COREFAB: Concurrent Reconfigurable Fabric Utilization in Heterogeneous Multi-Core Systems

Artjom Grudnitsky, Lars Bauer, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{grudnitsky, lars.bauer, henkel}@kit.edu

ABSTRACT

Application-specific accelerators may provide considerable speedup in single-core systems with a runtime-reconfigurable fabric (for simplicity called “fabric” in the following). A reconfigurable core, i.e. processor core pipeline coupled to a fabric, can be integrated along with regular general purpose processor cores (GPPs) into a reconfigurable multi-core system with widely improved system performance. As most applications only use a fraction of the available fabric at a time, making the fabric usable by the GPPs (in addition to the reconfigurable core) in such a multi-core system is desirable. Existing work focused on algorithms that decide the amount of fabric that is assigned to each core in a multi-core system. However, when multiple cores access the fabric simultaneously, they are either limited to serialized fabric access or, when parallel access is supported, the size of the fabric share assigned to a core is inflexible and tends to be over- or undersized for the running application, thereby not efficiently utilizing the fabric. We propose a novel approach that allows GPPs to access the fabric of the reconfigurable core and that enables concurrent fabric utilization on-the-fly through merging fabric accesses from different cores at runtime. Compared to state-of-the-art, our approach improves performance of the GPPs in a reconfigurable multi-core system by 1.3× on average, without reducing the performance of the reconfigurable core.

Categories and Subject Descriptors

[Computer systems organization]: Multicore architectures

General Terms

Design, Algorithms, Performance

Keywords

Reconfigurable Processor; Heterogeneous Multi-Core; Reconfigurable Fabric Sharing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESWEEK’14, October 12–17 2014, New Delhi, India.
Copyright 2014 ACM 978-1-4503-3050-3/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656106.2656119.

1 Introduction

General purpose processor (GPP) cores connected to a reconfigurable fabric (called reconfigurable cores in the following) allow execution of computationally intensive application parts (i.e. kernels) in hardware, resulting in improved execution speed (speedups of 3.6×–7.2× for video encoding applications are reported in [1, 2, 3]). The fabric can be loaded with application-specific accelerators providing speedup for a diverse set of kernels. This work focuses on reconfigurable cores where a fine-grained reconfigurable fabric (an embedded FPGA) is coupled to the pipeline of the core as an additional functional unit (i.e. tightly coupled). This allows low-latency access to the fabric and therefore the ability to attain speedup even for applications consisting of kernels with short durations (e.g. in the range of 10–100 cycles when executed on the fabric). The instruction-set architecture (ISA) of the GPP is extended with Special Instructions (SIs) that are used by applications to access accelerators on the fabric and thereby execute computationally intensive kernels on it.

The speedup that can be attained by executing an SI on the fabric instead of executing the equivalent ISA instructions on the pipeline, depends on the inherent parallelism of the respective kernel and the amount of fabric used for its implementation. For example, exploiting its full parallelism and thereby attaining its highest performance, the JPEG encoding kernel requires 5× as much area as a kernel performing SHA cryptographic hashing [4]. Assuming that (i) the hardware implementation of the JPEG and SHA kernels require 100% and 20% of the fabric area, respectively, (ii) a reconfigurable core executes SHA and JPEG in alternation for the same amount of time, and (iii) the fabric is reconfigured when the application switches between JPEG and SHA, then the reconfigurable core will use 60% of its fabric area on average. We call this effect fabric underutilization, i.e. at some points in time resources on the fabric are available but idle. Fabric underutilization results in inefficient use of computational resources of the system. Reconfigurable cores are advantageous in aspects like flexibility, runtime adaptivity, and time-to-market. However, when targeting a specific application, then reconfigurable cores are less area-efficient compared to ASICs and ASIPs that are statically optimized for the target application. Improving area efficiency of reconfigurable cores by reducing fabric underutilization can offset this disadvantage and smooth the way for wider adoption of reconfigurable cores.

Fabric underutilization can be reduced by executing multiple kernels concurrently on the fabric. This can be realized by integrating the reconfigurable core into a multi-
core system and allowing multiple cores to access the fabric, i.e. allowing non-reconfigurable cores to offload their kernels onto the fabric. Multi-core systems with reconfigurable fabrics have been proposed (such as [5, 6, 7]), but they do not provide significant utilization improvements compared to a reconfigurable single-core, as (i) they use serialized fabric access, where at a given point in time only one core can access the fabric, (ii) partitioning of the reconfigurable fabric is restricted (e.g. only equal fabric shares for each core), or (iii) the fabric has a very simple structure (e.g. no ports for direct memory access), which severely limits the kernels that can be implemented on it.

To improve performance of reconfigurable multi-cores while addressing the above mentioned issues, we propose COREFAB, a technique to allow GPP cores in a heterogeneous multi-core system remote execution of Special Instructions (Remote-SIs) on the fabric of the reconfigurable core. The novel contributions of COREFAB are:

1. the protocol for accessing the reconfigurable fabric in a heterogeneous multi-core system,
2. the concept for merging simultaneous fabric accesses on-the-fly, allowing concurrent utilization of the fabric by different cores, and
3. the required hardware architecture for arbitrator and merging multiple concurrent accesses.

Compared to state-of-the-art fabric sharing techniques, COREFAB improves the performance of the GPP cores in a reconfigurable multi-core system by 1.3x on average, without reducing performance of the reconfigurable core.

Paper organization: Section 2 briefly reviews related work. The architecture of the underlying multi-core reconfigurable system is described in Section 3 and the problem definition is provided in Section 4. Our COREFAB approach for concurrent fabric utilization is presented in Section 5 and evaluated in Section 6. Section 7 concludes the paper.

2 Related Work

We will first provide a short overview of reconfigurable single-cores and then focus on reconfigurable multi-core architectures that allow sharing the fabric between the cores.

An overview of reconfigurable cores is provided in [8, 9, 10]. In this work we focus on architectures where the pipeline is tightly coupled to a fine-grained reconfigurable fabric, such as those presented in [11, 12]. We envision that such a system will use an embedded FPGA, where the non-reconfigurable components and interconnect are fabricated as an ASIC and reconfigurable logic is used only for the fabric, similar to the architecture in [13].

A multi-core architecture that features reconfiguration is presented in [14]. A reconfigurable interconnect between the Execute stages of the GPPs of a 4-core system allows switching the processor into specialized modes, such as Wide-Word-ALU mode, where up to four ALUs can be combined for SIMD operations. Ipek et al. present an architecture where simple cores in a multi-core system can be fused on-the-fly into more complex cores [15]. These architectures feature reconfiguration between processor cores, but do not provide reconfigurable accelerators which is the topic of this work.

Yan et al. present a multi-core architecture that uses a fine-grained reconfigurable fabric in [6]. The fabric is partitioned into reconfigurable processing units (RPUs) of equal size. The cores and the fabric are connected via a crossbar, allowing any core to access any RPU. However, the fabric can not access main memory directly (instead a GPP core must perform the access and provide data to the RPU) and there is no communication between RPUs. This impairs memory-intensive kernels, as memory accesses are handled by a GPP core and the implementation of the kernel is restricted to the size of one RPU. Fabric area distribution in multi-core systems is discussed in [7], where the fabric is assumed to be accessible by any of the cores. Their focus is on deciding the share of the fabric that can be used by a core, which is determined using a minority game based approach. However, details on the interfaces between the fabric and the cores, the fabric structure (e.g. how many independent memory ports are assumed to be available) and how it is possible to execute SIs concurrently, are not provided. COREFAB provides novel protocol, concept, and hardware that actually allow a flexible way for concurrent fabric accesses for the first time and that are independent of the fabric distribution.

In [16] a reconfigurable fabric is shared between multiple cores in a shared-memory system. The work focuses on online-scheduling of tasks that use the fabric by making future reservations of portions of the fabric for tasks. The scheduler also considers fabric resource conflicts between tasks (similar to our approach), however conflicts lead to SI rejection, while our COREFAB approach stalls one of the conflicting SIs and retries concurrent execution in the next cycle. Chen et al. [17] no longer reject conflicting requests, but manage them. They propose a reconfigurable fabric shared by two cores with the focus on fabric distribution. Access to the fabric is exclusive to a single core at any point in time. If multiple cores wish to access the fabric at the same time, a round-robin arbiter determines the core which is granted access in a given cycle. Contrary, COREFAB allows simultaneous access of multiple cores to the fabric. In the evaluation Section 6, we compare COREFAB to the Gap-Filling method that uses a similar fabric access management than [17].

In [5] reconfigurable fabrics are loosely coupled to multi-core clusters in many-core systems. Cores share the fabric either in a time-multiplexed fashion, or the fabric is partitioned into equal shares and each core uses a private fabric share. While private sharing is a viable approach for simple SIs that perform arithmetic/logic operations, complex operations that require memory access would suffer if memory bandwidth would be reduced due to partitioning. Furthermore, partitioning allows only equal-sized fabric shares for each core, thus fabric underutilization is not reduced efficiently.

The commercial Xilinx Zynq-7000 SoC consists of a dual-core ARM with peripherals and a Xilinx 7-Series reconfigurable fabric. Multiple interfaces are provided for accessing the fabric from the dual-core system, however, even when using the interface with the lowest latency (i.e. ACP), using the fabric at instruction-level parallelism granularity is not recommended [18]. Furthermore, while the fabric as a whole can be accessed by both cores, no system is provided for partitioning the fabric between both cores or handling simultaneous accesses to different parts of the fabric. Therefore the Zynq can be used as a platform for the proposed approach, if the typical duration of the kernels run on the fabric is long enough to offset its fabric access latency.
3 Reconfigurable Multi-Core System

Figure 1 shows a heterogeneous reconfigurable multi-core system (generalized from [5, 6, 7, 16, 17]) and highlights the novel contributions of COREFAB. In this example, the heterogeneous multi-core system (shaded blue) consists of three GPP cores and one reconfigurable core connected to the system bus. Additionally, an SRAM scratchpad memory, DDR controller, Ethernet core, and other peripherals are connected to the system bus.

The fabric consists of reconfigurable processing units (RPUs) that can be loaded with application-specific accelerators at runtime, and RPU links that allow data transfer between accelerators. This allows implementing complex SIs that use multiple (potentially different) accelerators in parallel. The fabric is also capable of accessing the memory hierarchy (caches and off-chip DRAM, not shown in the figure) and the SRAM scratchpad using high bandwidth memory ports that are connected to the RPU links, allowing data transfer from/to accelerators. Using memory ports allows fetching input data and writing back output data directly from the fabric with low latency and high bandwidth, as the core pipeline is not involved.

The SI Execution Controller determines the fabric operation, i.e. the operation of already loaded accelerators (an ALU-like accelerator can perform different operations), memory ports (load or store mode, address, data), and RPU links (to establish data transfer between RPUs and/or memory ports). Most SIs are multi-cycle operations, as shown in the simplified example SI in Figure 2a): in cycle 0 input data is loaded from the SRAM scratchpad into memory port 0; in cycle 1 input is sent over link 1 to RPU 3 which starts processing it; in cycle 2 RPU 3 finishes processing this data set (it might actually take multiple cycles and for complex operations multiple RPUs might be involved); in cycle 3 results are sent over link 3 from RPU 3 to memory port 0; in cycle 4 the results are stored in memory (note: typically computation is pipelined, i.e. during data processing the next input data is fetched). Most real SIs are far more complex, performing multiple memory accesses using an address pattern generator and stream data through multiple RPUs, thus often utilizing multiple RPUs/memory ports in one cycle. Thus, the fabric operation is SI specific and typically changes in each cycle of the SI execution. Note that the fabric operation determines the operation of and communication between RPUs and memory, but not the RPU configuration (i.e. the accelerator loaded into an RPU). The RPU configuration does not change during an SI execution as reconfiguration of an RPU takes up to several milliseconds [12].

The fabric operations for an SI are encoded in a micro-program (similar to the micro-programs in MOLEN [11]). The SI Execution Controller reads one micro-program operation (μOp in the following) per cycle from the SI micro-program memory (see Figure 1) and controls the fabric according to this operation. For the SI in Figure 2a), each of the 5 cycles would be encoded by one μOp, with the μOp for cycle 1 shown in Figure 2b). Micro-programs can be generated offline and stored in DRAM, with the OS transferring them to the SI micro-program memory as necessary.

When implementing such a heterogeneous multi-core architecture as an SoC, only the RPUs need to be implemented as an embedded reconfigurable fabric, the rest of the system can be implemented as an ASIC on the same chip.

4 Problem Description and Solution Overview

We first describe the fabric underutilization effect, which we aim to reduce in order to improve system performance.
Next, we provide a formulation of our proposed solution, SI merging, which allows concurrent use of the fabric by multiple cores. The hardware for remotely accessing the fabric by a GPP core is described in the next chapter.

### 4.1 Fabric Underutilization

The goal of this work is to improve performance of the GPPs in a reconfigurable multi-core system without sacrificing performance of the reconfigurable core. In order to do this without incurring high area overhead, underutilization of computational resources on the fabric needs to be reduced. As described in Section 3, the fabric consists of three types of resources: RPUUs, RPU links, and memory ports. Underutilization occurs if at any point in time not all computational resources (RPUs or memory ports) are used. RPU links are excluded from this definition, as they are not a computational resource. Thus, there are two possible reasons for underutilization: accelerator underutilization and memory port underutilization. For the following definitions we assume that an SI is being executed on the fabric. If no SI is running, the fabric resources are not used at all, and fabric underutilization is at its maximum.

As noted in [4], synthesizing different kernels will generally result in different area requirements, i.e. different amounts of accelerators. **Accelerator underutilization** occurs if a kernel does not use all accelerators available on the fabric because the fabric was optimized for larger kernels or because—even though the kernel could use all accelerators—it is not beneficial to do so. Most kernels provide diminishing performance improvement as the area available for their implementation increases. In a fine-grained reconfigurable fabric the time to load a kernel onto the fabric depends on the area used by its implementation. Thus, even though a small performance improvement may be provided by a larger kernel implementation, it may be negated by the increased reconfiguration time required to load this implementation onto the fabric.

Formally, consider an SI micro-program consisting of $K$ $\mu$Ops. Let $N$ be the set of RPUs on the fabric and let $a_{k,i}$ equal ‘1’ if RPU $i \in N$ is used in $\mu$Op $k \in [1,K]$. **Accelerator underutilization** exists if the constraint in Eq. (1) is satisfied, i.e. there is at least one accelerator that is not used during the entire SI execution.

$$\forall i \in N: \sum_{k=1}^{K} a_{k,i} = 0 \quad (1)$$

**Memory port underutilization** occurs if at least one memory port is idle at any point during execution of an SI. Table 1 shows the memory port utilization for several SIs running on a fabric with two 128-bit wide memory ports (same experimental setup as detailed in Section 6.1). Many SIs do not consistently exploit the available memory bandwidth to full capacity. Ref. [19] analyzes memory accesses of some representative algorithmic patterns, concluding that only about half of them are memory bound. SIs implementing other kernels cannot benefit from the full memory bandwidth available in the fabric. Another reason is that SIs often follow the following pattern: (i) perform a burst read from external memory to get input data, (ii) process input data using one or multiple accelerators, (iii) write back result data. During (ii) no memory traffic is performed, leading to underutilization of the memory port.

### Table 1: Examples of Memory Port utilization in Special Instructions

<table>
<thead>
<tr>
<th>SI</th>
<th>Utilization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmm</td>
<td>60%</td>
<td>Multiplication of 32\times32 floating point matrices</td>
</tr>
<tr>
<td>sad</td>
<td>47%</td>
<td>Sum of Absolute Differences on a 16\times16 block</td>
</tr>
<tr>
<td>sha</td>
<td>43%</td>
<td>SHA-1 hashing</td>
</tr>
<tr>
<td>dct</td>
<td>33%</td>
<td>Discrete Cosine Transform on a 4\times4 block</td>
</tr>
<tr>
<td>adpcm</td>
<td>25%</td>
<td>Encoding of audio data to ADPCM format</td>
</tr>
<tr>
<td>aesenc</td>
<td>10%</td>
<td>AES Encryption</td>
</tr>
</tbody>
</table>

Formally, memory port underutilization can be defined as follows: let $M$ be the set of memory ports available on the fabric and $m_{k,j}$ equal ‘1’ if memory port $j \in M$ is used in $\mu$Op $k$. Then memory port underutilization exists, if the constraint in Eq. (2) is satisfied, i.e. during execution of an SI, there is at least one $\mu$Op where not all memory ports are used.

$$\exists k \in [1,K]: \sum_{j \in M} m_{k,j} < |M| \quad (2)$$

An SI micro-program underutilizes the fabric if Eq. (1) or Eq. (2) is satisfied. If no underutilization exists or if it occurs only to a negligible degree, then the fabric already has high area efficiency. Our proposed COREFAB approach is optimized for scenarios with middle to large underutilization.

### 4.2 Merging of Fabric Accesses

To reduce fabric underutilization, COREFAB merges $\mu$Ops of SIs of different cores on-the-fly, to allow their concurrent execution. In this section, the prerequisites for merging will be formalized.

We assume that resources only support exclusive utilization, i.e. in any cycle a particular resource instance may be used by at most one SI. For merging of SIs we define fabric resources $R$ as computational resources (RPUs $N$ and memory ports $M$) and communication resources (RPU links $L$):

$$R = N \cup M \cup L \quad (3)$$

If multiple SIs shall run in parallel on the fabric, it needs to be ensured that there are no resource conflicts between the SIs, i.e. no two SIs $x$ and $y$ that are executed concurrently on the fabric may access the same resource in a given cycle.

Let SI $x$ aim to execute its $\mu$Op $k_x$ in the same cycle as SI $y$ aims to execute its $\mu$Op $k_y$. Furthermore, let $R_{k_x}$, $R_{k_y}$ correspond to the set of resources required on the fabric by $k_x$ and $k_y$, respectively. Then, the resource requirements of the $\mu$Ops of both SIs need to be disjoint in the current cycle to allow their parallel execution on the fabric:

$$R_{k_x} \cap R_{k_y} = \emptyset \quad (4)$$

This allows the operations to be combined into one **merged** operation, which is the disjoint union of the requirements of both operations (Eq. (5)).

$$R_{k_x} \bigcup R_{k_y} = \begin{cases} R_{k_x} \bigcup R_{k_y} & \text{if Eq. (4) satisfied} \\ R_{k_x} & \text{otherwise} \end{cases} \quad (5)$$
The protocol used for this communication.

5 Concurrent Fabric Utilization

COREFAB increases fabric resource utilization by allowing concurrent execution of multiple SIs. To keep the complexity of the required hardware low, we limit the implementation to two concurrently running SIs. One SI is the Primary-SI, which is always issued by the reconfigurable core that is directly coupled to the fabric (see Figure 1). The other SI can be issued by any other core in the system and is called the Remote-SI. To allow execution of Remote-SIs, additional hardware is required for the reconfigurable multi-core system (red shaded modules in Figure 1), i.e. the Fabric Access Manager for providing non-reconfigurable cores access to the fabric, the SI merger for merging μOps of the Primary-SI and Remote-SI, and the Remote-SI micro-program memory.

μOps are stored in a dedicated on-chip SRAM memory, which is connected to the system bus and the SI Execution Controller. If the same memory would be used for Remote-SIs, an additional read port would need to be added to allow retrieval of μOps for both Remote-SIs and Primary-SIs in the same cycle. This is costly in terms of area, so we propose to add a smaller dedicated micro-program memory for Remote-SIs instead. The size of the memory is a system design parameter which depends on the amount of different SIs executed by the GPPs.

A GPP core that wishes to offload a kernel to the fabric, will issue Remote-SI requests to the Fabric Access Manager (FAM), which will interact with the SI Execution Controller to allow co-execution of a Remote-SI and the currently running Primary-SI. Execution of a Remote-SI involves communication between the FAM and a GPP core for the Remote-SI requests and operand/result transfer. The corresponding protocol is described in the next section.

The SI merger merges μOps from Primary-SI and Remote-SI on-the-fly. When no Remote-SI is executed, the SI merger is in single mode and forwards the μOps from the Primary-SI micro-program memory to the SI Execution controller. Upon start of Remote-SI execution, FAM sets the SI merger into concurrency mode and a Primary-SI μOp k_x and Remote-SI μOp k_y are fetched from the respective memories. The SI merger tests if the operations are disjoint (Eq. (4)) by AND-ing those bitslices in k_x and k_y that encode the use of a particular SI. The merged μOp is then generated according to Eq. (5). If k_x and k_y are not disjoint, the Remote-SI is stalled for one cycle, and the merged operation directly corresponds to k_x. If the operations are disjoint, the merged μOp is generated by OR-ing k_x and k_y. The SI merger has a program counter for the Primary-SI and Remote-SI micro-programs, which is not incremented if an SI needs to be stalled.

5.1 Fabric Access Manager

Executing a Remote-SI proceeds as follows. A GPP core sends a Remote-SI request to the FAM ①. The request encodes which accelerators are required at which location. The FAM checks if the required accelerators are available on the fabric ②, ③. If they are not, the Remote-SI cannot be executed and thus the GPP is notified ④ and executes the kernel containing the Remote-SIs as regular software code. If the required accelerators are available, the GPP is notified ⑤ and transfers the operands (1 per cycle ⑥) for the Remote-SI to the FAM. The FAM starts execution of the Remote-SI ⑦ by sending the operands to the fabric and instructing the SI execution controller to switch into concurrency mode. The fabric then proceeds with co-execution of the Primary-SI and Remote-SI ⑧. When the Remote-SI is finished, FAM notifies the GPP, sets the SI execution controller into single mode and transfers the SI result to the GPP ⑨. If the Primary-SI is finished while a Remote-SI is still running, the GPP proceeds as if a Primary-SI was still running, i.e. the SI execution controller stays in concurrency mode. The only difference is that as upon termination of the Primary-SI, the SI merger sets the Primary-SI program counter to NONE, thus only Remote-SI μOps are fetched until a new Primary-SI starts.

Figure 4 shows the implementation of the FAM. Each GPP has a channel to the FAM with an input multiplexer selecting the GPP which may co-execute on the fabric. If a Remote-SI request arrives while a different GPP is co-executing (i.e. FAM FSM is not in IDLE state), then the Remote-SI request is rejected. In case of simultaneous Remote-SI requests, a fixed priority conflict resolution is performed, where the core ID is used as the priority. The Remote-SI protocol is implemented by the FAM FSM. Transition edges either are annotated with the edge labels from Figure 3 or are unlabeled, meaning that the state transition occurs in the following cycle, after the actions of the current state are finished. Communication with the SI merger is performed via the mode and SI state signals. ‘Mode’ sets the SI merger into concurrency mode when FAM enters state REQ_RCVD and into single mode when entering state IDLE. ‘SI state’ is used to signal that a Remote-SI has finished execution and to transfer result data from the fabric.
SI micro-programs the memory ports with the lowest IDs. The strategy used for memory port accesses: for Primary-SI, the highest possible IDs to Remote-SI micro-programs. A similar strategy is used for memory port accesses: for Primary-SI and Remote-SI micro-programs differently, by an SI. A way to reduce resource conflicts is to generate specialized micro-programs for Remote-SIs and (ii) rewriting micro-programs for the Memory Stage (ME) of the GPP, thereby no stalling is required. In total, the protocol for executing Remote-SIs incurs an additional latency of 1 cycle for the pipeline from Figure 1.

5.2 Reducing conflicts during merging

Merging fabric accesses is not always possible if their resource requirements conflict, i.e. if Eq. (4) is not satisfied. Resource conflicts are handled by stalling the Remote-SI, thus resource conflicts reduce the performance improvement of COREFAB, but there are ways to reduce the number of conflicts. To resolve RPU conflicts, each core is assigned a number of RPs, depending on the requirement of the application running. Such assignment can be done statically, or managed dynamically by the operating system (e.g. Ref. [7] as discussed in related work) and is not the focus of this work. To reduce the number of RPU link conflicts and memory port conflicts, there are two possibilities that are discussed in the following: (i) generating specialized micro-programs for Remote-SIs and (ii) rewriting \(\mu\)Ops on the fly.

Micro-programs determine which fabric resource is used by an SI. A way to reduce resource conflicts is to generate Primary-SI and Remote-SI micro-programs differently, in such a way that the chance for conflict occurrence is low. RPU link conflicts are minimized by allocating the lowest possible RPU link IDs to Primary-SI micro-programs and highest possible IDs to Remote-SI micro-programs. A similar strategy is used for memory port accesses: for Primary-SI micro-programs the memory ports with the lowest IDs are used while for Remote-SI micro-programs the highest IDs are used first.

To rewrite \(\mu\)Ops on the fly, the SI merger would need to be extended significantly. Upon detecting a conflict (by comparing the appropriate bitslices of the current Primary-SI and Remote-SI \(\mu\)Ops), the Remote-SI operation would need to be modified in the same cycle to resolve this conflict. For example, consider that at a particular cycle, Primary-SI and Remote-SI require the same RPU link for transferring data. In a fabric where link IDs are coded as 20-bit bitslices in the \(\mu\)Ops, upon detecting a conflict the SI merger would need to look for a free RPU link by comparing both 20-bit slices with the unused link ID. It would then use this unused link ID in the appropriate 20-bit bitslice in the merged operation. Furthermore, the bitslices for the source and target (e.g. RPU and memory port) of the RPU link would need to be updated so they used the new unused link. The advantage of micro-program rewriting is that the same micro-program could be used for a Primary-SI or Remote-SI and existing conflicts are actually resolved, while specialized micro-program generation only reduces the chance for conflicts to occur. The disadvantages of this method are larger area of the SI merger (increases with the number of links) and an increased critical path, thus reducing system frequency and thereby performance.

Stalling Remote-SIs reduces the predictability of the system. To counteract this, several methods can be employed, such as allowing the FAM to be configured into an “primary-only” mode, which completely prohibits Remote-SI execution, causing SIs from the non-reconfigurable cores to be executed in software. Less drastic measures could monitor Remote-SI latency over a time-frame and reject Remote-SIs if a configurable threshold for Remote-SI latency is exceeded. The GPP that performed the Remote-SI would then execute the SI kernel in software, putting an upper bound on SI latency.

6 Experimental Evaluation

6.1 Setup

COREFAB was evaluated for a multi-core system based on the Gaisler LEON3 multi-core SoC [20]. The SoC consists of four SPARC V8 [21] in-order cores connected to an AMBA AHB bus and additional components such as scratchpad memory, Ethernet and DDR controller. To extend the SoC with a reconfigurable core, the 7-stage pipeline of Core 1 was modified as follows: i) the Decoder stage was extended to recognize SIs, ii) the Register Access stage checks if a detected SI can be executed on the fabric and throws a trap for later emulation in software if not (the emulation code is part of the application, the trap handler glue code is part of the OS), iii) the Execute stage allows stalling the pipeline, passing operand data to the SI Execution Controller, reading back result data from the SI Execution Controller, and un-stalling once it signals that SI execution on the fabric is finished. The interface and protocol between the pipeline and the SI Execution Controller is similar to the optional Divider Unit used in the LEON3. Ref. [12] provides an overview of the reconfigurable core architecture and runtime system.

The fine-grained reconfigurable fabric, SI Execution controller, SI micro-program memory and a DMA-capable accelerator loader for reconfiguring RPs were added to the SoC to allow execution of SIs on the fabric. The fabric is con-
Algorithms: Audio Video Crypto Crypto Image Image

Based on the LEON3 GPP on an Virtex-5 LX110T. With 50 MHz, with RPU dimensions of $20 \times 5$ CLBs and the time to reconfigure an RPU of 274 μs. Resource utilization details are provided in Table 3. The increased amount of BlockRAMs required for the reconfigurable core compared to the LEON3 is due to a 5-port register file (4 read, 1 write) instead of a regular 3-port register file (2 read, 1 write) used in a LEON3. The increased number of ports allows providing operands fast to an SI. The large fabric size compared to the GPP size is due to the flexibility of links that connect the RPUs, which requires a large number of multiplexers, resulting in a large area when implemented on an FPGA. The multiplexers are not part of the RPUs, and thus are not reconfigurable. According to [23], multiplexers can be implemented in a more area-efficient manner in ASICs compared to FPGAs, thus in an ASIC implementation of a reconfigurable core, where only the RPUs are runtime reconfigurable, the area overhead would be smaller compared to our prototype, where everything is implemented on an FPGA. For faster and wider evaluation, the COREFAB extensions were also integrated into a cycle-accurate SystemC model of this SoC. There, we do not encounter the area limitation of the prototyping system and thus we use a quad-core system (dual-core on the prototype) and vary the number of RPUs between 5, 10 and 15 (5 on the prototype).

For benchmarking we use workloads consisting of applications representing different computational domains: an in-house H.264 Video Encoder, SHA hashing, AdPCM encoding, AES encryption, JPEG Decoding and SUSAN image processing from the MiBench suite [24]. The characteristics of these applications are detailed in Table 2 and the application speedup (depending on fabric size) is shown if Figure 5. SHA, AdPCM and AES each have one kernel (although of different complexity) that is accelerated by one Special Instructions (SIs) each. The H.264 Video Encoder has a very dynamic execution behavior: to encode a frame, three different kernels (implemented by several executions of altogether 9 different SIs) are used, with the SIs of the second kernel ("encoding engine") being highly dependent on the type of frame (i.e. input data dependent). JPEG and SUSAN also have multiple kernels, although they exhibit less dynamic behavior than H.264.

Table 2 also shows the inter-SI gap characteristic that is defined as the time spent running non-SI code between two subsequent SI executions. The ratio of inter-SI gap to the SI Latency is of particular interest and will be used to analyze the results. If the ratio is large enough, then fabric accesses by different cores can be serialized efficiently (as done by the Gap-Filling strategy described below), thus reducing the benefit provided by concurrent fabric access. For AdPCM, H.264 and SHA the inter-SI gaps are small, i.e. the system spends little time running non-SI code between two subsequent SI executions, compared to AES, JPEG and SUSAN. The reason for the large inter-SI gaps for SUSAN is that the output of most SUSAN-SIs requires control-flow intensive post-processing, which is performed on the GPP.

The workloads composed out of these applications are shown in Table 4. The "Fabric Use" column shows the number of RPUs that allow the workload to achieve near-maximum speedup. We do not use the number for actual maximum speedup, because allocating more RPUs to an application yields diminishing returns (as shown in Figure 5), e.g. H.264 can use up to 19 RPUs, but the speedup when increasing allocation from 5 to 19 RPUs is only 1.19×. The number of RPUs to achieve actual maximum speedup would be very high, and would not show how many RPUs would be useful in a realistic scenario, where fabric space is expensive. The values from the table are not provided to COREFAB for guiding its operation (fabric allocation is described below), but they are presented here because they are useful for analyzing workload performance at different fabric sizes later on (Figure 7).
6.2 Comparison Partners

We compare COREFAB with the following techniques:

- **Reconf-Base** – a SoC with one reconfigurable core and three GPPs, without any possibility for sharing the fabric with the GPPs.

- **Spatial-Partitioning** – the fabric is split into equally-sized shares (one share per application in the workload) that can be used concurrently, a technique also used in ReMAP [5]. In a quad-core system, at most four applications can execute at the same time and thus the fabric is split into at most four shares. To provide true independent access to the scratchpad memory for each fabric share, each has two dedicated memory ports with a correspondingly reduced bandwidth. Thus for a workload that uses 4 cores, the fabric memory bandwidth is \(\frac{1}{4}\) compared to the other approaches, while for a workload that uses 2 cores, the bandwidth is \(\frac{1}{2}\).

- **Gap-Filling** – the fabric can be accessed by GPPs, but an SI may only execute if the fabric is not currently used by the reconfigurable core, i.e. fabric accesses are serialized, prioritizing accesses from the reconfigurable core. As this technique allows access by GPPs to the fabric, the FAM module is required, incurring its protocol overhead of 1 cycle per Remote-SI execution. The serialization of fabric accesses is similar to the fabric access technique used in [17]. While the Remote-SI latency overhead of 1 cycle is not present in [17], as their system is limited to sharing the fabric between only 2 cores. For COREFAB and Gap-Filling, the RPUs were statically assigned to the cores as follows: Using the application performance characteristics from Figure 5, a branch-and-bound solver chose the allocation that resulted in the best average performance. While the focus of this work was not on fabric partitioning in multi-cores, COREFAB can be combined with sophisticated online or offline techniques, such as those proposed in [17] or [7] to further improve performance.

6.3 Results

The goal of COREFAB is to improve performance of a reconfigurable multi-core system by increasing GPP performance (i.e. by extending the benefits provided by a reconfigurable core to the non-reconfigurable cores in a multi-core system). Figure 6 shows the normalized per-core runtimes relative to COREFAB for all approaches and workloads, and Table 5 shows the average runtime split by reconfigurable and GPP cores for each strategy.

Gap-Filling has the same performance as COREFAB on the reconfigurable core, as COREFAB’s SI merging only benefits applications running on GPPs. On the GPPs, Gap-Filling has 30% worse performance than COREFAB on average, as it does not provide true concurrent execution. Spatial-Partitioning has much worse performance on the reconfigurable core (more than 3× worse than COREFAB) as it partitions the memory bandwidth and the fabric among the cores (e.g. for Workload 2, each of the 3 fabric shares has \(\frac{1}{3}\) of the memory bandwidth and \(\frac{1}{3}\) of the RPUs) and thus generally the reconfigurable core has a lower fabric share compared to the other approaches. However, the GPPs obtain larger fabric shares and may use them exclusively with independent memory ports, which leads to better performance than COREFAB on the GPPs (2% on average). For workloads where the fabric shares of the GPPs are sufficiently large and the kernels are not memory-intensive (so that the reduced bandwidth has lower impact), Spatial-Partitioning achieves even better performance on the GPPs (16% better than COREFAB for Workload 1). However, that does not compensate for the lower performance of the reconfigurable core.

Reconf-Base has equal or better performance on the reconfigurable core than the other methods (12% better than...
COREFAB on average), as all RPUs are assigned to it, where for the other strategies a part of the fabric is assigned to the other cores. However, it has drastically worse performance on the GPPs (more than 4× worse than COREFAB), as they cannot offload their kernels onto the fabric, instead having to run them on an unaccelerated GPP.

The accumulated time that all cores in the system were busy processing the workload (i.e. total system activity) for each workload is shown in Figure 7, with COREFAB leading to the smallest accumulated runtime (on average 10% less than Gap-Filling, the closest competitor).

Performance of the evaluated techniques depends on the workload, e.g. examining only Workloads 1–3, Gap-Filling is 49% worse on the GPPs than COREFAB, while for Workloads 4–6 Gap-Filling is only 1% slower. The reason for this is that Workloads 4–6 consist of applications that have very large inter-SI gaps (see Table 2), such as SUSAN. Such gaps leave enough “idle time” on the fabric to allow applications on other cores to run their SIs on the fabric, with a very low chance of concurrent fabric access. However, COREFAB is designed to facilitate these types of concurrent fabric accesses, thus if the workload profile is similar to Workloads 4–6 COREFAB would provide little benefit (however, COREFAB also does not perform worse than Gap-Filling). Spatial-Partitioning benefits from workloads consisting of tasks that (i) do not have high memory bandwidth requirements or at least all applications of the workload having similar bandwidth requirements and (ii) achieve most of their speedup within their assigned fabric share or at least have similar performance characteristics (see Figure 5). These requirements are satisfied by Workload 6, where AES and SHA achieve most of their speedup with 1 or 2 RPUs, respectively.

COREFAB achieves the best results due to its ability of merging concurrent fabric accesses, while the other fabric sharing strategies (except for Spatial-Partitioning) need to serialize fabric accesses. However, sometimes concurrent fabric accesses result in resource conflicts, leading to stalling of the Remote-SI. The major reason for conflicts was a busy memory port (98% of all conflicts) with insufficient RPU link capacity responsible for the remaining conflicts (2%). No RPU conflicts occurred, as the minimal amount of RPUs required for running the application from the workload in hardware was statically assigned to each GPPs.

To measure fabric utilization, we examine the time-span when all applications of a workload are being executed (as in this time-span the potential to reduce underutilization

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory Port Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>COREFAB</td>
<td>50.1%</td>
</tr>
<tr>
<td>Gap-Filling</td>
<td>42.9%</td>
</tr>
<tr>
<td>Spatial-Partitioning</td>
<td>14.5%</td>
</tr>
<tr>
<td>Reconf-Base</td>
<td>30.4%</td>
</tr>
</tbody>
</table>

Table 6: Fabric utilization (Memory Ports) for Workload 3.

To show in detail how the fabric is accessed by two different cores, Figure 8 provides an excerpt from COREFAB executing Workload 1 on a fabric with 8 RPUs. The green bars in the lower part show the SI executions on the fabric of H.264 running on the reconfigurable core (core 0). This particular kernel (Motion Estimation) of H.264 uses SIs that take approx. 60 cycles on the fabric, with little time between SIs, i.e. the inter-SI gaps are small. A technique such as Gap-Filling would allow SI executions by other cores only in these small gaps. However, COREFAB allows core 1 (a GPP running SHA – orange bars), access to the fabric even while the SIs of core 0 are currently running (e.g. at 1 the fabric is accessed by both cores at the same time). The different length of the orange bars is due to unresolved conflicts during concurrent fabric accesses by both cores, and therefore the SIs of core 1 being stalled (longer orange bars). At 2,
the SHA SI of core 1 is being executed during an inter-SI gap of core 0, thus core 1 has exclusive and thus conflict-free access to the fabric, allowing the SI to be processed quicker than at 1, when both SIs access the fabric. The number of conflicts depends not only on the SIs, but also on how the SIs are “aligned”, thus the variation in the SHA SI latency.

We implemented the SI Merger and FAM modules (the two main components of COREFAB) as standalone RTL modules for area overhead analysis using Synopsys Simplify H-2013.03 for a Virtex-5 LX110T. The area overhead of the COREFAB extensions is 1231 LUTs (98 LUTs for FAM and 1133 LUTs for the SI merger, no BlockRAMs were needed for either).

7 Conclusion
In this work we present COREFAB, an approach for using a reconfigurable fabric in a multi-core system efficiently, thereby increasing overall system performance. We observe that typically, a kernel executing on the fabric does not utilize all of the fabric resources, therefore COREFAB enables simultaneous fabric usage by a non-reconfigurable core and the reconfigurable core by merging fabric accesses, when possible. The novel contributions of COREFAB are: a protocol for allowing access to the reconfigurable fabric in a multi-core system and the concept and hardware architecture to merge simultaneous fabric accesses on-the-fly, enabling for the first time concurrent use of the fabric by different cores in a flexible way. This allows offloading computationally intensive kernels from non-reconfigurable GPP cores to the fabric of a reconfigurable core in a multi-core SoC.

Compared to state-of-the-art techniques for multi-core systems, COREFAB achieves the best results in performance by improving runtime on non-reconfigurable cores by 1.3× on average, without slowing down the reconfigurable core. Additionally, the total system activity (accumulated runtime on all cores) for processing workloads is decreased by 10% on average.

8 Acknowledgments
This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).

References