in TSMC 65nm technology.

**Table 3: HW overhead for DMA-C and the GAM ($1cm^2$ chip)**

| | Clock speed | Area ($\mu m^2$) | Power (mW) |
|---|---|---|---|
| Global Accelerator manager (GAM) | 2 ns | 12270 (0.01%) | 2.64 |
| DMA-C | 2 ns | 10071 (0.01%) | 0.59 |

The AutoPilot [5] behavioral synthesis tool is used to synthesize the C modules into FPGA or ASIC. Then the estimated clock frequency and latency are back-annotated to our Simics modules. In this way, the timing for the Simics modules can be simulated.

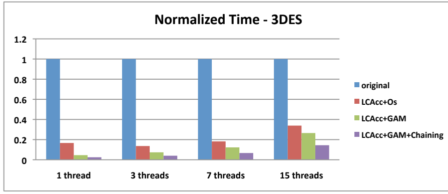


**Figure 10:** **3DES timing results**

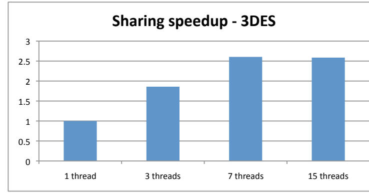


**Figure 11:** **3DES sharing speedup**

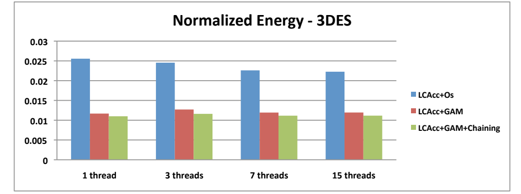


**Figure 12:** **3DES energy results**

Three sample benchmarks, namely MPEG4, 3DES, and Rician Denoise are chosen to evaluate the system functionality and performance. MPEG4 implements the compression codec for visual data. We implemented two functions in MPEG4 in hardware as accelerators: the motion compensation and IDCT. 3DES is a triple data encryption algorithm which applies the DES algorithm three times with three different keys on a block. Finally, Rician Denoise is a widely used benchmark to remove noise from medical images. We chose MPEG4 because it is a good candidate for showing our accelerator sharing scheme, and it is often needed to support multiple video streams on a single server chip. The reason to choose 3DES is for showing our accelerator chaining scheme on it, and it is quite common that multiple applications need data encryption/decryption. The Rician Denoise was selected because it is widely used in the medical imaging domain and because of its 3D input/output data ordering in memory, the linear design of DMA-C is inefficient and shows the need for a domain-specific design for the accelerator-rich platform.

In each case we implemented a multi-threaded version of the application so that we can observe the sharing of accelerators between multiple threads. The applications are re-written to overlap the communication and computation to gain the maximum achievable speedup.

For computing energy we used the power reports from AutoPilot for the accelerators and McPAT [15] for measuring CPU power. Energy here is the accumulated energy of accelerator and processors. We assume that each processor will be in idle mode with power gating when waiting for an accelerator, or when it finishes its job and is waiting for other cores to finish.

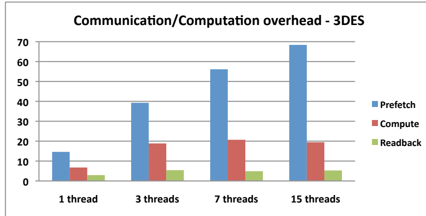


**Figure 13:** **Percentage of communication and computation for 3DES (no chaining)**

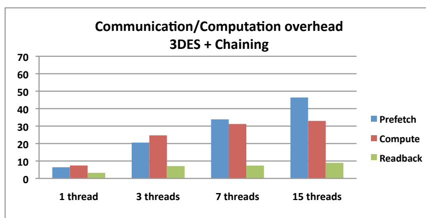


**Figure 14:** **Percentage of communication and computation for 3DES (with chaining)**

## 3.1 Evaluation schemes

Our evaluation schemes are:

- SW-Only (Original): The baseline for the experiments is the execution of these multi-threaded benchmarks on multiprocessors (one thread per processor) without any accelerator.
- Accelerator + OS management (LCAcc+OS): This is a system which has the accelerator, but the accelerator management tasks are being handled by OS drivers.
- Accelerator + HW management (LCAcc+GAM): This is a system which has the accelerator together with HW accelerator management and the support for virtual to physical address translation exposed to application.
- Accelerator + chaining (LCAcc+GAM+Chaining): This has everything "Accelerator + HW management" scheme has plus the ability to stream data from the scratch-pad of one accelerator to the scratch-pad of another accelerator.

# 4. EXPERIMENTAL RESULTS

In this section we present the result of applying our proposed techniques to three sample benchmarks. One major focus of our experiments is how well we can share an accelerator between N cores. In the worst case, we should get a slowdown of N times (gain of using one accelerator divided by N) compared to the case in which we have one accelerator for each core. But as we see in our results, this is not the case. By applying our scheduling algorithm we achieved much better results (sharing speedup). In other words, this is a measure of how close we are to the performance of one accelerator per core.

## 4.1 Case study – 3DES

3DES is an example of an application that benefits significantly from the memory streaming scheme that we have developed (as it is clear from our results). The DES engine was synthesized using the AutoPilot synthesis tool. Table 4 shows the synthesis results of the area and timing in TSMC 65nm technology. The clock speed has been back-annotated into our 3DES Simics module, and the whole system has been simulated in our modified Simics-GEMS platform.

**Table 4: Synthesis results**

| | 3DES | Motion comp. | IDCT | Rician Denoise |
|---|---|---|---|---|
| Clock(ns) | 1.44 | 1.14 | 1.65 | 4 |
| Cell count | 22091 | 3104 | 19243 | 7346 |
| Cell area $\mu m^2$ | 66641 | 17851 | 58878 | 34854 |
| Area % | 0.06 | 0.02 | 0.06 | 0.03 |

### *4.1.1 Energy and performance results*

Figure 10 shows the results of the normalized timing simulation for 3DES. The results show that we are getting a maximum of 40.2X

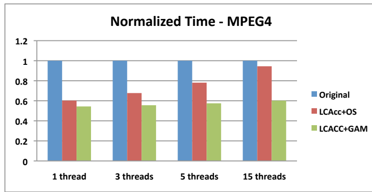


**Figure 15:** MPEG4 timing results

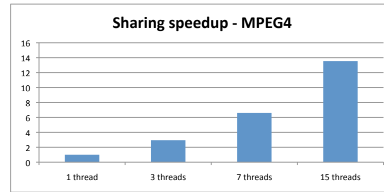


**Figure 16:** MPEG4 sharing speedup

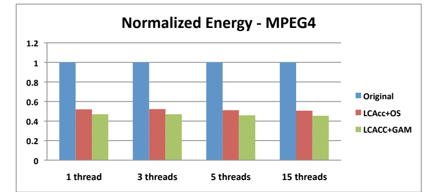


**Figure 17:** MPEG4 energy results

speedup over software-only solutions. Also results show a significant speedup for the HW managed cases over the OS managed scheme: 6.7X speedup when using accelerator chaining and 3.64X speedup without using accelerator chaining. Figure 11 shows the speedup of sharing one accelerator between 3, 7, and 15 threads compared to the worst case of sharing an accelerator between 3, 7 and 15 threads. As can be seen for the case of 15 threads, we can get 2.5X speedup compared to the worst case of sharing one accelerator between 15 threads.

Figure 12 shows the normalized energy consumption results for each of the three mentioned schemes over the original case, considering a 2 GHz CPU. For 3DES when using chaining, we obtained an almost 90X energy saving compared to software-only for all the cases. The reason for this constant energy saving is that the cores are shut off while waiting for an accelerator.

#### *4.1.2 Effect of chaining*

Figure 13 shows the breakdown of communication and computation for 3DES. The three tasks shown are all executed in parallel. It is clear 3DES is a very communication-bounded application. As shown, we have a significant decrease in both prefetching and readback time when we add chaining to our system (Figure 14). On average, we could get a 1.8X speedup over the case in which we use the accelerators and the GAM only.

### 4.2 Case study – MPEG4

MPEG4 is a sample application which shows great benefits from accelerator sharing, as will be clear from our results. Here, we assume a scenario where multiple video streams are being viewed, and each one needs an MPEG4 decoder. Here, we have synthesized two highly executed functions in the application, namely motion compensation and IDCT. Table 4 shows the synthesis results (area and clock speed) for both of these engines using the AutoPilot synthesis tool on TSMC 65nm ASIC technology.

Figure 15 shows the normalized timing simulation results for MPEG4 on one frame for the first three schemes mentioned in Section 3. The speedup (compared to SW-only) for 1, 3, 7 and 15 threads is almost constant, varying between 1.85X to 1.65X. This is different from 3DES which shows a decreasing speedup when increasing the number of threads (decreased from 40X to 7X). Here, because of the low utilization of the motion compensation and IDCT engines, when increasing the number of threads, we didn't see any decrease in speedup. This can also be seen in speedup for MPEG4 with accelerator sharing. Figure 16 illustrates the benefit of sharing by showing how much speedup we get by sharing the accelerators between 3, 7, and 15 threads. As can be seen, MPEG4 again displays a very good case for accelerator sharing because of the low utilization of the motion compensation and IDCT. Figure 17 shows the energy consumption results for each of the three mentioned schemes. Like performance, normalized energy is almost constant for all the cases (2.1X to 2.2X). We obtained a maximum of 36% improvement (1.56X) over the OS-based approach.

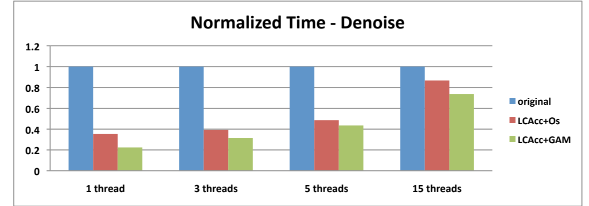


**Figure 18:** Denoise timing results

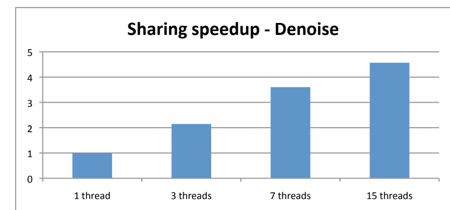


**Figure 19:** Denoise sharing speedup

### 4.3 Case study – Denoise

Three-dimensional denoise is an example of those applications which benefit from customized design for DMA-C. This application accesses the data in a three dimensional scheme. So the linear scheme for DMA-C does not work well for this application. Because of this we added a new mode of operation to our DMA-C so it can copy between the cache and accelerator scratchpad in an n-dimensional based tile. Table 4 shows the result of synthesis of the denoise module on TSMC 65nm technology.

Figure 18 shows the normalized timing simulation results and Figure 20 shows the normalized energy simulation results for Rician Denoise. It shows an almost 4X speedup for the case where we use an accelerator together with HW management techniques. Normalized energy varies between 8.5X to 11X. Again because of sharing one accelerator between N threads, we observed a slowdown compared to the one thread case, but this slowdown is not N (Figure 19), as Figure 19 shows we get a maximum speedup of 4.5X compared to the worst case of sharing an accelerator between 15 threads.

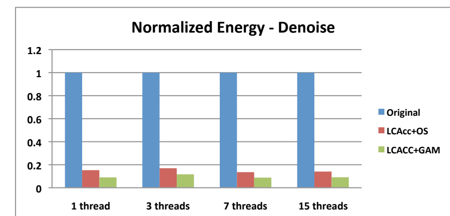


**Figure 20:** Denoise energy results

## 5. RELATED WORK

There is a large amount of work that implements an application-specific coprocessor or accelerator through either ASIC or programmable fabric like FPGA (e.g. [4] [8]). This work mostly considers a single accelerator dedicated to a single application. Commercial vendors, such as Convey [1] and Nallatech [3], manufacture computers that target reconfigurable computing in which customized accelerators are off-chip from the processors. Although this study also focuses on loosely coupled accelerators, we target future CMP architecture with on-chip integration of multiple kinds of accelerators in the "many-core, many-accelerators" setting. Some previous work considered on-chip integration of accelerators. Garp [10], UltraSPARC T2 [13], Intel's Larrabee [20] and IBM's WSP processor [12] are examples of this. Most of these platforms (except WSP) are tightly coupled with processor cores (or core-clusters). A thread on one processor can not use the accelerator coupled with another processor. Our paper focuses on loosely coupled accelerators in a way where accelerators can be shared between multiple cores. OS support for accelerator sharing and scheduling is presented in [9]. In contrast, we focus on hardware support for accelerator management. To the best of our knowledge, this is the first work that addresses this issue. EXOCHI [23] presents an architecture (exoskeleton sequencer) and a programming environment (C for heterogenous integration) for a heterogenous multi-core multi-threaded system. Like our work, the authors focused on a heterogenous non-uniform ISA. But unlike our work, their work assumed a software approach for accelerator management. HiPPAI [21], like our work, eliminates system overhead involved in accessing accelerators, but they do it using a software layer (portable accelerator interface), unlike ours that advocates a hardware approach. Please note that we also consider a separate software path, if the accelerator is not available.

Hardware support for efficient and flexible task scheduling is presented in Carbon [14] and ADM [18]. This work considered software tasks running on multi-core processors and did not consider our accelerator-centric environment.

## 6. CONCLUSION AND FUTURE WORK

We have discussed hardware architectural support for accelerator-rich CMPs. First, we presented a hardware resource management scheme for accelerator sharing and arbitration of multiple requesting cores for a common set of accelerators. Second, we introduced an efficient cache management scheme for accelerators to mitigate memory latency by overlapping data transfer with computation. Finally we proposed a mechanism that allows us to compose a larger virtual accelerator out of multiple smaller accelerators, and pipeline multiple accelerators together with minimal involvement of the requesting core. Our experimental results showed significant performance (up to 6.7X for 3DES) and energy improvements (more than 2.5X for 3DES) compared to approaches using OS-based accelerator management, with minimal hardware overhead. We also showed a significant performance benefit by efficiently sharing accelerator resources between multiple cores (up to 14X for MPEG4 on 15 cores).

## 7. ACKNOWLEDGEMENTS

This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127 and also by GSRC under contract 2009-TJ-1984.

## 8. REFERENCES

[1] *Convey computer*, http://conveycomputer.com/.

[2] *ITRS 2007 system drivers*, http://www.itrs.net/.

[3] *Nallatech FSB - development systems*, http://www.nallatech.com/.

[4] Dimitris Bouris et al., *Fast and efficient FPGA-based feature detection employing the SURF algorithm*, FCCM '10, 2010, pp. 3–10.

[5] J. Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang, *Platform-based behavior-level and system-level synthesis*, SOCC '06, 2006, pp. 199 –202.

[6] Jason Cong et al., *Instruction set extension with shadow registers for configurable processors*, FPGA '05, 2005, pp. 99–106.

[7] Jason Cong, Guoling Han, and Zhiru Zhang, *Architecture and compiler optimizations for data bandwidth improvement in configurable processors*, IEEE Trans. Very Large Scale Integr. Syst. **14** (2006), 986–997.

[8] Jason Cong and Yi Zou, *FPGA-based hardware acceleration of lithographic aerial image simulation*, ACM Trans. Reconfigurable Technol. Syst. **2** (2009), 17:1–17:29.

[9] P. Garcia and K. Compton, *Kernel sharing on reconfigurable multiprocessor systems*, FPT 2008, 2008, pp. 225 –232.

[10] J.R. Hauser and J. Wawrzynek, *Garp: a MIPS processor with a reconfigurable coprocessor*, FCCM'97, 1997, pp. 12 –21.

[11] Weirong Jiang and Viktor K. Prasanna, *Large-scale wire-speed packet classification on FPGAs*, FPGA '09, 2009, pp. 219–228.

[12] C. Johnson et al., *A wire-speed power$^{TM}$processor: 2.3ghz 45nm SOI with 16 cores and 64 threads*, ISSCC'10, 2010, pp. 104 –105.

[13] Tim Johnson and Umesh Nawathe, *An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2)*, ISPD '07, 2007, pp. 2–2.

[14] Sanjeev Kumar et al., *Carbon: architectural support for fine-grained parallelism on chip multiprocessors*, ISCA '07, pp. 162–173.

[15] Sheng Li et al., *McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures*, MICRO 42, 2009, pp. 469–480.

[16] Peter S. Magnusson et al., *Simics: A full system simulation platform*, Computer **35** (2002), 50–58.

[17] Milo M. K. Martin et al., *Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset*, SIGARCH Comput. Archit. News **33** (2005), 92–99.

[18] Daniel Sanchez et al., *Flexible architectural support for fine-grain scheduling*, ASPLOS '10, 2010, pp. 311–322.

[19] Patrick Schaumont and Ingrid Verbauwhede, *Domain-specific codesign for embedded security*, Computer **36** (2003), 68–74.

[20] Larry Seiler et al., *Larrabee: A many-core x86 architecture for visual computing*, IEEE Micro **29** (2009), 10–21.

[21] P.M. Stillwell et al., *HiPPAI: High performance portable accelerator interface for SoCs*, HiPC 2009, 2009, pp. 109–118.

[22] Ganesh Venkatesh et al., *Conservation cores: reducing the energy of mature computations*, ASPLOS '10, 2010, pp. 205–218.

[23] Perry H. Wang et al., *EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system*, PLDI '07, 2007, pp. 156–166.