Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip

Paolo Mantovani Emilio G. Cota
Christian Pilato Giuseppe Di Guglielmo Luca P. Carloni

Department of Computer Science
Columbia University - New York, NY, USA
{paolo,cota,pilato,giuseppe,luca}@cs.columbia.edu

ABSTRACT

Local memory is a key factor for the performance of accelerators in SoCs. Despite technology scaling, the gap between on-chip storage and memory footprint of embedded applications keeps widening. We present a solution to preserve the speedup of accelerators when scaling from small to large data sets. Combining specialized DMA and address translation with a software layer in Linux, our design is transparent to user applications and broadly applicable to any class of SoCs hosting high-throughput accelerators. We demonstrate the robustness of our design across many heterogeneous workload scenarios and memory allocation policies with FPGA-based SoC prototypes featuring twelve concurrent accelerators accessing up to 768MB out of 1GB-addressable DRAM.

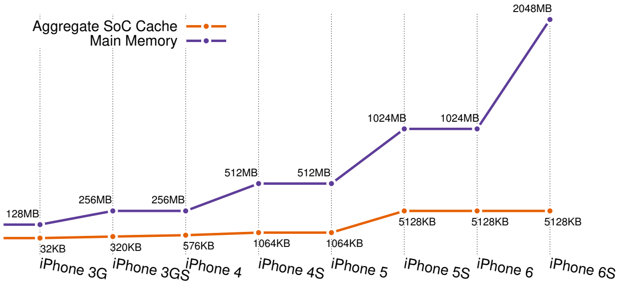


Figure 1: The growing gap between the aggregate size of the SoC on-chip caches and the main-memory size, across seven years of Apple iPhone products.

1. INTRODUCTION

The end of Dennard's constant-field scaling has led designers towards *heterogeneous system-on-chip (SoC) architectures* that exploit the large number of available transistors to incorporate a variety of customized *hardware accelerators* along with the processor cores [3]. To achieve energy-efficient high performance in embedded applications, both academia and industry have developed many different classes of accelerators and accelerator-rich architectures [5, 7, 8, 13, 18, 24, 28, 31]. A recent analysis of die photos of three generations of Apple SoCs, which empower the iPhone product line, shows that more than half of the chip area is consistently dedicated to application-specific accelerators [29].

These SoCs integrate *high-throughput loosely-coupled* accelerators [10] to implement complete application kernels, such as video encoding [23]. Each of these accelerators leverages a dedicated, highly-customized, *Private Local Memory (PLM)* and fetches data from DRAM through DMA. The PLM is key to achieving high data-processing throughput: by integrating many independent SRAM banks whose ports can sustain multiple memory operations per cycle, it enables concurrent accesses from both the highly-parallelized logic of the accelerator datapath and the DMA interface to main memory. Recent studies confirm the importance of the PLM, which occupies 40 to 90% of the accelerator area [10, 21] and contributes, together with the processors' caches, to the growing fraction of chip area dedicated to memory.

However, despite this trend, the data-set sizes of embedded applications are increasing much faster than the available on-chip memory. As an example, Fig. 1 illustrates the growing gap between the aggregate size of the SoC on-chip caches (accounting for L1 and L2 caches, and, starting with iPhone 5s, a 4MB L3 cache) and DRAM size, across seven years of Apple iPhone generations. Relative to the first product generation, the difference between the two sizes has grown by a factor of 16×, to reach almost 2GB[1]. The growth in DRAM size reflects the need for supporting applications with increasingly large footprints, which pose new challenges for high-throughput accelerators.

A loosely-coupled accelerator has a twofold nature: while it is similar to an on-board peripheral, in that the processor core can offload a specific task to it, the accelerator does not have a large and private storage system (e.g. a dedicated off-chip memory), and therefore shares the external memory with the processor core. At the same time, the accelerator's computation modules are unaware of the physical memory allocation, which can be even on multiple physically-separated DRAM banks [34]. Thus, an accelerator can be likened to a software thread, where physical memory is abstracted by virtual memory and multiple levels of caches. In an accelerator, the PLM gives the illusion of a contiguous address space, allowing the accelerator to perform concurrent random accesses on data structures. This contiguous address space, however, is limited by the size of the PLM, and processing large data sets necessarily involves multiple data transfers between DRAM and PLM.

[1] Admittedly, the cache numbers do not include the aggregate sizes of the PLMs of the Apple SoC accelerators, which are likely to be large but are not publicly known. Still, even if we assume that their contributions could double or triple the reported figures, the on-chip memory sizes would remain very small compared to DRAM.

CASES '16, October 01-07 2016, Pittsburgh, PA, USA

ISBN 978-1-4503-4482-1/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2968455.2968509

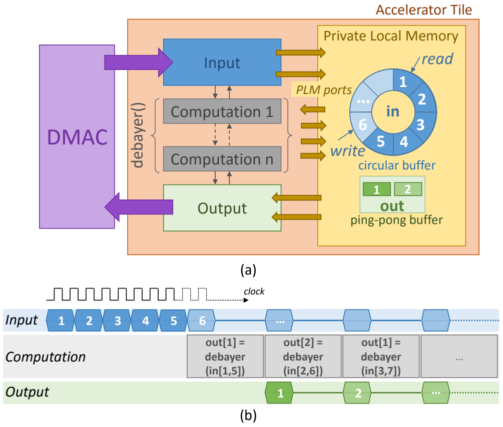


Figure 2: (a) The DEBAYER accelerator structure. (b) Overlapping of computation and communication.

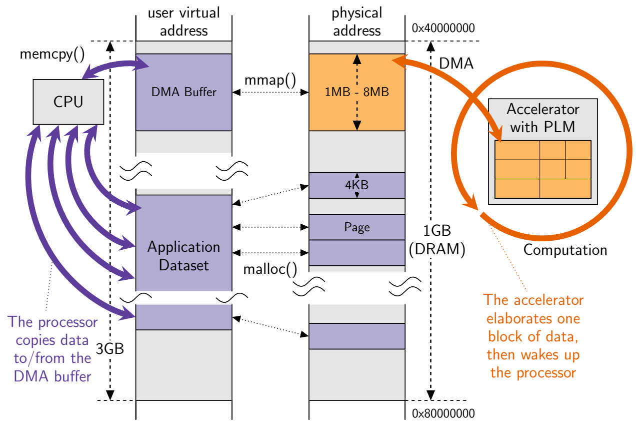


Figure 3: Traditional software-managed DMA.

We define *the Large Data Set (LDS) Problem for SoC Accelerators* as the problem of finding a high-performance and low-overhead mechanism that allows hardware accelerators to process large data sets without incurring penalties for data transfers. A possible solution to the LDS Problem is to have the accelerators share the virtual address space of the processor in a fully coherent way. This is obtained by replacing the PLM with a standard private L1-cache and sharing the higher levels of the memory hierarchy and the memory-management unit with general-purpose cores [34]. This approach, however, is not effective for high-throughput accelerators because it degrades their performance by depriving them from their customized PLMs. Moreover, as the data set grows, the overhead of maintaining coherence further limits the accelerator speedup over software [10]. Alternatively, one could expose the PLM to the processor and let it manage data transfers across separate address spaces in suitable small chunks [16]. However, as we show in Section 2, this increases software complexity and forces accelerators to stall waiting for the software-managed transfers, thus wasting most of the speedup offered by the dedicated hardware.

In this paper, we present an effective hardware-software solution to the LDS Problem that avoids most common shortcomings of accelerators coupled with embedded processors. Our solution achieves this by combining the following features:

- a low-overhead *accelerator virtual address space*, which is distinct from the processor virtual address space (to reduce the processor-accelerator interaction);
- direct sharing of physical memory across processors and accelerators (to avoid redundant copies of data);
- a dedicated DMA controller with specialized translation-lookaside buffer (TLB) per accelerator (to support many heterogeneous accelerators coexisting in the same SoC, each with its specific memory-access pattern);
- hardware and software support for implementing run-time policies to balance traffic among available DRAM channels.

After motivating our work in Section 2, we describe our approach in Section 3 and its system-level integration in Section 4. Section 5 shows a full-system evaluation on FPGA. Our experiments demonstrate that we are able to preserve the accelerators' speedup as the number of concurrent workloads and the size of the data sets scale.

2. PRESERVING THE SPEEDUP

A loosely-coupled accelerator executes very efficiently as long as it keeps its computation and communication phases balanced. As an example, Fig. 2(a) shows the high-level block diagram of a high-throughput accelerator for the DEBAYER kernel [2]. This kernel takes as input a Bayer-array image with one color sample per pixel and returns an image with three-color samples (red, green and blue) per pixel, where the missing colors are estimated via interpolation. The accelerator consists of a *load* module (to fetch data from DRAM), one or more *computation* modules, and a *store* module (to send results to DRAM). These modules communicate through a PLM, which is composed of multiple banks and ports. Such PLM architecture allows the computational modules to process multiple pixels per clock cycle. Additionally, circular and ping-pong data buffers support the pipelining of computation and DMA transfers with the off-chip DRAM. This choice derives directly from the functional specification of the kernel: the DEBAYER interpolates pixels row-by-row and uses 5×5-interpolation masks centered in the pixel of interest. To start the computation, the accelerator needs at least the first five rows of the input image in the circular buffer (input bursts from 1 to 5 in Fig. 2(b)). Then, while the computation modules work, the input module can prefetch more rows for future processing (input burst 6 in Fig. 2(b)). As soon as a computation step completes, an interpolated row is stored in the first half of the ping-pong buffer so that it can be transferred back to DRAM (output burst 1). Meanwhile, the computation modules can start processing the additional row in the circular buffer and storing the result in the second half of the ping-pong buffer (output burst 2).

This behavior represents well many high-throughput accelerators. However, the specifics of the micro-architecture of any given accelerator, including the PLM organization, may vary considerably depending on the particular computation kernel. The timing diagram in Fig. 2(b) shows a hypothetical scenario, where the communication (i.e. input and output) and computation phases are overlapping, and the latency of DMA transfers is hidden by the local buffers. Intuitively, if such latency becomes larger than processing time, then the accelerator must be stalled until new data are available for computation. This can limit the efficiency of the accelerator, reducing its advantages over software execution. Next, we present an experiment demonstrating that, when loosely-coupled accelerators process large data sets, traditional memory handling for non-coherent devices leads to such undesirable scenario.

By using an FPGA board, we realized a simple SoC that integrates one embedded processor with a 32-bit architecture, which runs the Linux Operating System (OS), and two loosely-coupled accelerators. These two accelerators implement the DEBAYER and SORT computational kernels [2]. The virtual memory available to user-level applications is 3GB, while the actual physical memory

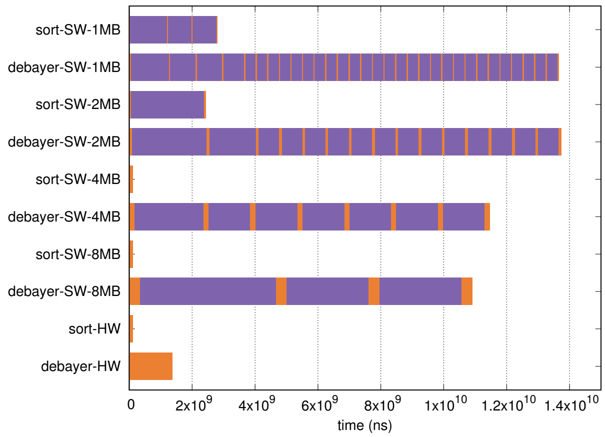


Figure 4: Software-managed DMA versus hardware-only DMA execution time breakdown. Orange segments correspond to accelerator DMA and computation, while purple segments represent software-handled data transfers.

is 1GB. For this experiment we considered a memory footprint of 32MB for DEBAYER, which elaborates one 2048×2048-pixel Bayer-array and the corresponding bitmap image (16-bit colors), and of 4MB for SORT, which processes in place 1024 vectors each containing 1024 single-precision floating point numbers. The two accelerated applications share the processor in time multiplexing according to the Linux scheduler. Each of them can invoke the appropriate accelerator through the API provided by a device driver.

Fig. 3 shows the memory layout of one application: the physical memory is usually allocated in 4KB pages and remapped to a contiguous virtual memory area where the program stores the application's data. To allow a non-coherent device to access these data, the driver typically implements a memory-mapping function that serves three main tasks. First, it requests the OS to reserve a contiguous area in physical memory for DMA and pins the corresponding pages (orange-shaded memory area in Fig. 3). Then, it passes the physical address to the device, referred to as `dma_handle` of the allocated buffer. Finally, it remaps the *DMA buffer* to the virtual memory (purple-shaded area) and returns a pointer to the user-level application. The exact amount of contiguous memory that the OS can allocate depends on the target processor architecture, but it is usually limited to a few megabytes. If we set the size of the DMA buffer to 1MB, the DEBAYER computation can be easily split into 32 parts, each processing a different portion of the input image. Similarly the input vectors for SORT can be divided into 4 sets of 256 vectors each. The bars of Fig. 4 show how hardware acceleration (orange) and software execution (purple) interleave over time. The orange segments include the time for fetching the input data from DRAM via DMA, elaborating them, and transferring results back to DRAM also via DMA. The purple segments, instead, correspond to the time spent by the user application in saving results from the DMA buffer to another virtual memory area and copying the next block of input data into the DMA buffer. Note that the first and the last segments of each bar are always orange since we are not considering the application setup and wrap-up phases, which are constant across all scenarios. We repeated the experiment four times, varying the size of the DMA buffer from 1MB up to 8MB. As the size of the DMA buffer increases, the data processed by the accelerators are split into fewer blocks and the overhead of interleaving hardware and software decreases. Further, the execution of SORT benefits from having a DMA buffer large enough for its memory footprint: the accelerator is able to complete the entire task without the intervention of the processor, thus obtaining a speedup of 21× over the test case with a 2MB DMA buffer. For DEBAYER, however, the software-based data management is always responsible for the largest part of the execution time, because its memory footprint never fits into the DMA buffer. Additionally, the execution of multiple accelerators creates contention on the processor, which must handle multiple concurrent transfers between each DMA buffer and the virtual memory of the corresponding application. This is shown by the purple bars that become longer when the two accelerators execute at the same time.

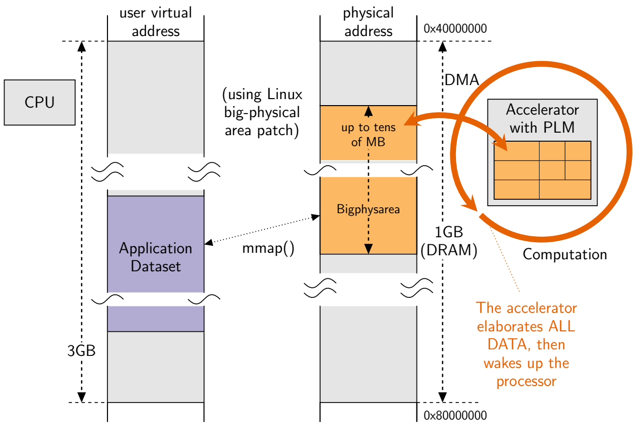


Figure 5: Hardware-only DMA using Linux *big-physical area* patch to reserve up to tens of MB of contiguous memory.

Following the intuition that avoiding the intervention of the processor core in DMA transfers (except from the initial setup) benefits the accelerated application, we executed again the experiment leveraging a Linux patch known as *big-physical area*. When enabled, this patch forces the Linux OS to reserve a region of contiguous memory configurable in size up to a few tens of megabytes. Fig. 5 shows the updated memory layout made possible by the patch: the entire application data set for both SORT and DEBAYER can be mapped to contiguous physical memory. Hence, the accelerator needs only the base address of the buffer to process all data, while the processor can remain idle or perform other tasks. The result is reported in the last two bars at the bottom of Fig. 4: the accelerator for DEBAYER achieves a speedup of 8× with respect to the scenario with an 8MB DMA buffer.

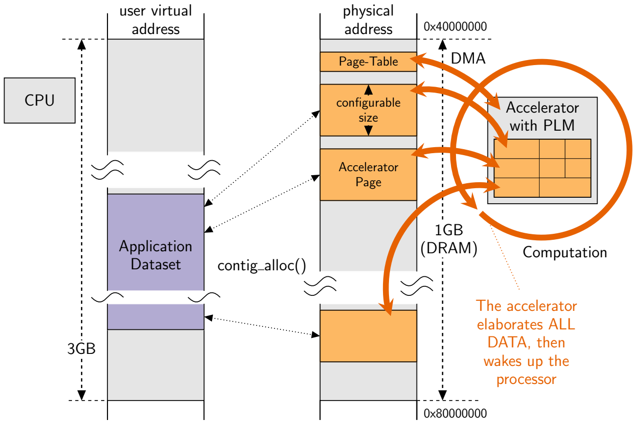
This experiment proves the benefits of reducing the processor intervention when loosely-coupled accelerators move data with DMA transactions. The *big-physical area* patch, however, is only viable for applications with medium-sized memory footprints. As the number of accelerators and the size of data sets grow, it is necessary to adopt a more scalable and flexible approach.

3. HANDLING LARGE DATA SETS

In the context of general purpose processors, cache hierarchy and virtual memory are typically used to give user applications the illusion of accessing the entire address space with low-latency. As the number of accelerators integrated in SoCs keeps growing, designers need a similar efficient solution dedicated to special-purpose hardware components. In this section we describe a combination of hardware and software techniques that gives accelerators the illusion of accessing contiguous physical memory. Each accelerator can therefore issue memory references using an *accelerator-virtual address (AVA)*, equivalent to a simple offset with respect to its data

```
/* Data structure for contig_alloc */
struct contig_alloc_req {
 unsigned long size;                  /* aggregate size required */
 unsigned long block_size;                    /* size of one block */
 struct contig_alloc_params params; /* DRAM allocation policy */
 unsigned int n_blocks;         /* number of contiguous blocks */
 contig_khandle_t khandle;     /* handle for the device driver */
 unsigned long __user *arr; /* blocks physical addresses (PT) */
 void __user **mm;        /* user-space mapping of the blocks */
};
```

Figure 6: Data structure to request an accelerator buffer.



The accelerator elaborates ALL DATA, then wakes up the processor

Figure 7: Memory layout after calling *contig_alloc* to enable low-overhead scatter-gather DMA for accelerators.

structures, without requiring any information about the underlying system memory hierarchy. Combined with a lightweight dedicated DMA controller, this makes all transactions occur across the entire data set without intervention of the processor, thus allowing the accelerators to preserve the speedup they were initially designed for.

Scatter-Gather DMA and accelerators. For off-chip peripherals and non-coherent devices, the standard Linux API provides routines to create a list of pages reserved for any virtual buffer. This list, called *scatterlist*, represents the *page table* (PT) for the buffer. This name refers to *scatter-gather DMA*, which is a common technique mostly applied to move data between main memory and the dedicated DRAM of on-board peripherals. To reduce the size of the PT, Linux tends to reserve blocks of contiguous pages whenever possible so that it is sufficient to store the base address and length of each block. A typical transaction to an external peripheral implies transferring all data stored in the area pointed by the PT. Hence, the scatter-gather DMA controller must simply walk the PT and gather data from all memory areas in order. Conversely, on-chip accelerators must deal with PLMs having limited size. Therefore, they have to issue several random accesses to memory, following a pattern that is highly-dependent on the implemented algorithm. Since the blocks may have different sizes, the access to a scattered memory buffer with a random offset requires the addition of every block length until the requested data is effectively reached. Moreover, long DMA transfers may easily span across multiple blocks, incurring further overhead to complete the transaction.

Alternatively, Linux can guarantee a set of equally-sized blocks, each consisting of one page (typically 4KB). However, if we consider a data set of 300MB, we need 76,800 entries in the PT, equivalent to 300KB on a 32-bit address space. A traditional TLB, holding only a few of these PT entries, would incur high miss rates. In fact, high-throughput accelerators do not typically reuse the same data multiple times and very little spatial locality can be exploited. To overcome this issue, we implemented a kernel module, named `contig_alloc`, and a companion user-space library to replace the standard `malloc` interface. Fig 6 shows the request data structure for our module. A request to `contig_alloc` includes the size of the requested memory area (*size*), the desired size of each contiguous physical block (*block_size*) into which the memory region will be divided, and some allocation policy parameters. These parameters are intended for load balancing in case of multiple DRAM banks. If we specify only the parameter *size*, as typically done for `malloc`, then default values are used for the other parameters.

The kernel module generates a DMA handle for the accelerator's driver, the resulting number of equally-sized blocks (also called *accelerator pages*), the corresponding PT, and the virtual-memory mapping for the user-space application. Fig. 7 shows the memory layout after calling `contig_alloc`. Note that only the calling process is allowed to access the allocated memory region and the user-level application can still operate transparently on the data in its virtual address space (purple-shaded area). However, differently from standard allocation mechanisms, the corresponding physical pages have larger size (orange-shaded regions in physical memory). For very large data-sets, we can set a medium size for the accelerator pages (e.g. 1MB) so that the resulting PT has a size on the order of a few KBs and can be thus stored contiguously in memory, as shown in Fig. 7.

This approach enables a low-overhead version of scatter-gather DMA specialized for loosely-coupled accelerators, while maintaining shared memory across processors and accelerators[2], without requiring coherence with the PLMs.

TLB and DMA controller for accelerators. Once the data are ready in memory, laid out as shown in Fig. 7, the application can run the accelerator by invoking the device driver through the traditional `ioctl` system call. The driver takes the configuration parameters from the user application and passes them to the accelerator through memory-mapped registers. Such parameters include application-specific variables to be used directly by the accelerator kernel (e.g. the size of the image for the DEBAYER application), and the information for the DMA controller (e.g. the memory address where the PT is stored). To guarantee memory consistency without coherence, the device driver performs a simple flush of the cache lines holding data from the shared buffers right before sending the start command to the accelerator. This operation is completely transparent to the user-level applications and incurs negligible performance overhead.

At this stage, DMA and computation are entirely managed by the accelerator. The accelerator requests are composed of a set of control signals to: distinguish memory-to-device from device-to-memory transfers, set an offset with respect to the data structure to process (corresponding to the AVA), and determine the transaction length. To serve such requests we equip every accelerator with a DMA controller (DMAC) and a parametrized TLB. These components autonomously fetch the PT through a single memory-to-device transaction, and store it inside the TLB. Once the TLB is initialized, every accelerator request is translated in only four cycles. When operating on large data sets with very long DMA transfers, this address translation overhead is negligible. The TLB is configured to match the requirements of a given accelerator in terms of number of supported physical memory pages and their size. In fact, thanks to `contig_alloc`, the number of pages is kept under control and set according to the size of the required mem-

[2] Differently from Linux `huge_pages`, our module supports dynamic allocation of blocks; the latter can have variable sizes, trading off PT size for memory fragmentation; and it is supported across all architectures.

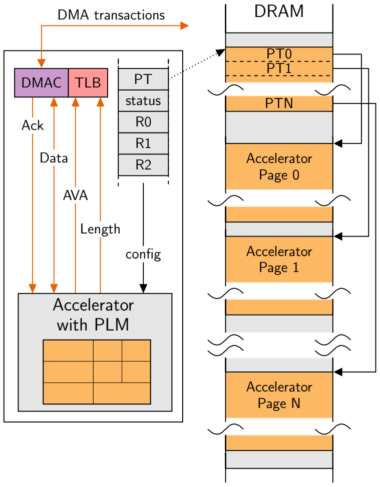
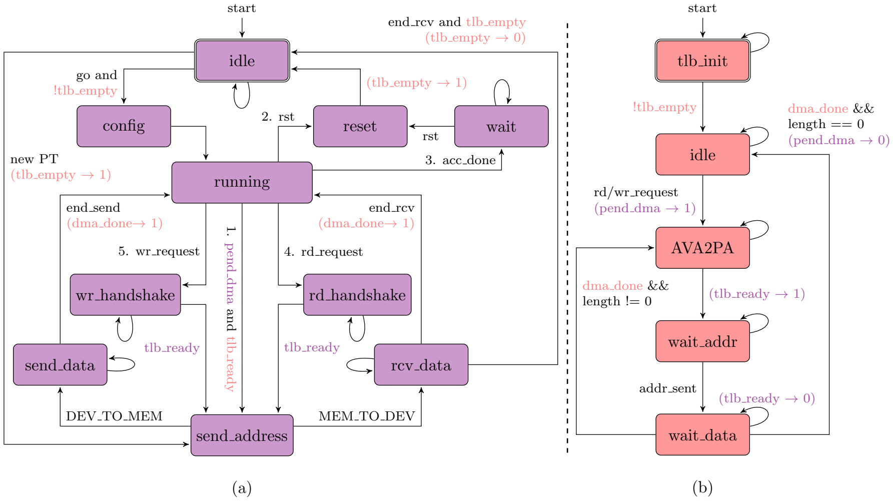


Figure 8: DMA controller interface.



(a)

(b)

Figure 9: DMA Controller (a) and TLB (b) finite state machines.

ory area. Therefore it is possible to have the number of PT entries match the TLB size. This not only simplifies the design, but it also minimizes the performance degradation due to scatter-gather DMA. Indeed, filling in the TLB can be done with one single transfer before activating the computational blocks. Results reported in Section 5 confirm that preparing and using the TLB has a negligible impact on the overall execution time of the accelerators. Across the analyzed workloads, we set the accelerator page size to 1MB, which is a reasonable trade-off between the complexity of the memory allocation performed by the operating system and the PT size, resulting in few hundreds entries. Should an application require more entries, in order to handle even larger data sets, the TLB can be parametrized to hold more pointers in exchange for silicon area. Note that the relative performance overhead would not increase, because transactions and computation would also scale with the data set.

Fig. 8 shows the organization of the accelerator, the DMAC, the dedicated TLB, and the bank of configuration registers. Note that one of the registers stores the DMA handle generated by `contig_alloc`. This corresponds to the PT base address and is used to initialize the TLB. The DMAC and TLB behaviors are described by the finite state machine in Fig. 9(a) and Fig. 9(b), respectively. As soon as the PT register is written by the device driver, the DMAC engine initiates an autonomous transaction to retrieve the PT, as shown by the transition from *idle* to *send_address* in Fig. 9(a). The request includes the PT address and the number of entries to fetch. Then, following the control flow of read requests (i.e. *MEM_TO_DEV* path), the DMA waits for the response of the memory controller before transferring the received pointers to the physical blocks into the TLB. The operation terminates when all entries are received: this corresponds to the transaction from *rcv_data* to *idle* in Fig. 9(a), where the signal *tlb_empty* is deasserted. This also corresponds to the transition from *tlb_init* to *idle* in Fig. 9(b).

After this TLB initialization, the DMAC steps through the states *config* and *running*, and starts its execution. Whenever the accelerator needs to perform a read or write request to DRAM, it sends a request to the DMAC through its DMA interface, as shown in Fig. 8. Specifically, the AVA and the length of the data transfer are sent to the TLB, which initiates the address translation, while the DMAC starts a handshake protocol with the DMA interface of the accelerator. The TLB determines whether the transaction needs to access one or multiple pages in memory, and computes the length of the transfer for the first accelerator page. In four cycles the TLB is ready to provide the physical address and the DMAC initiates a transaction over the interconnection system, following either the *MEM_TO_DEV* or the *DEV_TO_MEM* paths, for read or write operations, respectively. In the case of read requests, the acknowledge signal (*Ack*) shown in Fig. 8 is set when valid data are available. Conversely, in the case of write requests, the signal *Ack* is set when an output value (*Data*) has been sent to the DMAC. This simple latency-insensitive protocol [4] ensures functional correctness, while coping with congestion and DRAM latency. After the request has been sent to the interconnect, the TLB controller steps to a second waiting state (i.e. *wait_data* in Fig. 9(b)). In this state, if the current transfer length does not match the actual length requested by the accelerator, the controller reads the physical address of the next page in the TLB and initiates another transaction skipping the handshake with the accelerator. When the DMAC returns to the state *running*, it checks first for pending transactions, then it reads the command register to check for a reset from software, and finally waits for the accelerator to raise another request or for completion (i.e. signal *acc_done*). Note that, even considering the DMAC initialization, the delay introduced by each accelerator request is negligible when compared to the lengths of typical burst data transfers (thousands of words).

Main memory load balancing. As the number of accelerators grows, the system interconnect and the I/O channels to the external memory are responsible for sustaining the increasing traffic generated by many long DMA transactions. A system interconnect based on a network-on-chip (NoC) offers larger throughput and has better scalability than traditional bus-based interconnects [12]. However, since all accelerators need access to external memory, it is necessary to improve the traffic on the NoC to minimize congestion. The availability of multiple memory channels and DDR controllers on modern systems improves the NoC traffic by balancing the data allocation among such controllers. The optional parameters of `contig_alloc` enable the user to distribute traffic through different paths to the DRAM banks. In addition to such parameters, when loading the kernel module, it is possible to specify the region of the physical address space where `contig_alloc` is allowed to request accelerator pages. The presence of multiple channels to the external memory allows additional control of the interconnect traffic. In order to perform a sensitivity analysis of the proposed design with respect to load balancing, we devised three allocation policies:

1. `POLICY PREFERRED` returns the first available accelerator pages starting from the most affine DDR controller (i.e. the closest one in the SoC layout to the accelerator owning this set of pages).
2. `POLICY LEAST-LOADED` returns all pages from the least loaded DDR. This policy can be tuned with a user-defined threshold parameter that biases the priority among DDR controllers. For example, if the kernel module is allowed to allocate pages on half of the address space corresponding to the first DDR controller (namely `DDR0`), it is convenient to set a threshold for this policy. For instance, by setting the threshold to 16, the policy will allocate the requested pages to `DDR0` only if all other DDR nodes have at least 16 more allocated pages than `DDR0`. Note, in fact, that the region of memory exclusively dedicated to the standard operating system memory allocation mechanism is accessed more frequently by the processors. Hence, the system incurs higher contention between accelerators and processors when many accelerator pages are located on `DDR0`.
3. `POLICY BALANCED` returns sets of pages with specified cardinality. Each set of pages is alternatively allocated on the available DDR controllers. This policy accepts the same threshold as `POLICY LEAST-LOADED` to select the first DDR controller. Additionally, the number of pages per set is also specified by the user and determines the granularity for balancing the allocation.

4. SOC INTEGRATION

To evaluate our solution to the LDS Problem, we implemented loosely-coupled accelerators for a set of computing kernels from the Perfect Benchmark Suite [2]. This is a collection of applications and kernels targeting energy-efficient high-performance embedded computing. In particular, we selected eight kernels that are very heterogeneous as they process a variety of input/output data sets, with different memory-access patterns and communication vs. computation ratios. Consequently, we design a corresponding set of accelerators that share the general structure shown in Fig. 2 but have major differences in terms of the micro-architecture of the computational blocks and the PLM structure.

Accelerated kernels. The eight selected computational kernels operate on input data sets that are provided as part of the Perfect suite in three different sizes: Small, Medium and Large. The actual sizes vary across kernels and range from 1MB to 300MB. Except for Sort, which is executed in-place, the applications must allocate additional data structures to store output and temporary data. Thus, their memory footprint grows up to 345MB, as shown in Table 1. Also, our kernel implementations confirm the impact of the PLM on the overall design of an accelerator: the PLM ranges from 75% up to 98% of the accelerator area, when targeting an industrial 32nm CMOS technology.

Kernel	SW app.	PLM	FPGA			CMOS
	footprint		Resources			Area
	MB	KB	LUT	FF	BRAM	μm^2
Sort			36,868	31,300		281,045
−Mem.	18.2	24.00			6	74.95%
FFT2D			3,965	2,190		834,147
−Mem.	292.3	128.00			48	94.13%
Debayer			4,446	1,968		796,920
−Mem.	42.3	95.86			32	98.53%
Lucas-Kan			5,329	3,210		319,109
−Mem.	173.4	20.28			8	84.42%
Change-Det.			16,274	6,378		596,029
−Mem.	345.4	63.00			18	90.57%
Interp.1			20,836	9,119		492,647
−Mem.	109.4	48.05			12	69.65%
Interp.2			20,908	8,623		575,561
−Mem.	137.2	64.05			16	76.67%
Backproj.			14,040	5,588		782,263
−Mem.	329.3	99.00			81	91.61%

Table 1: Characterization of the implemented accelerators.

All the selected kernels execute heavy computation tasks, but they are very heterogeneous in terms of data access patterns. Sort, for instance, reorders iteratively and in-place *N* arrays of 1024 floating-point elements, where *N* can be 256, 512 or 1024, depending on the data set. FFT2D performs the Fast Fourier Transform (FFT) on each of the input rows of length 2^N, then it transposes the resulting matrix and finally it performs FFT on the transposed-matrix rows. Thus, it requires an additional workspace of the same size of the input matrix, i.e. $2^N \times 2^N$, where N is at most 12. The Debayer kernel takes as an input an $N \times N$-pixel Bayer-array image (with *N* ranging from 512 to 2048 pixels) with one color sample per pixel and returns an image with three-color samples per pixel. Thus, the resulting output is three times bigger than the input. Moreover, the algorithm interpolates pixels row-by-row and uses 5×5-interpolation masks centered on the pixel of interest. Lucas Kanade performs image alignment. The algorithm has a multiply-accumulate nature that stores the results in the Hessian output matrix. This has a fixed size (6×6) independently from the size of the input images. Its memory footprint grows significantly with the larger data sets due to the amount of intermediate results that the algorithm allocates on the memory stack. Indeed, it has the highest growth rate among all kernels. Furthermore, each iteration of its computation phase requires two independent memory-read operations: the access pattern of the first one is data dependent, while the second transaction has a behavior known ahead of computation. Since the accelerator can be implemented without considering the SoC memory subsystem, these irregular memory accesses do not exacerbate the complexity of the accelerator. Change Detection takes as input a sequence of frames and an initial training set, and it returns a new training set and a "ground-truth mask": certain portions of each frame are labeled as background. Both frames and training set are represented as a set of N×N-pixel matrices where *N* is at most 2048. This results in the biggest data set among the kernels (300MB). On the target platform, this application has a memory footprint of 345.4MB. The other three kernels are part of a radar-based imaging application that produces high-resolution imagery by composing data from relatively small images. Two alternative methods of image formation exist: polar format algorithm (PFA) and backprojection algorithm. The Interpolation-1 and Interpolation-2 kernels are the most computational intensive portions of PFA. All three operate on large matrices, but Interpolation-1 reads them row-by-row, Interpolation-2 accesses them column-by-column, and Backprojection has a data-dependent access pattern.

SoC architecture. We integrated this variety of accelerators in multiple SoCs that we designed with the *embedded scalable platforms* methodology [5]. Each SoC has a tile-based architecture and features four types of tiles. A *CPU tile* integrates a Leon3 embedded processor [15] that runs the Linux OS and the embedded software stack, including the `contig_alloc` module, the device drivers, and the user applications. Each *DDRx tile* has a memory controller offering one independent channel to the external memory. A *MISC* tile implements all other I/O channels and peripherals that are responsible for booting the system and supporting a debug

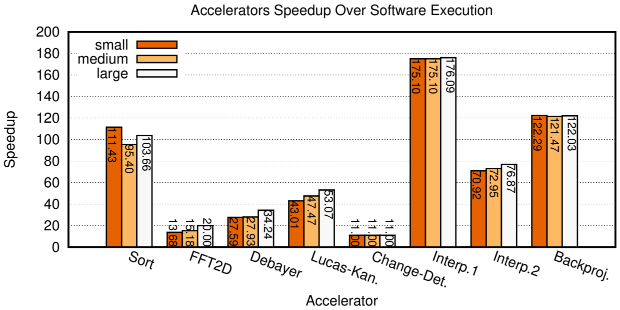


Figure 10: Speed-up of each accelerator with respect the corresponding software executions for three data-set sizes.

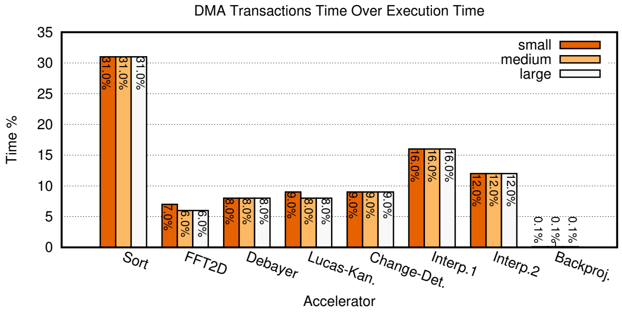


Figure 11: Time spent in data transfers expressed as a fraction of the total execution time for each accelerator.

interface. Lastly, each *accelerator tile* encapsulates a given accelerator together with an instance of the components of Fig. 8, which provide a simple *network interface* between the guest accelerator and the system interconnect. The flexibility of this interface allows us to easily swap or replace tiles to create different memory mappings and test scenarios. The corresponding routing tables are automatically generated [22]. Hence, we are able to quickly evaluate the impact of changing the set of accelerators and the SoC layout.

The tiles are interconnected through a packet-switched multi-plane NoC. Combined with multiple DDR controllers, the NoC supports more concurrent transactions than a bus-based architecture. Accelerators rely on two NoC planes that are dedicated to DMA transactions (one for memory-read and one for memory-write transfers) and guarantee deadlock avoidance. The accelerator DMA does not interfere with the NoC planes dedicated to the processor cache request-and-response transfers until the packets reach the memory. Non-cacheable register operations, control messages, and interrupts are delivered through a fifth plane, which is accessed by all tiles. While the size of the SoC instances are ultimately limited by the available resources on the target FPGA, our infrastructure is inherently modular and scalable: it allows for more tile and NoC planes as the number of integrated accelerators and memory controllers increases.

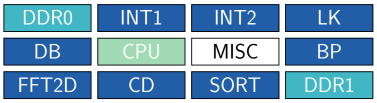
Probes and performance counters. A set of accurate performance counters are placed at the interface of each DMAC and NoC router. They serve as probes to gather statistics during system execution. In particular, probes placed between each accelerator and its DMAC measure the total number of cycles in which the accelerator is active, along with the cycles spent in communication (i.e. when DMA transfers are occurring) and in TLB access. Probes placed at each router port measure the number of cycles when a flit traverses each link. This information helps us quantify the contention for shared resources across the different scenarios.

Implementation details. We performed logic synthesis of all SoC instances for a target frequency of 80MHz and mapped them on a proFPGA Prototyping System [27], equipped with a Xilinx Virtex-7 XC7V2000T FPGA and two DDR-3 extension boards. This provides the system with dual-channel access to memory (namely `DDR0` and `DDR1`). The total addressable off-chip memory is limited to only 1GB by the LEON3 default mapping, which is however sufficient to execute all our workloads. We split this address space into two partitions, each of size 512MB. The lower portion is mapped to `DDR0` and includes 128MB of memory exclusively reserved for the OS that cannot be used by `contig_alloc`. The rest of the address space, instead, is dynamically shared between the processor and the accelerators.

5. FPGA-BASED EVALUATION

We evaluate our approach for solving the LDS problem with a set of experiments across four SoC instances mapped to FPGA.

Hardware solution overhead. The components for translating the requests from each accelerator to the corresponding NoC interface require about 600 look-up-tables (LUT) and 600 flip-flops (FF). Without the logic to support `contig_alloc`, the same DMA engine would require 350 LUTs and 400 FFs. One additional block RAM (BRAM) is needed to store the TLB for each accelerator. However, this difference in terms of resources is negligible when compared to accelerators (see Table 1). We also evaluated the performance overhead to access the TLB, which in aggregate corresponds to a few hundred cycles and is negligible across all workload scenarios if compared to a total execution time that ranges in the hundreds of millions cycles. Address translation and TLB initialization time is indeed eight orders of magnitude smaller than the total accelerator execution time.



DDR0	INT1	INT2	LK
DB	CPU	MISC	BP
FFT2D	CD	SORT	DDR1

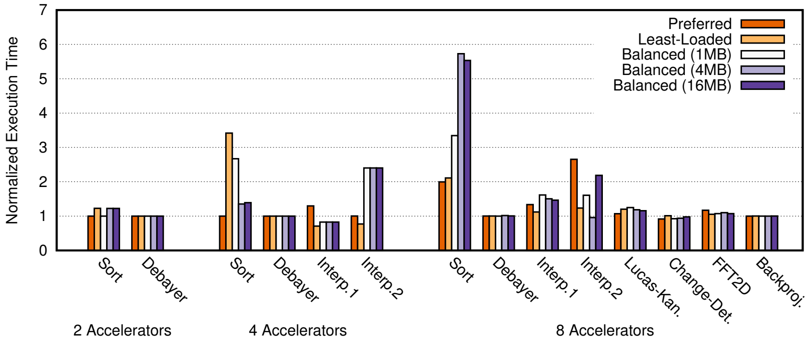


Figure 12: Test scenario (a): eight heterogeneous accelerators with two memory controllers located at opposite corners. The chart shows the execution time when running two, four and eight concurrent accelerators, normalized to single-accelerator workload.

LK	INT1	INT2	CPU
DB	DDR0	DDR1	BP
FFT2D	CD	SORT	MISC

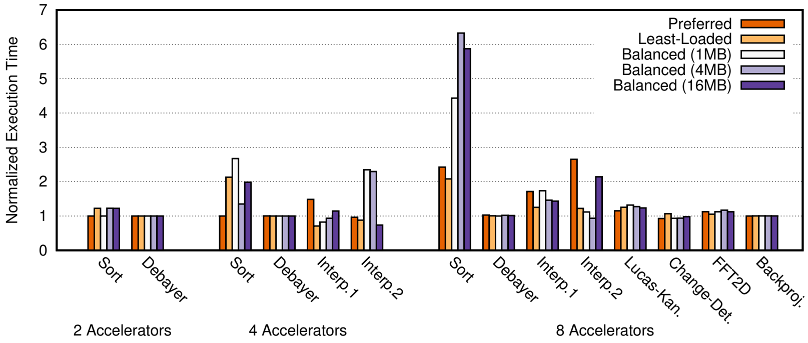


Figure 13: Test scenario (b): eight heterogeneous accelerators with two memory controllers located in the central tiles. The chart shows the execution time when running two, four and eight concurrent accelerators, normalized to single-accelerator workload.

Evaluation of single accelerators. Given the accelerators presented in Section 4, we start testing our architecture by running each of them standalone in order to asses its speedup over the corresponding original software implementation. For this set of experiments, data are allocated on `DDR1`, while `DDR0` is reserved for the processor. This allocation minimizes the contention between the processor and the accelerator. The results are reported in Fig. 10. The speedups range from 11× (for Change Detection) to 175× (for Interpolation-1). These tests were run on all data sets and show the scalability of our solution for up to 300MB of input data. The speedup is almost constant for five accelerators out of eight, while it shows a slight increase on larger data sets for the others. Average speedup across all input data sizes is about 70× and grows to almost 75×, when excluding smaller test sizes.

Fig. 11 reports the percentage of execution time during which each accelerator is involved in a data transfer. For example, the DMA controller for Sort is active for more than 30% of the execution time. This percentage includes both the time where useful data reach or leave the accelerator tile and the waiting time caused by DDR latency. Such metric is a key characteristic of the accelerator, which depends primarily on the ratio between computation time and communication time and on how much these two phases are allowed to overlap. Higher percentages of communication time correspond to a larger sensitivity to system congestion and memory bandwidth, because the accelerator tends to perform less operations on each byte of data brought to the PLM. For this reason, it can be easily stalled when varying the latency of memory transfers. Note that applications performing very little computation on each data token are not suitable for loosely-coupled accelerators[10].

Multi-accelerator workloads. We analyze the interaction of multiple accelerators by sweeping the number of concurrent accelerators and changing the memory allocation policy, chosen among those described in Section 3. The limited amount of addressable DDR does not allow us to execute *concurrently* all accelerators with large data sets at the same time. However, throughout the experiments we are able to allocate up to 768 MB, corresponding to as many accelerator pages. The first SoC instance integrates one copy of each accelerator implemented, and has two memory channels located at the corners of the NoC, as shown in Fig. 12. The second test case, reported in Fig. 13, is similar to the previous one, except for the location of the memory controllers. These are now positioned in the central tiles to investigate the sensitivity of our design to the placement of the most contended shared resources. The third SoC, shown in Fig. 14, integrates two copies of five different accelerators, for a total of ten accelerator tiles. Having two copies of each accelerator reduces the degree of heterogeneity and affects the traffic over the interconnect because there are more components having the same access patterns to memory. The last test case integrates twelve accelerators for the FFT2D kernel (Fig. 15) and stresses the system with homogeneous traffic patterns generated from all accelerator tiles. The bar charts next to each SoC layout report the execution time for every accelerator, across several experiments. Each bar is normalized against the corresponding single-accelerator execution time. Each group of clustered bars corresponds to a workload scenario with multiple accelerators running at the same time. For instance, the chart in Fig. 12 reports three workloads, running two, four and eight accelerators, respectively. For every workload, we repeated the experiment for five different allocation policies. The leftmost bar in each cluster corresponds to `POLICY PREFERRED`, which has no configuration parameters. The second bar (in yellow) shows the results for `POLICY LEAST-LOADED` configured with a penalty of 32MB for `DDR0`, so that `DDR1` is preferred when both banks are similarly loaded. Finally, the three bars in different shades of purple correspond to `POLICY BALANCED` with sets of 1, 4 and 16 pages, sized 1MB each.

The first conclusion we can draw is that for small sets of accelerators the execution time is mostly unaffected by concurrency. This is shown for heterogeneous workloads with two accelerators in Fig. 12, 13 and 14, and for the case of four FFT2D in Fig. 15. The little fluctuations reported are due to unpredictable behavior of the

DDR0	MISC	CPU	
FFT2D	FFT2D	DB	DB
LK	LK	INT1	INT1
	INT2	INT2	DDR1

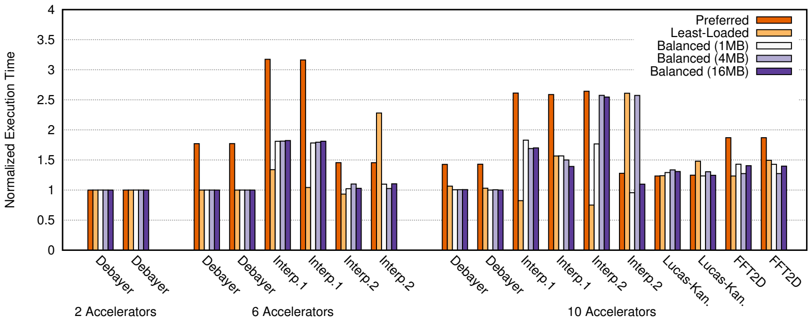
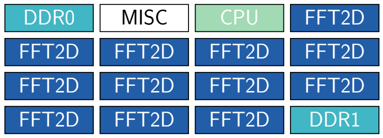


Figure 14: Test scenario (c): five couples of heterogeneous accelerators with two memory controllers located at opposite corners. The chart shows the execution time when running two, six and ten concurrent accelerators, normalized to single-accelerator workload.



DDR0	MISC	CPU	FFT2D
FFT2D	FFT2D	FFT2D	FFT2D
FFT2D	FFT2D	FFT2D	FFT2D
FFT2D	FFT2D	FFT2D	DDR1

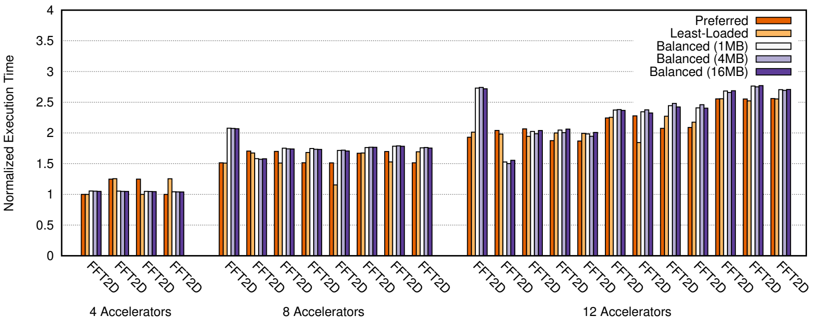


Figure 15: Test scenario (d): twelve homogeneous accelerators with two memory controllers located at opposite corners. The chart shows the execution time when running four, eight and twelve concurrent accelerators, normalized to single-accelerator workload.

system, where the OS is constantly running and generating "noise" in terms of memory utilization. As we increase the number of accelerators running concurrently, the effects of contention for the shared resources starts affecting the execution time. For instance, SORT is heavily penalized by the higher ratio between communication and total execution time, as already noted for Fig 11. Nevertheless, the aggregate performance of multiple concurrent accelerators keeps improving, even if with diminishing returns. For example, if we consider the second group of bars in Fig. 15, we distinguish eight clusters for as many instances of parallel FFT2D accelerators. As we can see from the chart, they not only perform better than a single accelerator running in series, but they also exceed the performance of the scenario with four FFT2D. The average execution time, across all policies, for eight FFT2D, in fact, is below the break-even point of 2×. From this viewpoint, even better results are shown for the heterogeneous SoCs. Having different accelerators, in fact, leads to the interaction of heterogeneous data access patterns, which tend to reduce contention for shared resources. These results show that, as long as the interconnect can sustain the bandwidth requirements of the accelerators, our design scales well with limited impact on performance. Furthermore, if we compare Fig.12 and 13, we can conclude that our technique is robust to variations of the SoC layout (i.e. position of tiles). In particular, moving the memory controllers from the corners to the central tiles has no impact on the system behavior, even though the traffic distribution on the NoC changes significantly.

Finally, by looking at the results for all workloads, we notice that the allocation policy has little to no impact on the performance in most cases and for most accelerators. Such behavior is highly desirable, because it does not constrain the operating system to use one specific load balancing technique for memory. Note that there are few exceptions to this observation. For instance, results for SORT in Fig. 12 and 13 show significant variations in the execution time based on the allocation policy. Indeed, on one hand, accelerators like SORT that have a higher ratio between communication and total execution time (see Fig. 11) require higher bandwidth with the memory. On the other hand, when multiple accelerators' buffers are scattered across the two DDR controllers, there are on average more packets colliding on the NoC and this can affect the performance. In fact, the probes located inside the NoC routers measured on average 3× more packets traversing the links around the tile for SORT when changing allocation policy from `PREFERRED` to `1MB-BALANCED`. Hence, we can conclude that the reported performance loss is not directly correlated with our DMA and address translation logic. Instead, it is a natural consequence of a higher NoC traffic.

6. RELATED WORK

Accelerator Memory. Addressing memory aspects of specialized accelerators is fundamental to design efficient SoCs. In fact, even if the PLM could reach 90% of the chip area, the amount of data that can be stored on-chip is usually limited to few MBs [21]. Efficient methods have been proposed to reduce the footprint of the on-chip memory by exploiting sharing techniques [8, 21, 26], to perform data prefetch and reduce the latency in accessing the external memory [33], and to improve utilization of the silicon area dedicated to memory [11, 9, 14]. However, before this paper, there has been no comprehensive analysis of the effects of multiple accelerators processing concurrently large amounts of data accessed through off-chip memory. The effects of multiple accelerators accessing the same memory controller has been studied before as part of a work that proposes a configurable module to manage many access patterns [17]. This module, however, is tightly coupled with the controller and the extension to multiple memory controllers is not straightforward. Further, in this case the accelerators need to know where the data are allocated in order to perform the request to the proper controller. Instead, we address a more general case: non-contiguous buffers are not determined by the algorithms (or the implementation of the accelerators), but they may be generated by the OS based on the current workload (i.e. the interaction with other accelerators and the size of their data sets); hence the accelerators cannot know in advance how the data will be allocated. Other approaches where accelerators are unaware of the memory subsystem [34, 32] address a type of accelerator accessing memory with small transactions. This is similar to what general or special purpose processors do when loading cache lines. High-throughput loosely-coupled accelerators, instead, tend to process larger data sets and access them at a coarser granularity and need long DMA transfers with DRAM [10]. Various solutions have been proposed to expose accelerator memory to the OS and transfer large data [30] through scatter-gather DMA mechanisms. However, these solutions are usually implemented inside the OS and performed by the processor. Further, data are transferred to the FPGA memory that is managed as a peripheral; the transfers are thus serialized. Instead, we allow the accelerators to autonomously manage the memory accesses, even in case of non-contiguous buffers.

Architectures with Multiple Memory Controllers. The impact of multiple memory controllers has been studied for multicore architectures [1], especially to manage data placement, handle the effects on memory-access latency, and avoid conflicts while scheduling accesses to the same physical memory [19]. In contrast, we build a dual-channel memory system that, combined with data placement techniques, can effectively parallelize the accesses to different physical memories. Techniques based on application profiling to place data across many memory channels [25] are orthogonal to our work and can be integrated in our infrastructure.

Architectural solutions for heterogeneous architectures are usually evaluated by simulation [8, 17, 21]. However, as the complexity of these architectures increases, this approach is becoming unfeasible. In fact, simulators developers usually need to abstract some behaviors to reduce the simulation time. Hence, it is impossible to have an accurate analysis of the interaction between processor, operating system, and accelerators, especially in the case of contention on resources (i.e. off-chip memory). On the other hand, FPGAs have been used to emulate specific aspects of the design such as NoC behavior [20], to evaluate the optimization of accelerator PLMs [26], and also to implement accelerator-rich architectures [6]. However, our work is the first to use FPGAs to study the interaction between on-chip and off-chip memories in complex SoCs when many accelerators are processing simultaneously very large data sets.

Accelerators and OS. Programmability of accelerators is another critical issue in the design of heterogeneous SoCs. Device drivers are usually adopted to configure the accelerators with specific parameters and to control the execution. However, there is a domain disparity between processor cores and hardware accelerators [30]. The processor core is usually responsible for the data allocation by leveraging the specific API of the OS, which may result in non-contiguous buffers that are not usually support by accelerators. When handling large data, however, the OS needs to adopt specific techniques to manage these large buffers, e.g. virtual pages. The solution we propose reduces this disparity.

7. CONCLUSIONS

We presented a combined hardware-software solution and evaluation for the Large Data Set Problem in accelerator-based SoCs. Our design includes dedicated hardware and a software stack to efficiently support the execution of high-throughput accelerators processing large data sets. We evaluated our design through a full-system FPGA-based implementation, demonstrating four main properties: (1) feasibility of our solution; (2) low sensitivity to workload characteristics and accelerator-specific behaviors; (3) low sensitivity to placement of accelerators with respect to the location of DDR controllers and the load-balancing policies; and (4) scalability across data sets and number of concurrent accelerators.

Acknowledgments. This work is supported in part by DARPA PERFECT (C#: R0011-13-C-0003), the NSF (A#: 1219001), and C-FAR (C#: 2013-MA-2384), an SRC STARnet center.

8. REFERENCES

[1] M. Awasthi, et al. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the International Conference on Parallel architectures and compilation techniques (PACT)*, pages 319–330, Sept. 2010.

[2] K. Barker, et al. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *Communication of the ACM*, 54:67–77, May 2011.

[4] L. P. Carloni. From latency insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, Nov. 2015.

[5] L. P. Carloni. The case for embedded scalable platforms. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016.

[6] Y.-T. Chen, et al. Accelerator-rich CMPs: From concept to real hardware. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 169–176, Oct. 2013.

[7] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 225–236, 2010.

[8] J. Cong, et al. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014.

[9] E. G. Cota, P. Mantovani, and L. P. Carloni. Exploiting private local memories to reduce the opportunity cost of accelerator integration. In *Proceedings of the International Conference on Supercomputing (ICS)*, June 2016.

[10] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.

[11] E. G. Cota, et al. Accelerator memory reuse in the dark silicon era. *Computer Architecture Letters*, 13(1):9–12, Jan-Jun 2014.

[12] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 684–689, 2001.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–460, 2012.

[14] C. F. Fajardo, et al. Buffer-integrated-cache: A cost-effective SRAM architecture for handheld and embedded platforms. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 966–971, June 2011.

[15] J. Gaisler. An open-source VHDL IP library with plug & play configuration. *Building the Information Society*, pages 711–717, 2004.

[16] R. Komuravelli, et al. Stash: Have your scratchpad and cache it too. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 707–719.

[17] B. Li, Z. Fang, and R. Iyer. Template-based memory access engine for accelerators in SoCs. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 147–153, Jan. 2011.

[18] K. Lim, et al. Thin servers with smart pipes: Designing SoC accelerators for Memcached. *SIGARCH Comput. Archit. News*, 41(3):36–47, June 2013.

[19] L. Liu, et al. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the International Conference on Parallel architectures and compilation techniques (PACT)*, pages 367–376, 2012.

[20] S. Lotlikar, V. Pai, and P. V. Gratz. AcENoCs: A Configurable HW/SW Platform for FPGA Accelerated NoC Emulation. In *Proceedings of Annual Conference on VLSI Design*, pages 147–152, Jan. 2011.

[21] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):48:1–48:22, Jan. 2012.

[22] P. Mantovani, G. D. Guglielmo, and L. P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2016.

[23] M. Mehendale, et al. A true multistandard, programmable, low-power, full HD video-codec engine for smartphone SoC. In *ISSCC Digest of Technical Papers*, pages 226–228, Feb. 2012.

[24] D. Melpignano, et al. Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications. In *Proceedings of ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1137–1142, June 2012.

[25] S. P. Muralidhara, et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, Dec. 2011.

[26] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. System-level Memory Optimization for High-level Synthesis of Component-based SoCs. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Oct. 2014.

[27] proFPGA Prototyping Systems. http://www.prodesign-europe.com/profpga.

[28] W. Qadeer, et al. Convolution engine: balancing efficiency & flexibility in specialized computing. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 24–35, June 2013.

[29] Y. S. Shao, et al. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, Feb. 2015.

[30] S. K. Shukla, Y. Yang, L. N. Bhuyan, and P. Brisk. Shared memory heterogeneous computation on PCIe-supported platforms. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept. 2013.

[31] G. Venkatesh, et al. Conservation cores: reducing the energy of mature computations. In *Proceedings of Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 205–218, Mar. 2010.

[32] P. Vogel, A. Marongiu, and L. Benini. Lightweight virtual memory support for many-core accelerators in heterogeneous embedded socs. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 45–54, Oct 2015.

[33] F. Winterstein, et al. MATCHUP: Memory abstractions for heap manipulating programs. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 136–145, Feb. 2015.

[34] H.-J. Yang, et al. LMC: Automatic resource-aware program-optimized memory partitioning. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 128–137, 2016.