System Level Synthesis for Virtual Memory Enabled Hardware Threads

Nicolas Estibals, Gaël Deest, Ali Hassan El Moussawi, Steven Derrien
Université de Rennes 1 - IRISA
{nicolas.estibals,gael.deest,ali-hassan.el_moussawi,steven.derrien}@irisa.fr

***Abstract*—Newly introduced ARM-based FPGA platforms enable transparent hardware/software multithreading by providing cache-coherent memory accesses to hardware accelerators. However, the lack of support for virtual memory on the accelerator side hinders the use of off-the-shelf software stacks, such as offered by a Linux-based system, which limits their applicability in a legacy environment. To address this problem, we propose a fully automated high-level synthesis-based source-to-source flow to efficiently support virtual memory in hardware accelerators.**

I. Introduction

FPGA devices combining ARM multi-core processors with FPGA logic (Xilinx Zynq [18] or Altera Soc), have been in mass production for a few years. Albeit designed for the embedded market, they are now luring toward the data center market [13]. Indeed, these platforms often outperform multi-cores and GPUs for only a fraction of their power budget. Despite these benefits, their adoption in the software world is hindered by the gap between hardware and software design flows: complex proprietary CAD tools on one side vs Off-The-Shelf (OTS) open software frameworks and compilers on the other side. FPGA vendors are aware of this problem, and put a great effort into technologies that help design FPGAs using software-like abstractions (HLS tools, OpenCL-based FPGA programming, etc.). Nevertheless, many barriers remain that prevent HPC or embedded experts from fully taking advantage of these devices. One of these reasons is the hardware/software integration stage, which requires designers/developers to have a very deep understanding of 1) the underlying hardware platform, 2) the internals of the underlying operating systems, and 3) the hardware accelerator itself. As of today, this integration is becoming the new bottleneck, either for performance or for productivity (if not both). Again, recent tools such as SDSoC/SDAccel aim at filling this gap, by blurring the hardware/software frontier. This is achieved through a unified programming model akin to conventional multithreading. These tools greatly simplify hardware/software partitioning, but still present some shortcomings when it comes to virtual memory management.

For many embedded or hardware designers, virtual memory support is only considered as a convenience. Indeed, their design is often built from scratch and is not constrained by a large legacy code base. The situation is quite different for data centers and/or high performance computing applications which are built on top on existing, legacy and/or third parties code base which have to be used 'as is'. For such markets, we believe there is a need for efficient and transparent virtual memory support.

In this work, we show how existing hardware/software interfacing techniques can be extended and customized to automatically and efficiently support virtual-to-physical address translation (AT) in hardware threads. We address some of the limitations of previous approaches and present an automatic flow for deriving highly customized address translation hardware. In addition, to ease the integration with existing tools and frameworks based on C-to-hardware approaches, we implemented our techniques as a set of source-to-source transformations targeting C-based HLS tools.

More specifically, our contributions are the following. (i) A survey of existing techniques for virtual memory management in *heterogeneous* multi-core platforms. (ii) Three new or revisited hardware/software virtual memory management techniques, offering a range of cost/performance tradeoffs. (iii) An implementation of these techniques as a set of source-to-source transformations. (iv) An experimental validation of the proposed techniques.

This paper is organized as follows. Section II provides the necessary background information for understanding our approaches, along with a review of related work. These approaches are presented in Section III. Our implementation in a source-to-source compilation flow is exposed in Section IV. In Section V, we evaluate and discuss our prototype implementation from both a qualitative and quantitative point of view. We conclude in Section VI.

II. Background & Related Work

Software *threads* are the defacto parallel programming model in modern multi-cores. Threads follow a shared memory programming model, where the result of a write operation performed by one thread is (almost) immediately visible to the others. As modern processor architectures heavily rely on caches (for instruction and data), this shared memory model must be enforced in hardware through complex and costly cache coherency mechanisms. Because of the wide acceptance of this model, allowing/enabling hardware accelerators to operate as *hardware* threads, sharing a cache-coherent memory hierarchy with software threads and able to run concurrently with them, would be highly beneficial.

A. Cache coherent hardware accelerators

The large bulk of FPGA-based heterogeneous multi-core architectures provide the hardware accelerator with direct access to main memory. However, they do not guarantee memory coherency w.r.t software threads, therefore limiting their usability[1]. It wasn't until recently that this issue was adressed by FPGA vendors, with ARM-based FPGA platforms such as the Zynq and Altera SoC. These platforms offer a direct and coherent access to the shared L2 cache, therefore fully supporting the execution of true *hardware threads*.

For example, in a Zynq system, the FPGA fabric has access to the dual Cortex-A9 shared L2 cache. This is achieved through the Advanced Coherency Protocol (ACP) proposed by ARM. This protocol enforces coherency between the content

[1]Without this mechanism, all the L2 cache lines that have been modified by the software threads must be flushed every time threads have to synchronize, leading to an unbearable overhead.

978-3-9815370-6-2/DATE16/©2016 EDAA

of the private L1 caches and shared L2 cache of the ARM cores. The result of a memory write by the FPGA fabric is therefore immediately visible to the threads running on the ARM processors. The ACP access port on the Zynq offers relatively high throughput (up to 600 MB/s for L2 cache resident data), but suffers from high access latency (approx. 300 ns). To achieve high effective throughput, burst-based memory accesses must be used. These bursts correspond to atomic read/write operations on contiguous data regions in physical memory. They are supported in the Vivado HLS framework through calls to `memcpy()`.

B. Managing virtual to physical address translation

Coherent access to L2 cache is, however, not enough. All advanced operating systems are based on a virtual memory system. In such a system, software threads operate on a virtual address space associated to their parent process. This virtual memory space is decomposed into pages (usually 4 kB-aligned pages), mapped to a physical address in external memory. When running a program, these virtual addresses are translated at run-time into physical addresses by the processor Memory Management Unit (MMU), which relies on a Translation Look-aside Buffer (TLB) to perform the translation. The TLB operates as a cache for virtual to physical page mappings. Whenever the MMU cannot find the corresponding physical page address in its TLB, it walks through the current process page table in main memory.

In contrast, a hardware thread directly manipulates physical addresses. From its point of view, the contiguous virtual memory region manipulated by the software appears as a collection of non-contiguous segments in physical memory. Transparent data sharing between hardware threads and software threads is therefore not possible[2] as is. In the following we discuss various solutions to this problem and point their strength and weaknesses, summarized in Table I.

1) Physically contiguous virtual memory allocation: The difficulty in managing virtual addresses lies in the fact that a contiguous region of b bytes in virtual memory may be fragmented into as many as $\lceil b/P \rceil + 1$ distinct physical memory areas, with P being the system page size. An obvious solution is to enforce that virtual memory regions shared between hardware and software threads also correspond to contiguous regions in physical memory. This approach is supported by Xilinx SDSoC tool through a specific `sd_malloc` function, and was also proposed by earlier work [10]. It suffers from two shortcomings:

- Calls to such memory allocation functions are more likely to fail than a standard malloc, as physical memory may quickly suffer from fragmentation (the problem is much less likely to happen in virtual memory space).
- Relying on a custom `malloc` prevents efficient reuse of legacy code, especially for closed-source binaries. Since all application data cannot be allocated in contiguous physical memory, the existing code base must be reengineered to identify which program objects actually need to be allocated in that way.

Modifying legacy code is costly, error-prone, and causes maintenance issues. Most users hence favor a *wrapping*-based approach and copy data from the legacy data structure to some physically contiguous memory region, so that is can be used by the accelerator. This pragmatic approach goes against the strategy used by high performance device drivers, which try to avoid as possible costly memory copy operations, and aim at *zero copy* implementations.

[2]Operating in a bare metal environment or MMU-less operating system is not possible in a general purpose/legacy environment.

2) Virtual memory-aware hardware accelerators: As the hardware thread cannot access the processor's MMU and TLB, another solution consists in extending the hardware thread with a custom virtual-to-physical AT mechanism. This AT hardware, embedded close to the accelerator, mimics the MMU/TLB. Several earlier works addressed the problem using a technique following this idea. [16], [15], [1], [11].

The weakness of these approaches is the way they deal with TLB misses. On ARM Cortex A9 processors, a TLB miss causes a small to medium performance penalty (10-30 cycles), as the processor has to walk through the global virtual-to-physical translation table. For a hardware accelerator, this kind of event is more difficult to handle. Earlier approaches [16], [15], [7] dealt with TLB misses by issuing a hardware interrupt, and let the software driver update the TLB. Given the high interrupt service latency ($> 8\mu s$) in complex operating systems such as Linux, this solution lead to very important performance overhead (see Section V for a quantitative comparison).

Another solution is to let the hardware directly read the Linux kernel page structure, as proposed [7]. This approach is viable on hardware having coherent access to shared memory. One weakness is that page tables manipulated by the Linux kernel are fairly complex and architecture specific. For example, x86-based machines use a 3-level page table structure, whereas ARM-based architectures use 2-level page tables [3]. But the strongest motivation to avoid this method is performance: as an access to the ACP port costs at least 30 cycles in the FPGA fabric at 100 MHz (300 ns), reading the page table on a ARM machine would lead at best to a latency of 90 cycles (almost 1 μs). This approach therefore incurs a large performance penalty.

C. Related Work

Hardware threads were introduced as a unifying programming model for heterogeneous computing systems [2]. In this abstraction, hardware and software threads can run concurrently, synchronize and share resources in a seamless manner. To support this idea, Peck et al. [12] proposed to extract hardware threads directly from multithreaded C code, HDL generation being delegated to a *High-Level Synthesis* (HLS) compiler. Lübbers et al. [10] introduced ReconOS, a real-time hardware/software OS supporting HDL hardware threads through a standardized interface. These works focused on the issue of thread synchronization, but the problem of *virtual* memory sharing, an essential feature in a legacy environment, was not fully handled. For example, early implementations of ReconOS relied on a uncachable, physically contiguous block of memory, advertised to software threads as a memory-mappable file. An accelerator could hence not directly operate on a data structure allocated by a legacy library.

Vuletic et al. proposed to address this problem by coupling a *Virtual Memory Window*, a page-level cache containing a subset of the process' virtual memory pages, with a TLB [16], [15]. On a TLB miss, the hardware generates an interrupt. The OS then copies the requested page into the VMW and updates the TLB accordingly. Upon page eviction, or at the end of the thread's execution, modified pages are written back to main memory. Performance-wise, their approach suffers from high interrupt service latencies and considerably large data transfers, especially for sparse memory accesses. Moreover, it

TABLE I: Summary of existing approaches

Approach	Support legacy	Scalability	Zero-copy	HW/SW multithreading	Memory access latency	Hardware overhead	Porting effort
[10]	×	×	×	✓	low	none	high
SDSoC	×	×	✓	✓	low	none	high
[16], [15]	✓	✓	✓	×	very high (IRQ)	high	low
[11], [7]	✓	✓	✓	✓	very high (IRQ)	high	low
[9], [1], [14]	✓	✓	✓	✓	high (pagemap walk)	high	low
Our approaches	✓	✓	✓	✓	Trade-off (moderate to low)		low

does not support true hardware/software multithreading, as the user process must be fully stalled to avoid consistency issues. A very similar implementation was recently presented by Ng et al [11], with presumably the same limitations.

Garcia et al. [7] proposed an architecture where the CPU and the accelerator share a coherent memory hierarchy, enabling true HW/SW multithreading. Memory accesses from the accelerator-side are performed through a MMU. TLB misses are still resolved by an interrupt routine, but the possibility of the hardware walking the page table without software intervention is mentioned. This idea was implemented by Lange et al. on a non-coherent memory hierarchy [9]. It is unclear, however, how consistency issues can be handled without large performance penalties.

Agne et al. built upon the work of Lange to implement virtual memory support in ReconOS [1]. Their implementation relies on *delegate threads*, one of ReconOS' key ideas. In ReconOS, each hardware thread is assisted by a delegate software thread, performing all OS interaction on its behalf. This approach allows for much less intrusive modifications to the operating system's kernel. For example, page faults are handled by having the delegate thread repeat the same memory accesses as the hardware thread, forcing the OS to load the corresponding page into memory.

Finally, IBM proposed the *Coherent Accelerator Processor Interface* (CAPI) for POWER8 systems [14]. Each accelerator must be wrapped in a IBM-supplied service layer, responsible for performing address translation by walking the page table in main memory. A proxy on the POWER8 chip is responsible for integrating the FPGA into a coherent memory hierarchy, as a peer to other processors.

Features and performance of these different approaches are summed up in Table I. Our approaches, presented in the next section, aim at providing all the features required for HW/SW multithreading in a legacy environment, at a low performance overhead.

III. Proposed Address Translation Mechanism

For each array accessed by the accelerator, we propose to add a specific AT unit. In this section, we describe three ways to design this unit, representing different cost/performance trade-offs. The reader may look Figure 1 to follow our discussion. Our implementation, based on a S2S tool, is presented in the next section.

A. Hardware-thread specific external memory TLB

Before spawning a hardware thread, we determine a superset of the virtual pages accessed during its execution, along with the corresponding physical addresses. For each array object, we then build a custom shadow virtual-to-physical mapping table: this is a simple table containing the physical address of each page of the array. The addresses of these tables are then forwarded to the hardware thread, enabling faster address translation with a single level of indirection, freeing the hardware threads from directly walking the complex page table data-structure.

Special care must be taken while building these tables: as some pages may be swapped out during program execution, this whole set of pages must be locked in memory before starting the thread, and unlocked upon completion.

Although this solution significantly improves address translation performance, this scheme still involves two external physical memory accesses per virtual memory access, effectively doubling access latency for scalar accesses. This approach is therefore only viable for hardware threads which perform very few memory accesses, or large burst transfers.

B. Hardware-thread specific on-chip memory TLB

To address the aforementioned problem, the TLB can be moved closer to the hardware accelerator, by mapping it to on-chip FPGA memory resources (i.e BRAMs). The address translation can then be performed almost on the fly (BRAMs have only one cycle latency). This approach still requires the external memory shadow tables to be built prior to thread execution. Their content is copied using burst transfers from the physical memory into hardware thread memory before execution. For large datasets, however, this approach is impractical. A 256x256x256 32 bits three-dimensional array implemented using a linearized layout leads to $\lceil 2^{8+8+8+2}/2^{12} \rceil = 2^{14}$ pages in physical memory. This translates into a 64kB on-chip TLB for this array alone. More generally, only 1MB of data can be accessed per BRAM used to store the shadow page table.

C. Hardware-thread specific cached external memory TLB

The previous approaches represent extremal tradeoffs between memory cost and access time. A natural intermediate solution is to use a cache-based mechanism, as in a regular MMU/TLB. The difference is that we only cache a subset of the virtual address space.

This approach is implemented by emulating a cache behavior in C code, which will be recognized as a memory and cache control circuit by the HLS tool. Our source-to-source

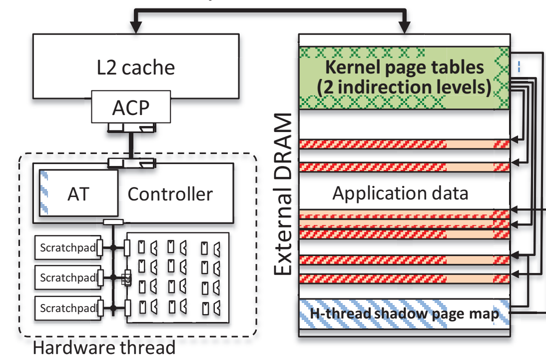

Fig. 1: Illustration to the proposed address translation schemes

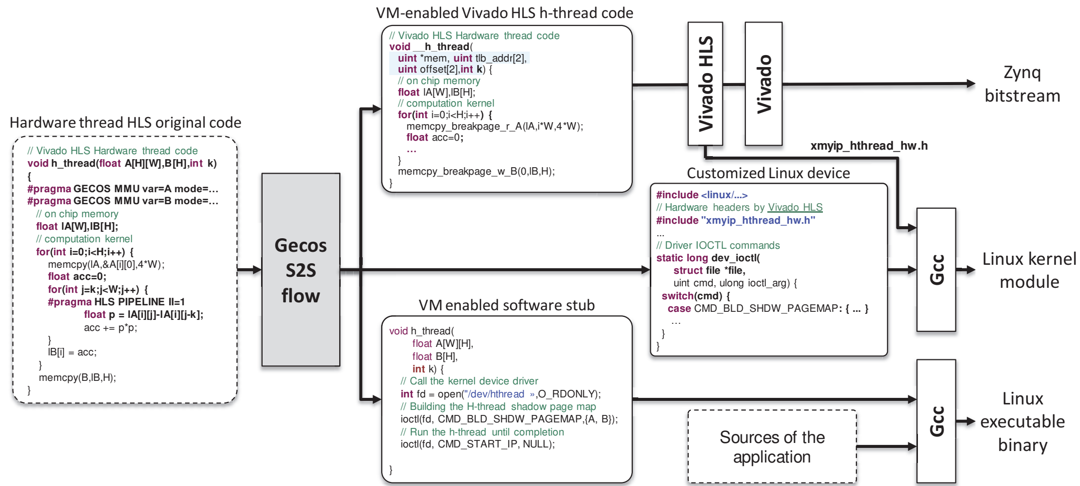

Fig. 2: Proposed source-to-source transformation flow

tool (described in next section) allows the exploration of many replacement policies.

IV. Our Proposed Source-to-source Flow

We propose to add support for virtual memory accesses trough high-level program transformations, directly performed at the hardware thread HLS-C code. Our flow is depicted in Figure 2. It takes as input the source description of a hardware thread, aimed at being synthesized with Vivado HLS [17]. The source code is parsed and analyzed to identify shared array objects. From there, the tool regenerates (i) a modified version of the hardware thread source code, and (ii) a software stub, at the device driver level, that will be used to initialize the structure needed by our address translation mechanism. Our tool is implemented within the source-to-source compiler GeCoS [5].

A. Implementation within a source-to-source compiler

Our goal is to offer seamless integration within existing toolchains such as Vivado HLS and/or SDSoC. To do so, we propose to specify these architectural/device driver modifications directly at the hardware thread source level, before it is synthesized to RTL through Vivado HLS.

From the application source code, we can extract from an analysis the list of arrays/objects that may be accessed by a hardware thread during its execution. In most cases, it is also possible to infer their exact size, or at least derive an upper bound of it (possibly through user directives). This analysis is simplified by the fact that most HLS compilers do not support aliasing (overlapping) between arrays.

Since several design choices are possible, our transformation is user-driven through the use of compiler directives (`#pragma`'s). This approach is widely used in modern HLS tools and is considered by many designers as the most convenient way of controlling the HLS process. The syntax of our compiler directive, along with some of the transformations that it performs, can be found in Table II. It takes as argument a variable identifier and the chosen mechanism for AT support along with optional parameters.

B. Linux kernel module and software interface

Since we rely on a customized page table data structure for each array, we extend the device driver of the accelerator with additional commands to allocate and populate these tables. The device driver is generic, and these commands are exposed to the user through `ioctl` calls. In Figure 2, we show a user application code excerpt performing all the necessary steps to operate the hardware thread.

To construct a specific page table, our device driver enumerates the virtual pages of all shared objects and uses the Linux kernel API provided in `linux/mm.h` and `asm/pgtable.h` to find the corresponding *page table entries* (thanks to the functions `pgd_offset()` to `pte_offset_map()`). These values are then processed through the function `pte_page()` to obtain the corresponding physical addresses. Each page is then locked using the function `mlock()`. More details about the API used by our device driver can be found in [4].

C. Features

1) Managing page breaks in burst-based access: ACP burst (i.e. `memcpy`-based) transfers are mandatory to achieve good memory throughput. They must however be handled with care, as a single `memcpy` operation may cross page boundaries. To address this issue, our source-to-source flow transforms all `memcpy` operations into a sequence of intra-page burst transfers. Assuming 4 kB pages, a transfer of size B requires at most $P = \lceil \frac{B}{2^{12}} \rceil + 1$ address translations.

To perform this decomposition, we use a specific `memcpy_breakpage` template function, specialized for each shared array. An example of such code is provided in Figure 2 (function `memcpy_breakpage_r_A`).

2) Pointers-of-pointers arrays: Our approach supports both linearized and pointers-of-pointers arrays, therefore covering most real-life use cases. In the latter case, the shadow page map becomes a two-dimensional array. An example of this kind of transformation is provided in Table II.

3) True hardware threads: With our implementation, there is no need for a delegate software thread to support the execution of the hardware thread. We thus let the CPUs completely free to execute other threads/applications.

D. Limitations of the tools

As we completely avoid interrupting the OS, our hardware thread cannot provide runtime errors for illegal memory accesses. However, we could add dynamic bound checking in a future version of the tool.

Moreover, as the shadow page map table is generated at the beginning of the hardware thread execution, the shared arrays should not be `free`'d or `realloc`'ed before its completion.

TABLE II: Transformation rules for the cache enhanced hardware thread specific TLB

Code	Transformed code

`typedef unsigned int uint32;`
 	
`void foo(uint32 a[N][M], ...) {`
`#pragma GECOS MMU \`
`var=a \`
`mode=DRAM_TLB`
`a[i][j] = ...}`
	`#define PAGE(addr) (addr>>12)`
	`#define OFFSET(addr) (addr&0xfff)`
	
	`void foo(volatile uint32 * mem,`
	`uint32 tlb_phys_addr_a, uint32 offset_a) {`
	`#pragma HLS INTERFACE m_axi port=mem offset=off`
	`volatile uint32 * mem_tlb_a = (volatile uint32 *) (mem + (tlb_phys_addr_a>>2));`
	`uint32 tmp = ((i*M+j)<<2) + offset_a;`
	`*( mem + ((mem_tlb_a[PAGE(tmp)]+OFFSET(tmp)) >> 2) ) = ...}`

`void foo(uint32 ** a, ...) {`
`#pragma GECOS MMU \`
`var=a \`
`mode=DRAM_TLB\`
`max_dim1=N\`
`max_dim2=M`
`a[i][j] = ...}`
	`void foo(volatile uint32 * mem,`
	`uint32 tlb_phys_addr_a[N], uint32 offset_a[N]) {`
	`#pragma HLS INTERFACE m_axi port=mem offset=off`
	`volatile uint32 * mem_tlb_a[N];`
	`for(int i=0; i<N; i++)`
	`mem_tlb_a[i] = (volatile uint32 *) (mem+(tlb_phys_addr_a[i]>>2));`
	`uint32 tmp = (j<<2) + offset_a[i];`
	`*( mem + ((mem_tlb_a[i][PAGE(tmp)]+OFFSET(tmp)) >>2 ) ) = ...}`

`void foo(uint32 a[N][M], ...) {`
`#pragma GECOS MMU \`
`var=a \`
`mode=LOCAL_TLB`
`a[i][j] = ...}`
	`void foo(volatile uint32 * mem,`
	`uint32 tlb_phys_addr_a, uint32 offset_a) {`
	`#pragma HLS INTERFACE m_axi port=mem offset=off`
	`uint32 * local_tlb_a;`
	`memcpy(local_tlb_a, mem + (tlb_phys_addr_a>>2), NB_PAGE_A*4);`
	`uint32 tmp = ((i*M+j)<<2) + offset_a;`
	`*( mem + ((local_tlb_a[PAGE(tmp)] + OFFSET(tmp)) >> 2) ) = ...}`

`void foo(uint32 a[N][M], ...) {`
`#pragma GECOS MMU \`
`var=a \`
`mode=CACHE_TLB \`
`cache_size=1k`
`a[i][j] = ...}`
	`void foo(volatile uint32 * mem,`
	`uint32 tlb_phys_addr_a, uint32 offset_a) {`
	`#pragma HLS INTERFACE m_axi port=mem offset=off`
	`volatile uint32 cache_tlb_a[CACHE_TLB_A_SIZE];`
	`uint32 tmp = ((i*M+j)<<2) + offset_a;`
	`*( mem + ((HLS_CACHE(cache_tlb_a, PAGE(tmp), mem, tlb_phys_addr_a)`
	`+ OFFSET(tmp)) >>2 ) ) = ...}`

We do not think of this as a large drawback, as this event is a sure sign of a faulty program.

V. Results & Discussion

In the following, we present some use cases for virtual memory-enabled hardware accelerators. We then show the performance of our approach on different application examples. Finally, we compare our results with previous works.

A. Use cases

To illustrate the application of our method to loop nest accelerators, we implemented a (blocked) 2048x2048 matrix product example, exposing moderately complex two-dimensional access patterns, and a tiled 2D stencil kernel operating on a 2000x2000 array (loosely based on the Horn-Schunk optical flow [8] algorithm). For the latter, access patterns are more complex due to the use of an *oblique* tiling execution scheme. We implemented those accelerators for two tile sizes.

We also implemented an FFT accelerator based on the Pease FFT algorithm with a radix of 256 words [6]. In this example, data is accessed in a non-contiguous fashion. Pease FFT was chosen as an illustration of problematic access patterns in which no burst accesses can be found. Implementation results are provided for a FFT of order 256K and 2M.

These three kernels share a common characteristic: their access patterns cannot be efficiently captured with *streaming* interfaces. Besides, the performance of all these applications is very sensitive to the cost of communications (memory bandwidth often being the performance bottle neck). Whenever possible, in order to maximize effective data throughput, memory accesses were coalesced to form burst accesses using the `memcpy()` function supported by Vivado HLS.

B. Experimental setup

In order to quantify the benefits of our approach, we first compare area and performance overheads for our three different methods. For the `CACHE_TLB` version, we chose the size of the cache for the shadow page map table to take exactly 1 BRAM (32kB) as it is the minimal overhead compared to the `DRAM_TLB` solution. Area (slices, BRAMs) and communication cost metrics are shown in Figure 3. Those results were obtained by executing the applications on a Zedboard (Zynq 7020 FPGA SoC) running Xillinux 1.3 (kernel version 3.12). Synthesis and bitstream generation were performed using Vivado 2014.4, targeting a 100MHz frequency (the default frequency of the ACP port). Each accelerator has been equipped with a cycle accurate performance counter in the PL fabric to obtain reliable latency measurements.

C. Performance analysis

In terms of area, an accelerator implemented with the `DRAM_TLB` approach features a negligible overhead and may be compared to the same accelerator operating in an non-virtual memory environment. However, compared to the non-MMU case, scalar memory accesses are considerably slower as twice the same number of memory operations is required. In contrast, the `LOCAL_TLB` approach comes with a non-negligible area cost, but enables seamless virtual memory translation with almost no latency overhead.

In compute-intensive applications that can benefit from burst transfers, such as the matrix product and the 2D-stencil, the latency cost of the `DRAM_TLB` approach is negligible compared to the computation time. It can be concluded that an off-chip shadow page map table is sufficient for such applications, as the `LOCAL_TLB` approach features no significant performance improvement. For other use cases such as the FFT, which

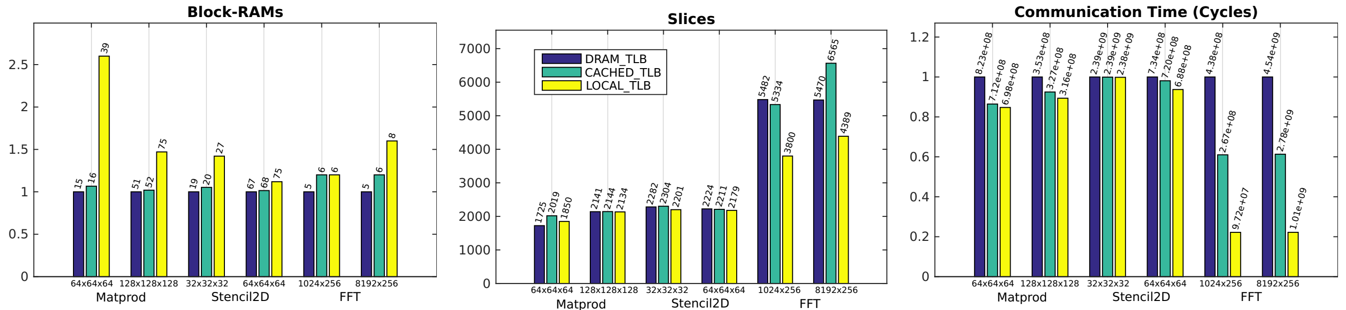

Fig. 3: Implementation results for the different example applications. (Block-RAMs and communication time graphs are given relatively to the DRAM_TLB version. Absolute values are provided on top of the bars.)

features mostly scalar accesses, the CACHE_TLB approach represents an interesting performance vs. cost tradeoff.

It should be noted that our hardware IPs have not been highly optimized. In more finely-tuned parallel architectures, we expect the relative impact of bursts to be even more significant, as the computation/communication ratio would decrease.

D. Comparison with previous work

Latest work [11] tackled the problem using a generic hardware MMU in the FPGA logic. Interestingly, the authors acknowledge that interrupt servicing delays caused by TLB misses have a very negative impact on performance.

To assess the benefits of our approach, we also compared our results with that of Ng et al. prior work [11], that were based on MMU/TLB approaches. Following the same experimental setup, we considered a 512 entries (direct-mapped) TLB stored in the accelerator, and simulated the execution of our matrix product algorithm in that configuration. We could observe TLB miss rates varying between 12% for the smallest tile size (64x64x64) to 24% for 128x128x128. We also experimentally measured the interrupt servicing delay between the hardware and the Linux kernel (assuming a system without any other workload) and observed delays varying between 800 cycles and 2000 cycles for the IP (at 100 MHz). For an average burst size of 128 words, we obtain a significant drop in communication performance (in the range of 10 to 40%).

In most related work, [15], [16], [9], [1], the hardware used for experiments was very different from Zynq-based ones. Moreover, those works implemented caches for data transfers, integrated in their cost evaluations. Therefore, it is hard to provide any numeric comparison.

Given our results, we believe that application-specific hardware threads should also use application-specific memory management techniques, as made practical and efficient using a source-to-source compiler. We have shown how domain/application-specific information could be leveraged to derive a low overhead, efficient support for the memory system of heterogeneous computing platforms.

VI. Conclusion

In this work, we have proposed a new approach for supporting cache coherent hardware threads in the context of virtual memory-based operating systems such as Linux. Our approach offers several low overhead hardware/software solutions to support page fragmentation and virtual-to-physical address translation for the accelerator. Our proposal is implemented in a source-to-source compiler framework and was validated on three representative kernels (matrix product, 2D stencils and FFT). Experimental results show that the proposed technique can offer improved performance (for a similar area cost) over state of the art techniques.

References

[1] A. Agne, M. Platzner, and E. Lübbers. Memory virtualization for multithreaded reconfigurable hardware. In *Field Programmable Logic and Applications (FPL)*, pages 185–188, Sept 2011.

[2] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro*, 24(4):42–53, July 2004.

[3] ARM. *Cortex-A9 Technical Reference Manual*. Revision: r2p2.

[4] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[5] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. GeCoS: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM)*, pages 100–105. IEEE, Sept 2013.

[6] F. Franchetti and M. Püschel. *Encyclopedia of Parallel Computing*, chapter Fast Fourier Transform. Springer, 2011.

[7] P. Garcia and K. Compton. A reconfigurable hardware interface for a modern computing system. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 73–84, 2007.

[8] B.K.P. Horn and B.G. Schunck. Determining optical flow. Technical report, Cambridge, MA, USA, 1980.

[9] H. Lange and A. Koch. Low-latency high-bandwidth HW/SW communication in a virtual memory environment. In *Field Programmable Logic and Applications (FPL)*, pages 281–286, 2008.

[10] E. Lübbers and M. Platzner. ReconOS: multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009.

[11] Ho-Cheung N., Yuk-Ming C., and H.K.-H. So. Direct virtual memory access from FPGA for high-productivity heterogeneous computing. In *Field-Programmable Technology (FPT)*, pages 458–461. IEEE, Dec 2013.

[12] W. Peck, E. K. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews. Hthreads: A computational model for reconfigurable devices. In *Field Programmable Logic and Applications (FPL)*, pages 1–4, 2006.

[13] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiaoi, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[14] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1), 2015.

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

[16] M. Vuletic, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design Test of Computers, IEEE*, 22(2):102–113, March 2005.

[17] Xilinx. *Vivado Design Suite User Guide High-Level Synthesis (UG902)*. v2015.2.

[18] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*. v1.10.