U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Task-Level Pipelining in Configurable Multicore Architectures

Ali Azarian

Supervisor: João Manuel Paiva Cardoso

Doctoral Program in Informatics Engineering

January, 2016

Faculty of Engineering, University of Porto (FEUP)

Task-Level Pipelining in Configurable Multicore Architectures

Ali Azarian

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor of Philosophy (Ph.D.) in Informatics Engineering

President: Prof. Dr. Eugénio da Costa Oliveira, University of Porto
Referee: Dr. José Gabriel de Figueiredo Coutinho, Imperial College London
Referee: Prof. Dr. João Luís Ferreira Sobral, University of Minho
Referee: Prof. Dr. Jorge Manuel Gomes Barbosa, University of Porto

---

January, 2016

This thesis is dedicated to:
Nasrin, Amir hossein and my parents

Abstract

Techniques to speedup and accelerate the execution of sequential applications considering the multicore synergies provided by contemporary architectures, such as the ones possible to implement using Field-Programmable Gate Arrays (FPGAs), are increasingly important. One of the techniques is task-level pipelining, seen as a suitable technique for multicore based systems, especially when dealing with applications consisting of producer/consumer (P/C) tasks. In order to provide task-level pipelining, efficient data communication and synchronization schemes between producers and consumers are key. The traditional mechanisms to provide data communication and synchronization between P/C pairs, such as FIFO-channels and shared memory based empty/full flag schemes, may not be feasible and/or efficient for all types of applications.

This thesis proposes an approach for pipelining tasks able to deal with in-order and out-of-order communication patterns between P/C pairs. In order to provide efficient communication and synchronization between producer/consumer tasks, we propose fine- and coarse-grained data synchronization approaches to achieve pipelining execution in FPGA-based multicore architectures. Our approach is able to speedup the overall execution of successive, data-dependent tasks, by using multiple cores and specific customization features provided by FPGAs. An important component of our approach is the use of inter-stage buffer schemes to communicate data and to synchronize the cores associated with the producer/consumer tasks. Recognizing the importance to reduce the number of accesses to shared and/or external memories, we propose optimization techniques for our fine-grained data synchronization approach, specially addressing the reduction of memory accesses. We evaluate our approaches with a set of representative benchmarks using an FPGA board and measurements on real hardware. The experimental results show the feasibility of our approaches in both in-order and out-of-order producer/consumer tasks. Moreover, the results using our approach reveal noticeable performance improvements for a number of benchmarks over a single core implementation without using task-level pipelining.

Keywords: Multicore architectures. Task-level pipelining. FPGA. Producer/-Consumer data communication.

Resumo

Técnicas que permitam acelerar a execução de aplicações sequenciais têm assumido uma importância cada vez maior no contexto de arquiteturas multinúcleo, incluindo as implementadas utilizando *Field-Programmable Gate Arrays* (FPGAs) . Uma dessas técnicas é a execução em *pipelining* a nível de tarefas, a qual permite lidar com aplicações que contêm tarefas do tipo produtor/consumidor (P/C). Para utilização desse *pipelining*, é importante haver comunicação de dados e formas de sincronização entre produtores e consumidores eficientes. Os mecanismos tradicionais que possibilitam comunicação e sincronismo entre pares de P/C, tais como canais FIFO e bits de sinalização vazio/cheio baseados em memória partilhada podem não ser factíveis e/ou eficientes para todos os tipos de aplicações.

Esta tese propõe uma abordagem para alcançar o *pipelining* de tarefas, incluindo tarefas com padrões de comunicação em ordem ou fora de ordem. Reconhecendo a importância da comunicação e da sincronização entre pares P/C, propõe-se utilizar abordagens de sincronização com granulosidades fina e grossa, possibilitando a execução em pipeline de tarefas P/C em sistemas multinúcleo baseados em FPGA. As abordagens propostas permitem acelerar a execução global de tarefas sucessivas e com dependência de dados, por meio da utilização de múltiplos núcleos e de características específicas proporcionadas pelos FPGAs. Um importante componente da abordagem proposta é a utilização de um *buffer* para a comunicação de dados e para a sincronização dos núcleos associados às tarefas produtoras/consumidoras. Além disso, são propostas técnicas de otimização para essa abordagem de sincronização de dados de granularidade fina, a fim de reduzir o número de acessos a memórias partilhadas e/ou externas. As abordagens propostas foram avaliadas com a execução de um conjunto de benchmarks em uma placa de desenvolvimento para FPGAs e com medições reais. Os resultados das experiências mostram que essas abordagens são factíveis para tarefas do tipo P/C, tanto com execução em ordem quanto com execução fora de ordem. Adicionalmente, os resultados revelam melhorias de desempenho significativas para a execução de diversos benchmarks quando comparados com uma implementação com um núcleo e sem pipelining a nível de tarefas.

Keywords: Arquiteturas multinúcleo (multicore). *pipelining a nível de tarefas.* FPGA. comunicação de dados produtor/consumidor.

Acknowledgments

First and foremost, I would like to express my special appreciation and thanks to my adviser Professor **João Manuel Paiva Cardoso** for taking me as his student, for all the time he spent around my work and for all his support, patience, motivation and immense knowledge. I would like to thank him for encouraging my research and for allowing me to grow as a member of his research group. He gave me invaluable guidance and pivotal feedback throughout and continuously drove me to do better and better. I could not have imagined having a better adviser and mentor for my Ph.D. study.

I also wish to express my gratitude to my committee members for serving as my committee members even at hardship.

However, this work would not have been possible without the support of the many people I have met during the several years that took this Ph.D., and they also deserve a mention. I would like to convey my sincere thanks to Adriano Kaminski Sanches, which has been as a close friend for me during his stay in the lab; my colleagues from the SPeCS lab, André C. Santos, Ricardo Nobre, João Bispo, Tiago Carvalho, Pedro Pinto, Luís Reis and Maria Pedroto for all of their technical and non-technical support, their kindness and the great events we shared together.

Also, I would like to thank Professor Jürgen Becker and Professor Michael Hübner for the opportunity to work during 11 months at the Institute for Information Processing Technologies (ITIV) in the Karlsruhe Institute of Technology (KIT, Germany). I thank Stephan Werner for the fruitful discussions that we have had and his kindness, and to Pedro Santos for his friendship during this stay.

I would like to thank my co-authors Professor João Canas Ferreira for his help and support and also Professor José Carlos dos Santos Alves for his help to learn Xilinx EDK.

I wish to thank my wife, **Nasrin**, for all of her support, patience, love, kindness, and self-sacrifice. This work could not be possible without her support and self-sacrifice. Also, I would like to thank my parents (Hassan Azarian and Tahereh Masjedi) for their kindness, love and support spiritually throughout writing this thesis and my life in general. In addition, I thank my parents-in-law for their support and kindness and also my sister-in-law, *Neda* who has always supported me until the last day of her life. God rest her soul.

I wish to thank the staff from the department of informatics engineering (DEI), the academic service and the international office of the faculty of engineering (FEUP) for their help and support.

This work was partially funded by the Ph.D. grant SFRH/BD/80481/2011 awarded by the Portuguese Foundation for Science and Technology (FCT - *Fundação para a*

Ciência e a Tecnologia), to whom I would like to thank for the support.

Last but not the least, I wish to thank Professor Mahmood Ahmadi from Razi University (Kermanshah, Iran), for his support, encouragement and his great advises before and during my Ph.D. period.

Ali Azarian

"Learn from yesterday, live for today, hope for tomorrow.
The important thing is not to stop questioning."

Albert Einstein

Contents

List of Figures

List of Tables

List of Abbreviations

CAKE	Computer Architecture for a Killer Experience
CLF	Concurrent lock-free
CMP	Chip Multiprocessors
CPT	Cycles Per token Transfer
DCT	Discrete cosine transform
DiscoPoP	Discovery of Potential Parallelism
DSL	Domain-Specific Language
FDCT	Fast Discrete Cosine Transform
FDTD	Finite-Deference Time-Domain
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
HLS	High-level Synthesis
IP	Intellectual Property
IPD	Input Port Domains
ISB	Inter-Stage Buffer
KPN	Kahn Process Network
MB	MicroBlaze
MoC	Models of Computation
NPB	NAS Parallel Benchmark
OPD	Output Port Domains
P/C	Producer-Consumer
PAM	Pipeline Application Modeling
PEACH2	High-Performance Communication Channel
RTL	Register-Transfer Level
SDF	Synchronous Data Flow
SMT	Simultaneous Multithreading
SPPM	Synchronized Pipelined Parallelism Model
TAB	Tag-less Access Buffer
TaLP	Task-level Pipelining

TTL Task Transaction Level

CHAPTER 1

Introduction

Chapter Outline

COMPUTER performance has been driven largely by decreasing the size of chips while increasing the number of transistors they contain. Based on Moore's law [Moo65], the number of transistors on integrated circuits doubles about every two years or even faster (18 months). This ongoing trend led the computer companies industry for many years. However, based on physical limitations, the transistors cannot shrink forever. Therefore, the computer companies and manufactures have struggled to power dissipation and heat generation. Even performance-enhancing approaches like running multiple instructions per thread have bottomed out. For these reasons, the chip performance increase has begun slowing.

In recent years, multicore processors and parallel platforms are providing the opportunity to increase the performance of applications by using multicores in a single processor. Multicore architectures have spread to all computation domains from embedded systems to personal computers to high-performance supercomputers. In addition, reconfigurable computing devices such as Field Programmable Gate Arrays (FPGAs) [BR96] are also provided the opportunity for computing and storage resources to the specific needs of an application. FPGAs provide several sufficient

capabilities (e.g., hardware customization) for implementing or prototyping multicore systems. FPGAs also have the advantage of being able to be reprogrammed in the field to add new unforeseen features or corrections (e.g., a new bitstream can be uploaded remotely, instantly).

Nowadays, other accelerators such as GPUs (Graphic Processing Units) have also a potential for high performance for many applications using a large number of cores which run in parallel to accelerate the execution of applications. However, GPUs requires the use of specific programming tools and techniques to achieve high performance. Also, the advantages of using GPUs are depend on the parallelism potential of the applications. Thus, GPUs are not suitable as accelerators for some applications.

To efficiently exploit the advantages of multicore architectures, parallel programming and parallelization techniques to speedup the processing of an application are becoming more and more important. The computations performed by a given program provide opportunities for parallel execution at different levels of parallelism such as data-level, task-level, and pipeline parallelism [HP11]. A variety of applications in the domain of image, video and signal processing (see, e.g., the PARSEC benchmark suite [BKSL08]) consists of linear sequential stages which can be dependent or independent with each other. In most of these sequential programs, the output of a stage is the input of the next stage. One possibility to improve performance is to provide pipelining schemes to allow the processing of the next input before the subsequent stages have completed their process. In pipeline parallelism [MSM04], the stages can operate simultaneously and process different data. A parallel execution in pipeline parallelism is obtained by partitioning the data into a stream of data elements that flow through the pipeline stages one after another.

In the domain of pipeline parallelism, task-level pipelining is also an important technique to speedup processing of an application, especially when dealing with applications consisting of producer/consumer *(P/C)* tasks (see, e.g., [KKK+09]) in multicore based systems. In these applications, producer tasks output data to be processed by the consumer tasks. Using task-level pipelining, a consumer computing stage (e.g., consisting of a loop or a set of nested loops and also identified herein as

task) may start execution, before the end of the producer computing stage, based on data availability. Performance gains can be achieved as the consumer can process data as soon as it becomes available. Task-level pipelining may provide additional speedups over the ones achieved when exploring other forms of parallelism. In the presence of multicore-based systems, task-level pipelining can be achieved by mapping each task to a distinct core and by synchronizing their execution according to data availability. It can accelerate the overall execution of applications by partially overlapping the execution of data-dependent tasks (herein: Computing Stages).

We can distinguish two types of data synchronization granularity between producers and consumers: *Fine-grained* and *Coarse-grained data synchronization*. In fine-grained schemes, each data element is used to synchronize computing stages. In coarse-grained data synchronization schemes, instead of each data element, chunks of elements or an entire array of elements (e.g., an image) is considered to synchronize computing stages.

For transferring data from producer stages to consumer stages, a common data structure is required which can be seen as a data buffer that can be accessed by both producer and consumer tasks. The producer stores the data elements into the buffer and the consumer loads data elements from the buffer for further processing. The size, implementation and the synchronization mechanisms of the buffer between the producer and the consumer can be a challenge. The synchronization mechanism between the producer and the consumer needs to ensure the correctness and efficiency of data communications.

This dissertation focuses on the communication and synchronization mechanisms between P/C stages of pipelining for both in-order and out-of-order communication pattern and also providing task-level pipelining in the context of FPGA implementation of multicore architectures. In the next section, we address the motivation, problem statement and the challenges related to the topic.

1.1 Motivation and Problem Overview

FPGAs [BR96] allow hardware customization capabilities. The hardware resources provided by FPGAs allow the implementation of multicore architectures (e.g., using softcore processors, custom hardware components), specific memory architectures, and specific interconnections between the components of the system (including interconnections between cores). These allow the implementation of complex System-On-a-Chip (SoC) solutions using an FPGA device [BR96] and implying acceptable Non-Recurring Engineering (NRE) costs. The hardware customization provided adds a design dimension to multicore/multi-CPU architectures. This design dimension can be explored in order to make more efficient the pipelining of producer/-consumer tasks. For example, specific communication channels and buffers can be implemented in order to communicate data between producers and consumers. The use of custom on-chip channels may promote data communications and synchronization between producers and consumers to on-chip and thus without needing to access external memories (i.e., memories outside the FPGA device). Possibly, not all data can be communicated on-chip due to the high memory requirements this might imply, but an efficient data communication scheme between producers and consumers may promote to on-chip as much as possible those communications.

In order to provide efficient data communication and synchronization mechanisms in multicore architectures, using a suitable communication structure between the producer and the consumer is essential. The simplest implementation of task-level pipelining uses a FIFO (first-in, first-out) channel between cores implementing P/C pairs. The FIFO can store data elements in the order they are produced to establish data synchronization between the producer and consumer. The FIFO is a suitable data communication scheme when the sequence of producing data is the same as the sequence of consuming data (referred herein as in-order data communication pattern or simply in-order). In this case, the data communication between the producer and consumer can use a FIFO storing one data element in each stage. Although using FIFO channels between producers and consumers is an efficient solution for in-order P/C pairs, it may not be efficient or feasible for out-of-order

P/C pairs and it might be necessary to use other data communication mechanisms [TKD05]. In out-of-order P/C pairs, the sequence of producing data is different from the sequence of consuming data.

In the presence of out-of-order P/C pairs, recent approaches address compiler-based task-level pipelining (see e.g., [TKD03b, TKD02], and [TKD05]). In those approaches, an extra storage and buffer memory are introduced, based on the order of the communication pattern between the producer and consumer, determined at compile time [TKD05]. The data communication in these approaches considers unbounded FIFOs that may prevent its use in a number of implementations. Also, they use memory access reordering techniques. Another recent approach, [ZHX+15] also use block FIFOs which is block-based data streaming technique to provide out-of-order data communication between P/C pairs. The problem with this approach is that the data communication between each block FIFOs is limited to the in-order data communication between P/C pairs.

This dissertation deals with a number of research questions regarding the task-level pipelining for in-order and out-of-order applications which can be posed as follows:

- **Question 1.** How can we provide task-level pipelining in the context of FPGA implementations using multiple microprocessors?
- **Question 2.** How can we synchronize the communications between P/C stages of the pipeline for both different in-order and out-of-order communication patterns at run-time?
- **Question 3.** How can we synchronize the communications between P/C stages of the pipeline with different ratios of production or consumption?
- **Question 4.** What is the suitable scheme for inter-stage communication between cores according to application demands?
- **Question 5.** What is the impact on the performance for task-level pipelining when using different inter-stage communication schemes?
- **Question 6.** How to reduce the number of accesses to the main memory when communicating data between P/C pairs?

1.2 Contributions

Although approaches to pipeline sequences of data-dependent loops have been addressed previously (e.g., [ZHD03, Car05, Sni02], and [WL06]), the work presented here is novel in three main aspects. First, we use fine-grained and coarse-grained, data-driven synchronization schemes between P/C stages of the pipeline. The implementation uses a hash-based scheme to limit local buffer size. Other techniques have focused on finding appropriate sized synchronization buffers [ZSHD02] to enforce the same P/C order thus sacrificing concurrency. Our fine- and coarse-grained synchronization schemes are similar in spirit to the empty/full tagged memory scheme used in the context of shared memory multiprocessor architectures (see, [Smi82, Smi86]). Second, the control scheme decouples the control units of each stage and uses inter-stage buffers (ISBs) to signal the availability of data elements to the subsequent stage. This approach allows out-of-order execution of loop iterations between tasks constrained by data dependencies. The overall benefit of these features is that we are able to achieve almost the theoretical speedup and to reduce the size of the buffers to communicate data between computing stages. Lastly, we describe the application of this technique in the context of configurable multicore architectures implemented using FPGA devices showing how task-level pipelining can be applied to multicore architectures and the impact on the results of different customized inter-stage buffers.

This dissertation makes the following specific contributions:

- **Contribution 1.** It presents a technique for pipelining the execution of sequences of data-dependent loops using fine-/coarse-grained synchronization.
- **Contribution 2.** It implements customized multicore architectures for the inter-stage communication to achieve pipelining execution of P/C pairs.
- **Contribution 3.** It presents techniques to improve out-of-order P/C pairs when a consumer uses more than once a data element output by a producer.
- **Contribution 4.** It presents a technique to improve the inter-stage buffer communication scheme.

- **Contribution 5.** It presents an evaluation of the proposed approach using measurement on real hardware.

1.3 Organization

This dissertation is composed of seven chapters and one appendix. Bibliographic references and information about the author are also provided. The remainder of this dissertation is organized as follows:

- **Chapter 2 –Related Work**. This chapter discusses and includes an overview over related work to our research, namely on producer/consumer synchronization and communication models; loop pipelining approaches to pipeline tasks; and code transformation and profiling tools to support task-level pipelining.

- **Chapter 3 –Task-Level Pipelining.** This chapter describes the main concepts of task-level pipelining and the traditional schemes to implement tasks-level pipelining in multicore architectures. This chapter also describes the partitioning of programs for multicore architectures using producer-consumer communication model. In addition, we present different types of communication patterns and ratios and dependencies between producer/consumer pairs.

- **Chapter 4 –TaLP Approach.** This chapter presents our approach to task-level pipelining and describes our fine-grained and coarse-grained data synchronization approaches to provide task-level pipelining. In addition, we introduce and discuss a possible design-flow for task-level pipelining in multicore architectures implemented in FPGAs.

- **Chapter 5 –Optimization Techniques.** This chapter presents optimization techniques for traditional FIFO-based schemes and also for our fine-grained inter-stage buffer scheme. The techniques on reducing the number of accesses to the main memory.

- **Chapter 6 –Experimental Results.** This chapter presents the experimental evaluation of our approach, the results achieved with optimization schemes,

and implementation details of our multicore architectures. The evaluation considers the measurements using FPGA development board and the selected benchmarks include some typical kernels of embedded applications, such as signal and image processing.

- **Chapter 7 –Conclusions and Future Work.** This chapter summarizes and discusses the most important aspects and contributions of this dissertation, the overall relevance of the subjects approached, and also future research activities.

CHAPTER 2

Related Work

Chapter Outline

THIS chapter presents a review of the related work around the concepts, approaches, implementations and tools to provide task-level pipelining (TaLP). In this chapter, first, we analyze the studies related to the synchronization and communication models in multicore architectures using the producer-consumer model for the communications and the synchronizations between tasks. Second, we provide a summary of recent approaches related to software pipelining to accelerate the execution of tasks. Those approaches include loop pipelining and coarse-grained pipeline parallelism mechanisms and code transformation techniques to support TaLP. Additionally, we describe high-level synthesis (HLS) and profiling tools to support the implementation and the evaluation of the parallelization potential of the applications in the context of TaLP. These tools can determine the data dependencies

between the tasks and also estimate the speedup of a program using different forms of parallelism. Finally, we review other parallel models of computation (MoC) which support pipeline parallelism and TaLP.

2.1 Producer-Consumer Communications

The producer-consumer communication and synchronization model [BA82] has been used in many studies as a model for inter-process communication in multiprogramming systems [Jef93] and also for fine-grained non-strict structured data pipelining [IKA99]. Several recent studies investigate synchronization and communication models for the communications between parallel processors in multicore architectures to increase the performance at the task level. By considering the variety of approaches and implementations used, we can categorize these studies into two main groups: the studies using flag-based data structures schemes (e.g., empty/full bit) for the synchronizations, and the studies dealing with data streaming over channels (e.g., FIFO channels) to communicate and to synchronize the execution of tasks in multicore architectures. In the following sections, we describe these studies and how their approaches can support TaLP.

2.1.1 Flag-based Data Communication

Flag-based synchronization models are usually based on the synchronization bit with two status (*empty* and *full*). The empty/full bit scheme has been introduced by Smith et al. [Smi82, Smi86]. They used an empty/full tagged memory approach for the synchronization between processors in the HEP multiprocessor computer system. The HEP computer system which is a large scale parallel computer uses MIMD architecture and a shared memory to provide concurrent processing. In this approach, to provide data sharing between the processors, they used shared memory locations. In these memory locations, the access state has two status (*empty* or *full*). The data can be loaded from the memory if the access state of the memory location is set to *full*. In a similar way, the data can be stored into the memory if the access state of the memory location is set to *empty*. The empty/full tagged

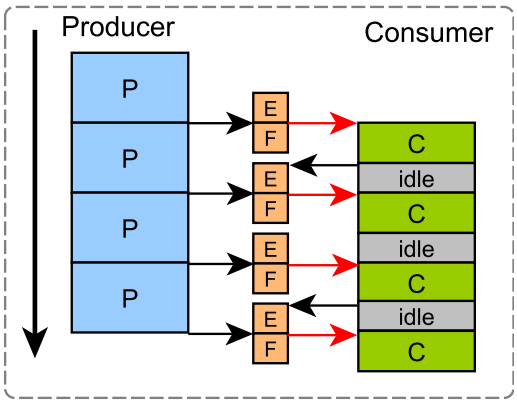


Figure 2.1: Register-Based Synchronization approach for SPPM Applications (source: [FJ08]).

memory approach has been used to read an empty location of the memory on the HEP-1 busy-waited rather than trapping. This scheme has been used in many other approaches and implementations.

For instance, Fide et al. [FJ08] have used the Register-based Synchronization (RBS) approach which is one of the flag-based synchronization schemes between cores. The RBS approach uses a hardware shared register to provide the synchronization support between cores (e.g., the producer and the consumer) and also the Synchronized Pipelined Parallelism Model (SPPM) [VJ04]. The SPPM model is a parallel execution approach which enforces blocking synchronization between the producer and the consumer. In the RBS scheme, the shared register has an empty/-full status bit, which is similar to the empty/full tagged memory scheme. Figure 2.1 shows the RBS approach for SPPM applications. When the producer reaches the synchronization point, it sets the register to the full status, which wakes the consumer. When the consumer is done, instead of spin waiting and consuming system resources, it goes into the idle mode to save power and energy and also changes the flag register to the empty status. When the register status changes, the consumer wakes up and the process repeats. Therefore, the producer can continue operating while the consumer is still consuming the previous produced data or when the consumer is in the idle mode.

In this study, the authors used three benchmarks to evaluate their pipelining approach: Red-Black Solver, Finite-Deference Time-Domain (FDTD) and ARC4

Stream cipher. The results show performance improvements of $2-5\%$ per iteration for Red-Block solver; $6-11\%$ for FDTD, and a negligible improvement for ARC4 when using the RBS synchronization technique. One of the bottlenecks of RBS scheme is the limitation of the number of shared registers for the communication transfers between producer and consumer pairs. However, this technique can improve the synchronization and communication support for multi-threaded applications.

The flag-based data structures can be implemented in different ways such as tables to implement the synchronization and communication between tasks. For instance, Bardizbanyan et al. [BGW+13] presents an efficient data accessing approach using a tag-less access buffer (TAB) to improve data access energy and performance. The TAB is located between the register file and the L1 data cache in the memory hierarchy. In this approach, the compiler detects the memory accesses and replace them with a TAB access. The compiler allocates a TAB entry before memory accesses by inserting TAB instructions. These instructions prefetch the L1 data cache line into the TAB. Although this approach provides a technique to access data references in a more energy-efficient manner, it provides small performance improvements. The results show 34.7% reduction of data-access energy with four TAB entries. The four-entry TAB configuration reduces energy in the L1 data cache and data translation lookaside buffer (DTLB) by 35.4% and 41.9%.

In the next section, we describe other studies dealing with data streaming over channels such as FIFO channels to communicate and to synchronize the execution of tasks in multicore architectures. In addition, we describe studies using shared or distributed memories for the synchronization and communication between tasks.

2.1.2 Streaming Data Over Channels

Many studies deal with data streaming between cores over channels to communicate and to synchronize the execution of tasks in multicore architectures. In these studies, they use standard FIFOs as data communication channels between cores when the sequence of producing data is the same than the sequence of consuming data (e.g., streaming applications where a set of computations is applied in sequence

on an input data stream). However, when the sequence of producing data is different from the sequence of consuming data, a simple FIFO might not be sufficient to implement the communication between producer and consumer pairs and other data structures implemented as shared or distributed memory can be used as data communication and synchronization between cores. In the following section, we describe the studies which proposed data communication and synchronization models to overlap the execution of tasks using FIFOs or shared memory as a communication and synchronization between P/C tasks.

FIFO Channels

A relevant FIFO-based approach is the one proposed by Ziegler et al. [ZSHD02, ZHD03]. It uses a coarse-grained synchronization scheme to overlap some execution steps of sequences of loops or functions. Their approach communicates data to subsequent stages using a FIFO mechanism. Each FIFO stage stores an array element or a set of array elements (e.g., a row of pixels in an image). Array elements in each FIFO stage can be consumed by a different order than the one they have been produced. Examples with the same order of producer-consumer only need FIFO stages with one array element. In the other case, each stage must store a sufficient number of array elements in order that all of them are consumed (by any order) before the next FIFO stage is considered. This is the major bottleneck of their technique since a FIFO stage may need to store many values as a consequence of using coarse-grained (the grain is related to the size of the FIFO stages) instead of fine-grained synchronization, and the number of data elements stored in each FIFO stage must be known at compile time.

Smith [Smi86] described an analysis to determine the communication requirements by this pipelining technique. The work in [Smi86] has been in the context of design space exploration and concerning speedups only one example is referred (with a speedup of 1.76×). Regarding their approach, our can be thought as a more generic approach, and also eliminating some previous constraints.

In another major studies, Turjan et al. (see, e.g., [TKD03b, TKD02, TKD05]) proposed a compiler-based approach considering FIFO buffers between tasks to solve

the data communication problem for out-of-order tasks. The focus of their approach is based on reorderings mechanism which can prevent the problem of out-of-order P/C pairs when using a FIFO channel. Based on the order of each P/C pair, a controller inside of the consumer core checks whether a FIFO channel is sufficient for every P/C pair or an additional memory is required [TKD03a]. This software approach is based on the Ehrhart theory [JB07] and assumes unbounded FIFO channels to synchronize the communication between the producer and the consumer.

Figure 2.2 presents an example of a producer and consumer process in their approach. In this example, the producer and the consumer communicate together point to point over unbounded FIFO channels using a blocking-read synchronization. As shown, in each iteration of the P/C pairs, the producer writes data into different FIFOs and also the consumer reads data from different FIFOs. An input port domain (*IPDs*) of a consumer stage is the union of the iterations at which the process's function stage reads data from the same FIFO. In a similar way, an output port domain (*OPDs*) of a producer stage is the union of the iterations at which the process's function writes data to the same FIFO. Each OPD is uniquely connected to another IPD via a FIFO. Over this FIFO, data are communicated to the mapping given by the mapping matrix M. By considering the mapping matrix, the data can be re-ordered in the order that consumer expects the data.

Recently, Zhang et al. [ZHX+15] presented an open-source system-level FPGA compilation framework called CMOST. In this study, they used a block-based data streaming technique to provide data communication between P/C pairs. In this approach, they proposed an extension of the traditional streaming framework (communication via FIFOs between the producer and the consumer) by introducing block FIFOs. Figure 2.3 presents an example of data communication between P/C pairs when using block FIFOs. As shown, the traditional FIFO is suitable only when the data communicated between the streaming stages are in the same order at the producer and consumer sides.

In this approach, the access pattern of the producer tasks and the consumer tasks are automatically reordered to create streaming buffers, and the corresponding address mapping is also performed to use the switching buffer allocated. Using

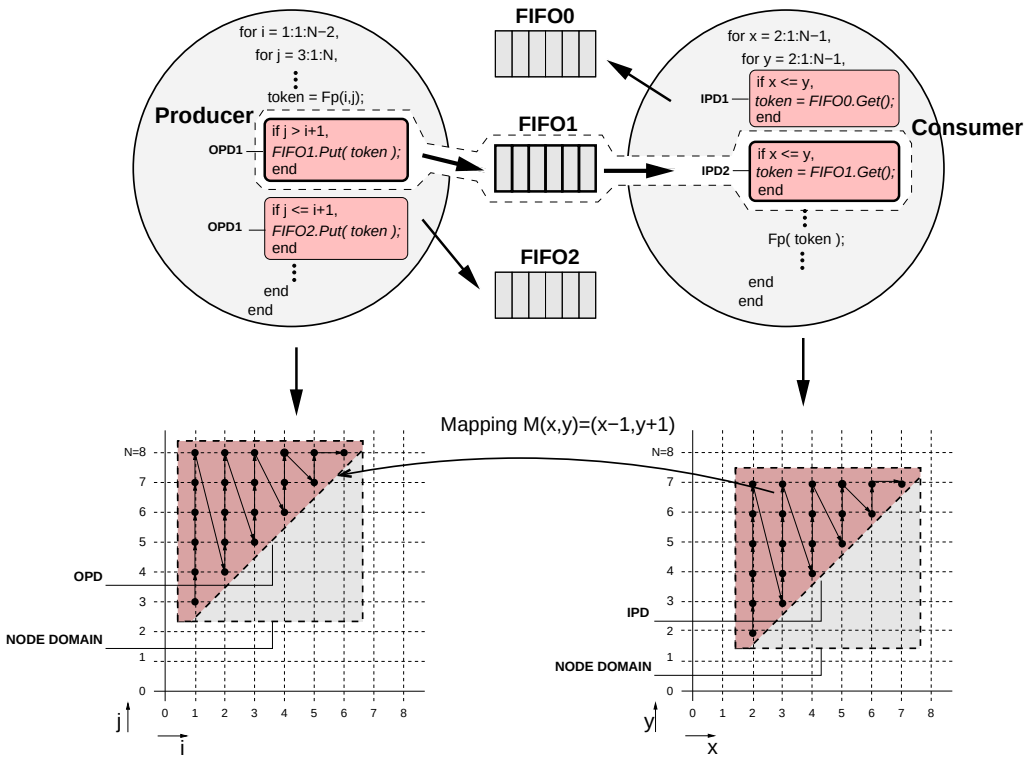


Figure 2.2: An example of a producer/consumer pair using multiple FIFOs between tasks and reordering the consumer (source: [TKD03a]).

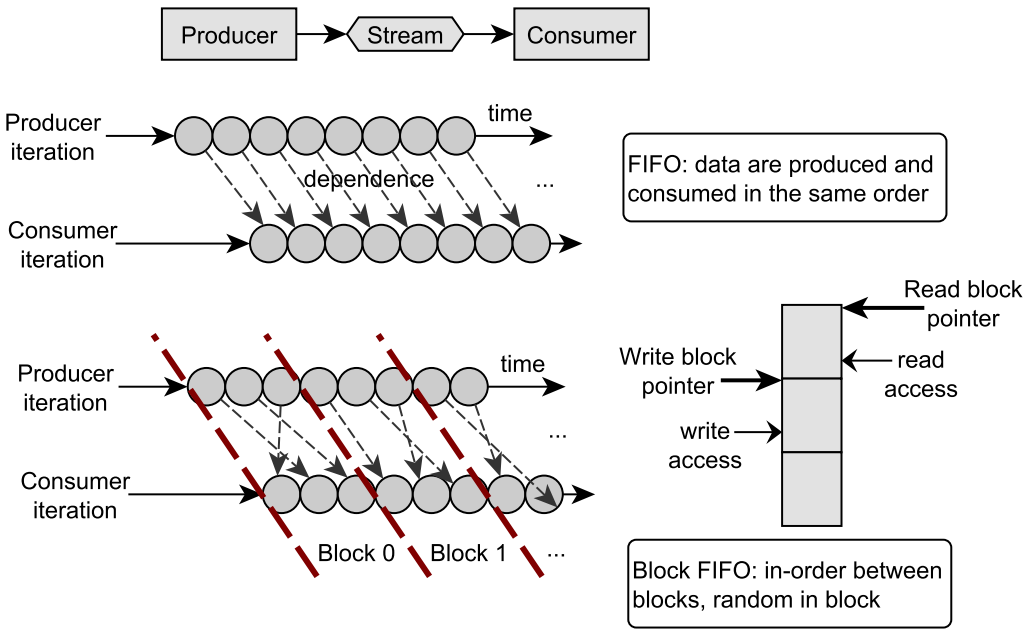


Figure 2.3: An example of P/C pair data communication using a traditional FIFO channels and using block FIFOs (source: [ZHX+15]).

block FIFOs between P/C pairs provides the out-of-order data communications in each block FIFOs. The authors used 5 real applications (MPEG, NAMD, Smith

Waterman, Black Scholes and Medical Imaging) to evaluate their approach. They measured the speedup and energy using a Xilinx Virtex-7 (VC707) FPGA-based board [Xil15a] and also compared the results with the 6-core CPU Intel Xeon E5-2640 [Int11]. The results show that CMOST can achieve speedups over 8× for MPEG, NAND and Smith Waterman benchmarks, and over 1.1× for Black Scholes and Medical Imaging benchmarks. Although the results for some benchmarks are considerable, the bottleneck is that in many applications, the consumer may request the data from the previous produced blocks or request data from the blocks which still are not available by the producer. Also, the size of the block FIFO can be also a bottleneck by considering different architectures. However, the authors mentioned that for these such cases, they need to add extra memory and reordering the pattern of producing and consuming data.

Shared Memory

When using shared memory synchronization, the tasks can read/write data in a global shared region. Therefore, shared memory can be used as a data synchronization and communication channel between cores. Also, flag-based synchronization models can be implemented using shared memory. In the context of the implementation of data structures using the empty/full scheme, the memory structure of multicore architectures can be classified into two categories: shared memory and distributed memory. Although many multicore systems are using both shared and distributed memory, we focus on studies which are using shared memory as a data synchronization and communication channel between cores to accelerate the execution of tasks in multicore architectures. Many studies have been focusing on performance improvements of the applications when using a shared memory synchronization model. These studies are mostly reducing energy and the latency of accessing shared memory. Here, we classify the studies which have used shared memory in multicore architectures into two main groups: performance improvement of applications when using shared memory as a data communication and synchronization channel between cores; inter-task data communications using producer-consumer model and shared memory.

First studies have been conducted by using two cores and a shared memory as synchronization and communication model. For instance, Byrd et al. [BF99] used a range of producer- and consumer initiated mechanism and their performance on a set of benchmarks in distributed shared memory multiprocessors. More recently, Miyanjima et al. [MKH+13, MKH+14] proposed an approach to achieve TaLP on multiple accelerators using a high-performance communication channel (PEACH2) which communicates data using shared memory between accelerators. In their approach, tasks are assigned to the accelerators and data can be computed using a sequence of GPUs in a pipeline manner. In [MKH+14], the authors achieved 52% of performance improvements compared to a single GPU by implementing TaLP using a shared memory synchronization mechanism. In this study, 100 images (1280×720 pixel of resolution) where processed using a sobel image filter consisting of three tasks.

The second group of studies has considered the inter-task communication in the producer-consumer model to improve the performance of applications by using a shared memory synchronization mechanism. For instance, Bei Li et al. [LW05] presents an implementation of Task Transaction Level (TTL) on the shared memory CAKE (Computer Architecture for a Killer Experience) architecture [EHM+05] which is a multiprocessor platform. In this study, they provide an efficient implementation of TTL inter-task communication on the CAKE tile architecture to improve the performance of streaming applications. The CAKE tile architecture consists of multiple general purpose CPUs (MIPS or TriMedia), various IP blocks, shared L2 caches and an interconnect network. The communication channels are implemented in the shared memory which consists of a channel buffer and channel administration's values. The channel buffer is a part of memory where the transferred data are stored (an ordered FIFO) and the channel administrations is a part of the memory where the status of the channel buffer such as buffer size, base addresses, and the synchronization constructs are stored. Bei Li et al. [LW05] also used shared memory for the synchronization between producer-consumer tasks implementing with ordered FIFOs. In addition, they use the CakeSim simulation framework [SH01] based on the TSS (Tool for System Simulation) model, a cycle-accurate C language

to evaluate the performance when using the CAKE tile architecture. The results show 80% improvements on the number of clock cycles per token transfer (CPT) and 23% reduction of the total clock cycles for running a JPEG decoder application.

In terms of inter-core communications on multicore architectures, many studies have been focusing on approaches which provide higher performance and power efficiencies when using shared memory as a communication and synchronization channel between cores. For instance, Zhiyi et al. [YXY+14] present a 16-core processor with shared memory and message passing inter-core communications to achieve higher performance and power efficiency for signal processing applications using both inter-core mechanisms. This 16-core processor consists of processor cores (PCore) and memory cores (MCore). Each PCore can be a source or destination processor core. The authors use a cluster-based memory hierarchy including the shared memory for embedded applications such as low-density parity-check (LDPC) decoder, a 3780-point fast fourier transform (FFT) module, a H.264 decoder and a long term evolution (LTE) channel estimator. The shared memory communication in this study can be summarized in three steps: source PCore stores data to shared memory in MCore; source PCore sends the synchronization signal to the destination PCore and destination PCore loads data from shared memory when synchronization signal is received. The results show that this FPGA-based approach has a considerable energy efficiency and performance compared to RAW [WTS+97] and CELL architectures [KDH+05].

Giacomoni et al. [GMV08] present a high-rate core-to-core software buffering communication mechanism called "FastForward" for multi-threaded pipeline parallel applications, such as network frame processing applications, which is implemented on multicore architectures. The FastForward technique is an optimization of single-producer/single-consumer concurrent lock-free (CLF) queues [Lam77], which provides very low latency and low communication overhead (36-40 ns per get or put operation) between processors and also provides higher (up to 4×) speedups for pipelining fine-grained stages compare to the other solutions such as Lamport's CLF queues [Lam77] (200 ns per operation). As the queues in the FastForward technique are single-producer (the program thread) and single consumer (a delegate thread),

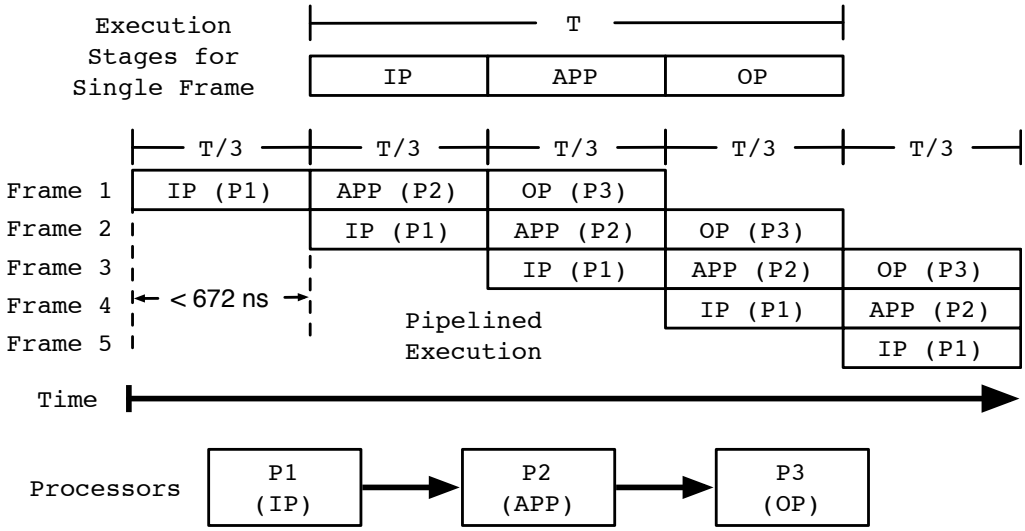


Figure 2.4: An example of a P/C pair network frame processing application using shared memory pipeline (source: [GMV08]).

for the synchronization, the *full* condition on the producer side and the *empty* condition on the consumer side needs to be checked frequently in a spin loop which might be critical for performance. In this study, they used a 2 GHz dual-processor dual-core AMD Opteron 270 for their evaluations.

Figure 2.4 shows an example of the network frame processing application. In this example, the execution stage for a single frame can be decomposed into three stages: input (IP) and output stage (OP) which transfer each frame to/from the network interface; the application (APP) stage which performs the computations of the frames. When considering the maximum frame rate in Gb Ethernet (1,488,095 frames per second), a new frame can arrive every 672 ns. As shown, as soon as each frame is available by the input stage, the application stage can process the input frame and then store the output frame into the shared memory.

By executing the input, the application and the output stages in parallel, the execution time of the stages can fully overlap in every time step. In order to reduce the communication overhead factor, the FastForward approach was developed to provide a fast communication primitive. This primitive only needs 36-40 ns per operation, a substantial gain when compared with lock-based queues, which require 200 ns per operation.

In summary, although the use of shared memory communication and synchro-

nization might be easy for programmers, it can face several challenges limiting in future many-core processors due to low scalability, high overhead and power consumption [YXY+14]. For instance, in a shared memory system, the traffic of communications when accessing the common memory is constrained by the shared bus. In addition, the synchronization primitives to provide mutual exclusion are complex and error-prone. Therefore, using shared memory can increase the potential for race conditions and deadlocks [SRI14]. Although using non-blocking synchronization, proposed first by Herlihy [Her91], can guarantee the correctness of the memory accesses, implementing a non-blocking communication scheme is typically complex for the programmer [Sut08] and might not provide higher performance for communication and synchronization between cores.

2.2 Loop Pipelining

Loop pipelining is one of the pipeline parallelism techniques to accelerate the execution of applications. Many studies described and analyzed the performance improvement of the applications by pipelining loops (see, e.g., [AJLA95]). Numerous studies have attempted to describe and analyze the performance of pipelining software loops onto reconfigurable architectures (see, e.g., [RCD07, Car05, Sni02, MKH+13, CW00]), especially when mapping innermost loops to FPGAs. Also, some of these studies used the producer-consumer pipelining model. In general, we can classify the studies related to loop pipelining into two main traditional groups:

- pipelining nested loops
- pipelining sequences of loops

In this section, we focus on approaches to pipeline sequences of loops.

In the context of loop pipelining of nested inner loops, Callahan et al. [CHW00] present an approach to accelerate the execution of loops using the Garp-C compiler and the Grap architecture. The approach uses a reconfigurable co-processor to reduce the overall execution of different types the loops (e.g., multiple control paths, multiple exits). In the approach used by Callahan et al. [CW00], only the loops of the application can be pipelined by the reconfigurable array. However, it depends

on the scheduling by the Garp-C compiler and also pipelining two data-dependent loops is not addressed. The results show 87% performance improvements compared to the original sequential execution time without using loop pipelining.

Ziegler et al. [ZSHD02] present a pipelining scheme at coarse-grained levels for accelerating loops by overlapping the execution of sequences of loops or functions. The idea is to allow sequences of loops or functions to start computing as soon as the required data items are produced in a previous function or by a specific iteration of a previous loop. This overlapping can reduce the overall execution of loops or functions. The compiler can also identify the computing stages and the dependencies between the iterations of loops, which can help to identify the potential for this coarse-grained pipelining, by analyzing the source code of the program.

Software pipelining has been focused on intense research efforts (see, e.g., [KKK+09, DG07, JA90, IKA99]) in order to generate a multi-threaded software-pipelined schedule for multicore architectures and to increase the execution rate of loops. For instance, Douillet et al. [DG07] present a solution to generate and extract threads, to schedule instructions and assign the threads to each core automatically in the context of multicore architectures. In order to provide the synchronization between cores, they use Lamport's clock on each thread unit. This approach has been implemented in the Open64 compiler [OPE05] re-targeted for the IBM Cyclops64 architecture, a dedicated petaflop platform for running high performance applications. The experimental results showed this approach can scale up well when the number of thread units increases across all the benchmarks tested, ranging from 57.5 to 81 relative speedup for 99 thread units.

By considering the importance of data communication between the sequences of loops, many studies have investigated data communication schemes between producers and consumers. In the context of producer-consumer synchronization schemes for task-level pipelining, several studies such as [RCD07, TKD05, BF99, Car05, ECR+10, VJ07] have been made. Additionally, [TKD05, TKD03b, TKD02] the producer-consumer synchronization is the main synchronization scheme for pipelining tasks. FIFO channels have been used for data communication between producers and consumers. In [VJ07], Vadlamani et al. proposed the Synchronized Pipelined

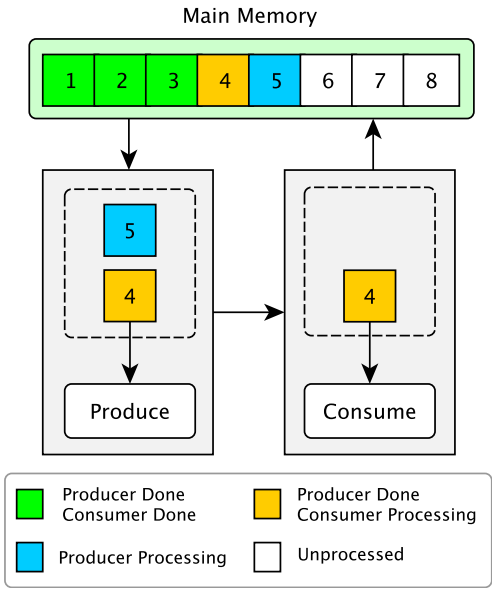


Figure 2.5: SPPM execution (source: [VJ07]).

Parallelism Model (SPPM) for parallelizing applications on Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMP). Figure 2.5 shows the logical representation of the SPPM model. As shown, the main memory holds the input and output data blocks. The producer loads the input data from the main memory and then send the produced results item to the consumer for further processing. The consumer can start processing data items as soon as data are available by the producer and then stores the final results into the main memory. In this approach, the producer and the consumer communicate through the shared memory.

Vadlamani et al. [VJ07] recognized that this model is not sufficient on CMPs when using private caches. In order to solve this problem, they developed the *Polymorphic Threads (PolyThreads)* model. In this model, the communication between cores is different from the SPPM model. In *PolyThreads* model, each core can have both the producer and the consumer code. As shown in Figure 2.6, when a thread's producer code is finished with a block, it sends a signal to other thread's producer to start and load the next input block from the main memory and also transforms itself into a consumer for the data just produced by itself. This approach reduces the overall miss rate and improves the performance of the parallelized applications.

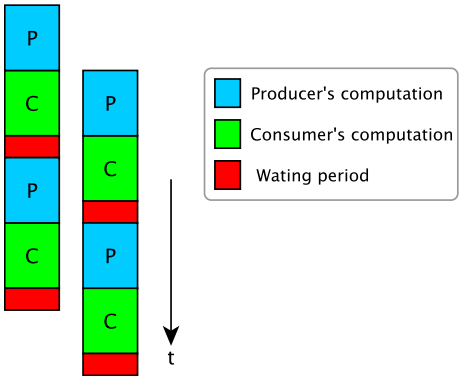


Figure 2.6: Polymorphic Threads model (source: [VJ07]).

2.3 Code Transformations

In order to explore and support parallel execution of computing stages (e.g., loops), the data dependencies between the tasks need to be determined. An application can be split into a producer-consumer pair and then map sections of the code into each core. Several recent studies investigated the difficulties of decomposing one application into many tasks and exploiting the parallelization among these tasks. Also, other studies focused on the detection of data dependencies between tasks. Here, we classify these studies into three main groups: the studies based on compiler support, using directive-driven programming models, and programming languages.

In order to use the benefits of multicore architectures, an application needs to be mapped into a multicore architecture to provide higher performance. Code transformations and task partitioning for multicore architectures are one of the most widely studied topics. Most approaches have focused on identification of the pieces of code (e.g., loops) which can be fully executed in parallel. As traditional programming languages are not suitable for multicore architectures, many studies have been done to provide and to improve parallel programming languages. However, the number of approaches which can identify the level of parallelism are very limited. Therefore, there is a strong need for techniques which can help the developer to manually partition an application to provide higher performance in multicore architectures. For instance, Larsen et al. [LKM11] introduced two compiler directives which help the programmers to express the data dependencies between tasks. In their experiments, the compiler directives can enable reductions of 40% to 57% in potential

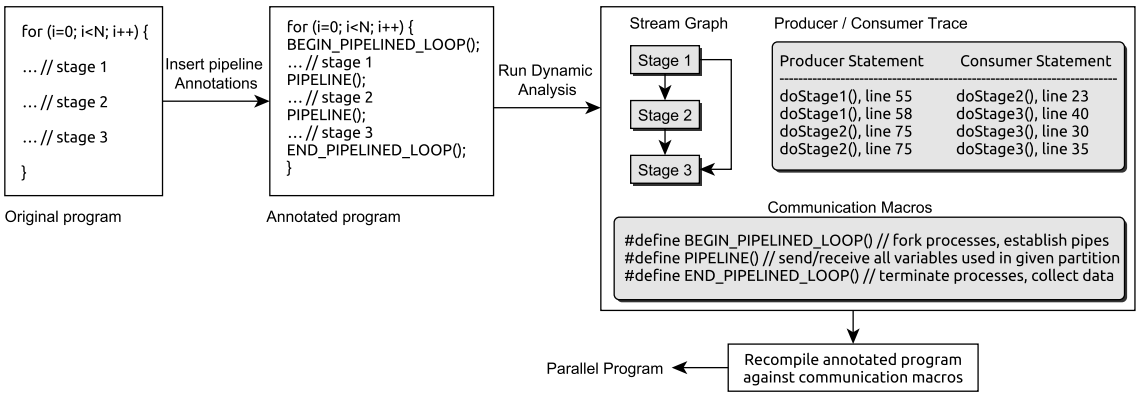


Figure 2.7: An overview of the coarse-grained pipeline parallelism approach in high-level languages (source: [TCA07]).

dependencies of the program.

Thies et al. [TCA07] present an approach to exploit coarse-grained pipeline parallelism in high-level languages such as C. The authors use streaming applications such as MPEG-2 decoding, MP3 decoding, GMTI radar processing, and three standard performance evaluation corporation (SPEC) benchmarks [SPE15] with regular flow of data between tasks. The output of their approach is a stream graph of the application and a set of macros to provide communication between tasks and to parallelize the program. In this study, to provide TaLP the programmer needs to determine the boundaries of pipelining stages and then insert pipeline annotations which are responsible for recording all communications across boundaries between the stages of the C program (see, Figure 2.7).

The pipeline annotations express the pipes and are responsible for sending and receiving all variables used in the given computing stage and finally terminate the computations and collect data. If the programmer is satisfied with the parallelism of the program (presented in the stream graph), he/she needs to recompile the annotated program against a set of macros that are emitted by their analysis tool. If the programmer is not satisfied with the parallelism analyzed by the tool, the annotation needs to be moved to eliminate cyclic dependencies between stages. In this approach, the authors achieved a mean speedup of 2.78× over a 4-core architecture containing two AMD Opteron 270 dual-core processors.

Gordon et al. [GTA06] used StreamIt, an architecture-independent programming

language for high-performance streaming applications [TKA02]), to map streaming applications into a 16-core RAW architecture [WTS+97]. In this approach, the authors provide a robust compiler system using a combination of task, data and pipeline parallelism techniques to achieve high multicore performance across a range of input programs. In the StreamIt language, a program is represented as a set of autonomous actors which communicate through FIFO data channels. This compiler-based system uses two parallelism techniques, one for data parallelism and other for pipeline parallelism. When targeting the RAW architecture, a coarse-grained data parallelism achieved a mean speedup of 9.9× over a single core and 4.4× over a task-parallel baseline. Similarly, coarse-grained software pipelining (instruction-level) achieved a 7.7× speedup over a single core and a 3.4× speedup over a task parallel baseline.

Note that pipeline parallelism in [GTA06] is applied to chains of producers and consumers which are directly connected in the stream graph. Similarly, in the previous work in [TKA02], they also exploited pipeline parallelism by mapping clusters of producers and consumers to different cores. In terms of high-level language annotations, Benkner et al. [BBM+12] also proposed the use of C/C++ to develop pipeline applications and to specify pipeline patterns on heterogeneous many-core architectures. In this approach, the authors also provide a source-to-source compiler which translates the pipelined applications to an object-oriented coordination layer on top of a heterogeneous task-based runtime system. They use a face detection benchmark and implement it in a pipelined manner using OpenCV [OPE16]. The results considering the use of one CPU and one GPU show speedup improvements of 3.14× compared to the execution time of using only one CPU.

In image processing pipelines, Ragan-Kelley [RKAP+12, RKBA+13] presented a language and a domain-specific language (DSL) compiler called *Halide*, for optimizing parallelism in image processing applications (a Laplacian filter with 99 stages is used as example). Halide is an open-source domain-specific language to express complex image processing pipelines. The results show that using Halide programs improves the performance up to 5× than optimized hand-written programs implemented in C and CUDA.

2.4 High-level Synthesis (HLS) for TaLP

Nowadays, the complexity of applications and the hardware resources available in contemporary FPGAs are very high. It is expected those complexities continue to increase this poses many challenges when considering the mapping of applications to FPGAs. Using high-level abstractions and synthesis (HLS) [1] methods can increase the productivity of designers by increasing the abstraction level in both software and hardware domains [CM08].

HLS tools rely on a portfolio of code transformations, optimizations, and on the exploration of customization parallelism and pipelining.

Commercial HLS tools provide some support to TaLP. For instance, *Catapult-C* [Bol08] and *Vivado HLS* [Xil12] are able to generate hardware with TaLP. In Vivado HLS, the optimization is named as *dataflow pipelining* and it is achieved by adding channels between blocks (functions or loops). Similarly, Catapult-C provides TaLP using *hierarchical synthesis* and local memories are placed between function datapaths.

The approach proposed by Rodrigues et al. [RCD07] considers an inter-stage buffer with empty/flag and local storage, but without access to the main memory to synchronize the execution of hardware datapaths of producer and consumer tasks in a data-driven way. In this approach, each computing stage is translated to a specific datapath and Finite State Machine (FSM) which interfaces to the inter-stage buffer. Their approach deals with both in-order and out-of-order P/C communication patterns, but critically needs to statically determine the worst-case size of the local storage of the inter-stage buffer in order to avoid deadlocks. They use register-transfer level (RTL) cycle accurate simulations to determine the size of the local buffer and thus their approach depends on the simulation of the worst cases. Their scheme was used in the context of a compiler of software programming languages to specific customizable architectures suitable for implementation in FPGAs.

[1] HLS is also called behavioral and architectural-level synthesis.

2.5 Profiling Tools for TaLP

Providing auto-parallelize programs with a complex control and data flow is one of the most challenging issues in parallel computing. Many studies have attempted to identify the potential parallelism in sequential programs. For instance, in [RVD10] the authors present a tool that consists of a profile-driven discovery of parallelism and also of automatic program transformations. The tool can analyze the memory dependencies in order to discover thread-level parallelism in a given program. However, the tool does not address TaLP.

Recently, many researchers such as [RVD10, KC12, LAUH+15] have focused on exposing dependencies using profiling techniques to identify the potential parallelism in sequential programs. For instance, the most recent study in [LAUH+15] present a profiling tool called *DiscoPoP (Discovery of Potential Parallelism)* which is based on identifying computational units following the read-compute-write pattern. This tool performs various types of static and dynamic analyses, and profiles the control and data dependencies of the input program. DiscoPoP also covers both loop and task parallelism. In the experiments, they use NAS parallel benchmarks [NAS15] and PARSEC [BKSL08]. In this approach, the authors found 92.5% of the parallel loops in NAS Parallel Benchmark programs and they achieved up to 2.67× speedup for independent tasks and up to 3.62× when using pipelining considering a maximum of four threads.

In terms of thread-level parallelism, Sean et al. [RVD10] presented a profile-based tool to determine pipelining parallelism potential in sequential programs. The authors evaluate their results by measuring the speedup on real hardware on a 32-thread Sun UltraSPARC T1 and on a 8-thread Intel i7 quad-core. The results show speedup improvements of 5.18× for bzip2 compression and 11.8× for an MPEG2-encoder on a Sun UltraSPARC T1.

The Pareon Profile from Vector Fabrics [vec16] is one of the commercial profiling tools which support TaLP. This tool uses a graphical user interface (GUI) and determines which serial code section has the capability for parallelization. The Pareon Profile tool can analyze the data dependencies between computing stages

and characterize them by type and whether if it is possible to split the task or not. The main advantage of the Pareon Profile tool is that the user can determine the capabilities to estimate parallelization speedups for the sequential applications. Note that the speedup estimation is based on a detailed execution time model, including processor instruction and memory cycles, inter-thread synchronization overhead, and task scheduling.

For example, by selecting one of the loops in a program, the tool can show whether a selected loop can be parallelized using data partitioning by considering the dependencies between the selected loop and other loops of the program.

Although these tools have many features to determine the capability of parallelism for each task, they might not support all types of multicore architectures. In addition, these tools (e.g., Pareon Profile) cannot partition the program properly into different cores by considering the pipeline balancing between computing stages. Note that the most advantages of the profiling tools is to the best of our knowledge to determine the part of the program which have the potential parallelism and also detecting the dependencies between computing stages when considering TaLP as well.

2.6 Parallel Models of Computation (MoC)

Recently, many studies (see, e.g., [NMSD09, CTLA12, ZNS13, MKTdK07]) have been focused on the use of traditional computation models such as Kahn Process Network (KPN) [Kah74], Synchronous Data Flow (SDF) [LM87b], and Communicating Sequential Processes [Hoa78]. In these studies, the models have been considered as graphs, where nodes represent units of computation and edges represent unbounded FIFO communication channels.

In order to provide a better load performance and scheduling, several approaches have been focused on modeling pipeline applications with SDF [LM87a] graphs and on determining the data dependencies between tasks. The SDF graphs are adopted to describe mostly streaming applications and partitioning them for multicore architectures. However, it does not mean that every application (e.g., a C program)

can be translated to an equivalent SDF graph. Because of these, in [LF13] the authors present an approach named Pipeline Application Modeling (PAM), which is a methodology to build an SDF graph describing all the aspects of a pipeline application including mapping, scheduling and the pipeline assignment.

Similarly, for the KPN computation model, several approaches have attempted to map KPN applications onto different multicore architectures such as on Intel IXP [MKWS07] or Cell BE [NMSD09]. For instance, Meijer et al. [MKTdK07] presented a process splitting transformation for KPN computation model. The splitting process in this study consists of a producer, a transformer, and a consumer. To define the data dependencies between a producer- consumer pair, they use the Compaan compiler [KRD00]. They address static affine nested loop programs and the compiler is able to analyze the data communication patterns between producers and consumers. In addition, for each subset of the process iteration space (partitioned piece), they use a unique FIFO channel. If the nested loop can be partitioned into four parts, the communication channel between producer-consumer pairs requires four separated FIFOs. The results show a 21% performance improvement by reducing the total execution time of the JPEG decoder application using GCC for the compilation.

2.7 Overview

Table 2.1 presents an overview of the related work. This table summarizes the main contributions, tools, techniques and technologies used in the approaches presented.

The first category of approaches such as [BF99, LW05, GMV08, TKD03b, TKD02, TKD05, Smi86, Smi82] use the producer-consumer communication model to improve the execution performance of tasks. When considering the importance of synchronization and communication between the producer and consumer, these studies can be categorized into two main groups: first, the studies focused on the synchronization improvements between cores; second, the studies researching data streaming communications between P/C pairs over channels.

In the case of P/C synchronizations, the empty/full bit scheme, such as the one presented in [Smi82], as a model to synchronize the communication in multicore architectures is popular. Many approaches use the flag-based approach in a variety of implementations such as register-based synchronization (RBS) [FJ08], table-based and hash-based indexing approaches. For instance, the RSB scheme uses registers to implement the empty/full bit flag. However, in this scheme, the limitation of the number of shared registers between P/C pairs might reduce the communication transfers between P/C pairs. The approach proposed in this thesis use the hash-based indexing approach to reduce the size of the inter-stage buffer between P/C pairs when using an empty/full flag as a synchronization model. Note that the flag-based synchronization approach can be implemented also in different memory architectures such as shared memory and/or distributed memory in a multicore system.

Although some of the studies use distributed memory (see, e.g., [BF99]) or a cluster-based memory (see, e.g., [YXY+14]) to provide the synchronization between cores, we consider the studies which use shared memory for data communication and deal with the P/C communication model to improve the performance of the applications (see, e.g., [BF99, MKH+13, MKH+14]) and also the studies which deal with inter-task data communication using shared memory and the P/C pair communication model (e.g., [LW05, YXY+14]).

The main contribution of these studies (see, e.g., [MKH+13, MKH+14, LW05] and [GMV08]) consist of achieving task-level pipelining on multicore architectures, improving the performance of streaming applications using two cores to implement the producer-consumer model, reducing the synchronization and communication overhead and data transfer costs between cores; and also providing pipelining parallelism between cores in multicore architectures.

In the context of streaming data between P/C pairs over channels, several approaches, such as [Smi82, Smi86, TKD02, ZSHD02, TKD03b, ZHD03, TKD05] and [ZHX+15], have used FIFOs as a communication and synchronization channel between the tasks using a producer-consumer model. The FIFO channels can also be implemented in software and using shared memory to communicate between cores

(see, e.g., [LW05]). Note that FIFO channels are suitable when the sequence of producing data is the same than the sequence of consuming data.

When considering the bottleneck of using FIFO channels for out-of-order data communications between P/C pairs, some studies, such as the ones presented in [TKD02, TKD03b, TKD05], use a reordering mechanism to solve the problem of out-of-order tasks when using a FIFO channel in the context of P/C pairs. However, using the reordering mechanism requires an extra memory to store the data which cannot be consumed directly by the consumer. Although reordering the pattern of producing and consuming data in compile-time might be a solution for some application with out-of-order data communication between tasks, there might be cases when a full reordering is not possible and out-of-order communication may not be fully avoided. Note that in our approach, we do not reorder the way data are produced and/or consumed. This is however orthogonal to our approach.

Other recent studies (see, e.g., [ZHX+15]) introduced an approach to provide an automate compilation flow mapping general C programs into full system designs on different FPGA platforms (called *CMOST*). Although this approach use block FIFOs to support out-of-order data communication between P/C pairs, the data communications between different block FIFOs are in-order. The bottleneck of this approach is when the consumer requests data from a previous produced block and/or the next produced block of the current block of the consumer data request. In this case, a reordering mechanism and an extra memory is still required. In our case, we use both local and main memory without the need to reorder the way data are produced and/or consumed.

The second category of the related work, such as [CHW00, CW00, DG07, VJ07] and [RCD07], deal with loop pipelining techniques and pipelining sequences of loops in multicore architectures. The main contributions of these studies are summarized as follows: pipelining the execution of applications using an inter-stage buffer between the stages; assigning, scheduling and mapping the tasks to each core at compile time; supporting different types of loops and pipelining their execution; parallelizing the applications on different multicore architectures (e.g., SMT and CMP) and also improving their performance by reducing the overall miss rates.

We also described other studies which support parallel execution of computing stages. These studies focused on detecting the dependencies between tasks and also finding the hotspots of the program (see, e.g., [LKM11]). In general, the contribution of these studies can be summarized as follows: detecting the dependencies between tasks at runtime or compile time; finding pipeline parallelism potential in a sequential program; providing automatic profiling tools which can also present some properties and impact of pipeline parallelism; and tracking the data communication between P/C pairs (see, e.g., [RVD10, KC12, LAUH+15]).

Table 2.1: Summary of the related work.

Authors	Category	Contributions	Tools/Technologies/Techniques	Findings/Results
Byrd et al. [BF99]	Synchronization: shared memory	• Reduce communication latency in bus-based SMPs by sending data to the consumer as soon as they are produced • Using the producer-consumer pair model	• Use producer initiated mechanism called StreamLine which is a cache based message passing mechanism • Use distributed shared memory multiprocessors	Provides good performance on the benchmarks with regular communication patterns between the producer and the consumer
Miyanjima et al. [MKH+13, MKH+14]	Synchronization: shared memory	• Achieve task-level pipelining on multiple accelerators	• Use a high performance communication channel (PEACH2) • Assign tasks to the accelerators and input data can be computed using a sequence of GPUs in a pipeline manner • Process 100 images (1280×720 pixel) using Sobel image filter	52% speedup compared to a single GPU by implementing TaLP using a shared memory synchronization mechanism
Bei Li et al. [LW05]	Synchronization: shared memory	• Provide an efficient implementation of TTL inter-task communication on CAKE tile architecture • Improve the performance of streaming applications • Provide a software solution to reduce the synchronization overhead and data transfer costs to achieve the performance in total communication time	• Use two cores to implement the producer-consumer model • Implementing shared memory using ordered FIFO technique • Use CakeSim simulation framework based on TSS model as a cycle-accurate C language used in Philips to evaluate the performance using the CAKE tile architecture	80% improvements on latency per token transfer (CPT) and reduce the total clock cycles of running the JPEG decoder application by 23%
Zhiyi et al. [YXY+14]	Synchronization: shared memory and message passing	• Achieve a higher performance of signal processing applications such as LDPC decoder, a 3780-point FFT module, an H.264 decoder and an LTE channel estimator	• Use cluster-based memory hierarchy for embedded applications • Use a printed circuit board with the prototype and also an FPGA board to test the processor	A 16-Core Processor with shared memory and Message-Passing Communications in 65 nm COMS.
Giacomoni et al. [GMV08]	Synchronization: shared memory	• Provide efficient Pipeline Parallelism between cores • Provide a high-rate core-to-core software buffering communication mechanism for multi-threaded pipeline parallel applications • Very low communication overhead between processors	• Use FastForward which is an optimization technique of single-producer/single-consumer concurrent lock-free (CLF) queues • Evaluation on multicore architectures • Use 2.0 GHz dual-processor dual-core AMD Opteron 270	Provides faster (up to $4\times$ times) speedup for pipelining fine-grained stages compared to the other solutions such as Lamport's CLF queues (200 ns per operation)

Table 2.1: The summary of the related work.

Authors	Category	Contributions	Tools/Technologies/Techniques	Findings/Results
Ziegler et al. [ZSHD02, ZHD03]	Synchronization: FIFO-based	• Overlap some execution steps of sequences of loops or functions • Describe an implementation of several parallelizing compiler analysis techniques and transformations required to automatically design platform and application-specific pipelines, which have been extended to map computations onto FPGA-based architectures	• Loop unrolling, data reuse, data layout, communication and pipelining analysis • Using DEFACTO system which combines parallelizing compiler technology from the Stanford SUIF compiler • Use Xilinx Virtex FPGA for platform implementations • Xilinx Foundation tools for the place-and-route phase • Communicate data to subsequent stages using a coarse-grained FIFO mechanism	HLS synthesis results of the compiler-optimized pipeline stages for a vision application with three stages. The results are presented for different unroll factors of the innermost loop of these three stages.
Turjan et al. [TKD03b, TKD02, TKD05]	Synchronization: FIFO-based	• Solve the data communication problem for out-of-order tasks • Provide task-level pipelining using the producer-consumer model	• A compiler-based approach using FIFO buffers between tasks • Exploiting reordering mechanisms to solve the problem of out-of-order data communications between tasks • Use Ehrhart and Polyhedral theory [JB07] for reordering • Assume multiple unbounded FIFO channels for synchronization and communication between tasks	A novel compile time technique for detecting whether a FIFO or additional reordering mechanism is required in the linearization step. Provide an implementation of reordering mechanism to give a lower bound on the reordering memory.
Zhang et al. [ZHX+15]	Synchronization: FIFO-based / block FIFOs	• Provide an automated compilation flow mapping general C programs into full system designs on different FPGA platforms (called CMOST) • Provide a unified abstraction model for combination of different microarchitecture optimization schemes using customization, mapping, scheduling and transformation	• Block-based data streaming technique to provide data communication between P/C pairs • Introduce block FIFOs which are an extension of the traditional streaming framework (FIFO channel) to solve out-of-order data communications • Measure speedups and energy on Xilinx Virtex-7 (VC707) and compare to a 6-core CPU (Intel Xeon E5-2640) • Use 5 real applications such as MPEG, NAMD, Smith Waterman, Black Scholes and Medical Imaging	Obtained over 8× speedups for MPEG, NAND and Smith Waterman benchmarks, and over 1.1× for Black Scholes and Medical Imaging benchmarks.

Table 2.1: The summary of the related work.

Authors	Category	Contributions	Tools/Technologies/Techniques	Findings/Results
Smith [Smi86, Smi82]	Synchronization: FIFO-based and Flag-based	Analyze and determine the communication needed between two processors using the producer-consumer model	• HEP computer systems • Using empty/full bit flag to provide mutual exclusion between processors in producer-consumer model	In the context of design space exploration and concerning speedups only one example with a speedup of 1.76×.
Fide et al. [FJ08]	Synchronization: Flag-based/ Register-based	• Register-Based Synchronization to avoid spin waits in multithreaded applications • Reduce miss rates, coherence traffic • Reduce the execution time of the applications and save power	• Register-Based Synchronization • Data Communications via Prepushing • Using Red-Black Solver, Finite-Deference Time-Domain (FDTD) and ARC4 Stream cipher benchmarks • Simulation Environments including Simics, GEMS Ruby and a multicore system using 2 GHz UltraSPARC III processors	RB Solver-RBS achieved $2-5\%$ access timer per iteration, FDTD-RBS achieved $6-11\%$ per iteration, ARC4-RBS achieved negligible.
Bardizbanyan et al. [BGW+13]	Synchronization: Flag-based/ table-based	• Reduce L1D energy by capturing many data memory references in the tagless access buffer (TAB) • Improve performance by prefetching cache lines into the TAB • Exploit amenable access patterns of the TAB-allocated memory references to eliminate unnecessary data transfers between memory hierarchy levels	• Use 20 benchmarks from the MiBench benchmark suite such as automative, consumer, network, office, security and telecommunication benchmarks • Use VPO compiler • Use SimpleScalar simulator	Total data-access energy usage is reduced by 34.7% with four TAB entries. The four-entry TAB configuration reduces energy in the L1D and DTLB by 35.4% and 41.9%.
Givargis [Giv06]	Synchronization: Flag-based and table-based	• Provide a zero-cost hash-functions for cache indexing • Avoid adding any overhead in terms of area or delay	• Bit positions are determined for indexing the cache aiming power consumption reduction by reducing cache misses • Use integer SPEC CPU 2000 benchmarks	Up to 45% reduction for data traces and up to 31% reduction for instruction traces in cache misses. The results show an average improvement of 14.5% for the Powerstone benchmarks and an average improvement of 15.2% for the SPEC'00 benchmarks.

Table 2.1: The summary of the related work.

Authors	Category	Contributions	Tools/Technologies/Techniques	Findings/Results
Callahan et al. [CHW00, CW00]	Software/Loop pipelining	• Present a scheme for pipelining the hardware execution of a variety of loops • techniques to support pipelined execution of loops on the co-processors	• Use Garp-C compiler and architecture for the implementation • Use the reconfigurable arrays • Only one benchmark, wavelet image encoding have been used in the results	The results show the speedup improvement of 87% compared with the original software sequential execution time. The results shown are from processing a 256×256 pixel image.
Douillet et al. [DG07]	Software/Loop pipelining	• Propose a solution to generate and extract threads, and to schedule instructions • Assign the threads to each core automatically in the context of multicore architectures	• Use software pipelining to leverage the multiple cores in a single chip • The technique can be applied to any parallel and non-parallel loop nest originally written in sequential language • Use Single-dimension Software Pipelining (SSP) to generate the SWP schedule • Using Open64 compiler and re-targeted for the IBM Cyclops multicore architecture for the evaluations	Experimental results shows that this approach scales up well when the number of thread units increases. The implementation uses a very light-weight synchronization method with only standard instructions of the IBM Cyclops64 architecture
Vadlamani et al. [VJ07]	Software/Loop pipelining	• Parallelize applications for Simultaneous Multi-threading (SMT) and Chip Multiprocessors (CMP) • Reduce the overall miss rates and improve the performance of the parallelized applications	• Use Synchronized Pipelined Parallelism Model (SPPM), which uses the shared cache as a high-speed communication channel between producer and consumer pairs • Develop C2CBench tool to evaluate the performance of the storage controllers at different levels of the memory hierarchy under varying workload conditions • Measure the overhead of maintaining cache coherence in parallel microprocessors • Use Red-Black Solver, Finite-Deference Time-Domain (FDTD), ARC4 Stream Cipher, and The Pipelined Equation Solver (EQN) as benchmarks	Performance improvements of the benchmarks using SDM, SPPM and Polymorphic threads on different architectures such as Opteron, Xeon and Core Due.

Table 2.1: The summary of the related work.

Authors	Category	Contributions	Tools/Technologies/Techniques	Findings/Results
Rodrigues et al. [RCD07]	Software/Loop pipelining	• Present a technique for pipeline sequences of data-dependent loops using fine-grained synchronization • Describe a hardware scheme and an analysis to reduce the size of memory buffers for inter-stage pipelining • Describe the application of the technique when compiling imperative programming languages to FPGAs	• A data-driven approach • Each computing stage is translated to a specific datapath and Finite State Machine (FSM) which interfaces to the inter-stage buffer • Use RTL cycle accurate simulations to determine the size of local buffer • Use the Nau compiler based on Galadriel/Nenya framework and also RTL simulation environment	The experimental results reveal noticeable performance improvements and buffer size reductions for a number of benchmarks such as FDCT and Sobel over traditional approaches
Larsen et al. [LKM11]	Code Transformations: Detecting dependencies	• Introduce two compiler directives: taskshare and depends which provide additional data dependence information at compile time • Detect the dependencies at runtime if it fails at compile time	• Use a micro benchmark, and three soft real-time embedded codes to evaluate the impact of the directives of the compiler • Use task graphs	The compiler directives can enable a $40\% - 57\%$ reduction in potential dependencies of the program.
Sean et al. [RVD10]	Code Transformations: Detecting dependencies	• Present a profiling tool for discovering thread-level parallelism • Find pipeline parallelism in sequential programs and coarse-grain parallelism in the program's outer loops	• Use several MiBench, SPEC2000 integer and BioPerf benchmarks • Measure the speedup on the real hardware on a 32-thread Sun UltraSPARC T1 and on 8-thread Intel i7 quad-core	Speedups of 5.18× for bzip2 compression and 11.8 for the MPEG2-encoder on a Sun UltraSPARC T1
Thies et al. [TCA07]	Code Transformations: Detecting dependencies	• Show the stability of streaming applications • Define an API for indicating the potential of parallelism in the program • A dynamic tool to track P/C communications between coarse-grained program partitions	• Use streaming applications such as MPEG-2 decoding, MP3 decoding, GMTI radar processing, and three SPEC benchmarks with the regular flows of data between tasks • Extract information from stream graphs of each application	Achieved a 2.78× mean speedup on a 4-core architecture containing two AMD Opteron 270 dual-core processors

CHAPTER 3

Task-Level Pipelining (TaLP)

Chapter Outline

TECHNIQUES to speedup processing are becoming increasingly important. In order to achieve parallel execution of software, the hardware has to support the simultaneous execution of multiple tasks. Multicore-based architectures provide hardware platforms suitable to accelerate the execution of applications by supporting different forms of parallel execution. There are many reasons for moving to multicore architectures. One fundamental reason is that the serial microprocessor processing speed is reaching a physical limit for increasing the clock frequency. Therefore, the processor manufacturers need to focus on a better support for multithreading such as the one provided by multicore processors. In addition, software developers are also forced to develop massively multithreaded programs as a way to better use the multicore processors.

One of the possibilities to efficiently use the advantages of multicore architectures is the use of parallel programming models and parallelization techniques to

accelerate the execution of applications. The parallel execution of applications can be performed by different levels of parallelism such as data-level, task-level, and pipeline parallelism [HP11]. The parallel programming models such as shared memory and distributed memory models, also provide opportunities for parallel execution of tasks in multicore architectures.

Task-level pipelining (TaLP) is also an important technique for multicore based systems, especially when dealing with applications consisting of producer/consumer *(P/C)* tasks (see, e.g., [KKK+09]). TaLP may provide additional speedups over the ones achieved when exploring other forms of parallelism [HP11]. In the presence of multicore-based systems, TaLP can be achieved by mapping each task to a distinct core and by synchronizing the execution of the tasks according to data availability. By partially overlapping the execution of data-dependent tasks (herein: Computing Stages), TaLP can contribute to overall application performance improvements. In the following sections, we describe the most important concepts which are essential to implement TaLP in multicore systems.

3.1 Partitioning Programs for Multicore Systems

An essential component of a successful embedded multicore implementation must include developing applications in a way to make concurrency available for the system to exploit. Applications can be written from the scratch using parallel programming languages or using legacy code and restructure it to run efficiently on a multicore system. One of the strategies to run a sequential program on a multicore system is partitioning the application into multiple independent stages and then execute each stage in one core concurrently. However, this strategy might not be sufficient when the sequential program has stages with dependencies. Therefore, identifying the dependencies between the stages is also essential to provide parallelism in multicore systems.

Data dependence analysis [KKP+81, Ban88] determines some of the code constraints for parallel execution in multicore architectures. To describe the dependencies, we use here the concepts related to producers and consumers of data. For

example, a dependency means that a consumer of data must wait until the producer has produced the data. Figure 3.1 presents an example of a program partitioned into two sections (A and B). In this example, the data (x) is produced in section A and consumed in section B. As shown, section B is data dependent of section A. In this case, the value of x needs to be communicated from section A to section B.

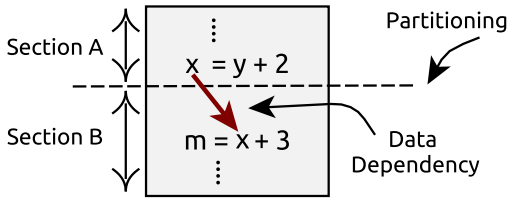


Figure 3.1: An example of partitioning a program with dependencies.

Identifying and locating the dependencies between tasks for any type of program manually is a difficult task. Therefore, we need to limit the scope of the dependencies in the program and the level of parallelism needs to be identified. There are many profiling tools which can help to identify the dependencies between tasks such as Intel Parallel Amplifier [Amp16], DiscoPoP [LAUH+15] and Pareon Profile [vec16]). These tools provide the automatic identification of stages and the *hotspot* of the program. A *hotspot* is a small part of the code which consumes much of the program's execution time.

3.2 Producer-Consumer (P/C) Pairs

To understand the dependencies, we assume a program consisting of producers and consumers of data. The main concept of the P/C pair is based on the fact that a section of a program (e.g., a function or a loop) does calculations and outputs data used by other section of the program: the first section is the producer and the second section is the consumer. In addition, a data dependency between the producer and the consumer means that the consumer before processing must wait until the producer has produced the required data.

Figure 3.2 presents an example of pipelining data-dependent computing stages using a P/C model. As shown, the consumer can start consuming data only when it is available by the producer. For instance, when data (e.g., data 0) is output by the

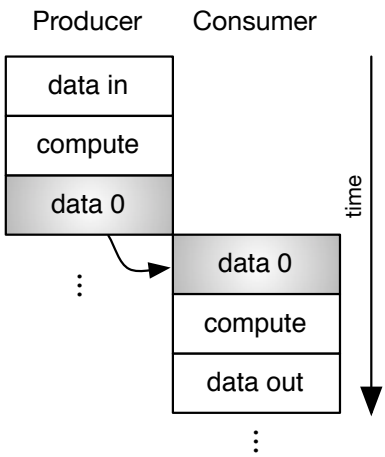


Figure 3.2: An example of pipelining data-dependent computing stages using a P/C model and with the identification of the data communication between the producer and the consumer.

producer, the consumer can immediately consume it and the producer starts producing the next data (data 1) concurrently. Many applications, such as image/video and signal processing, are structured as a sequence of data-dependent computing stages (e.g., consisting of a loop or a set of nested loops), and are thus amenable to pipelining execution [ZSHD02, RCD07]. Using TaLP, a consumer computing stage may start execution, before the end of the producer computing stage, based on data availability. Performance gains can be achieved as the consumer can process data as soon as it becomes available.

In general, each computing stage can have multi-input and multi-output stages. This means each computing stage can be a producer for the next stage, a consumer of the previous stage or a consumer from the previous computing stage and a producer for the next dependent stage. Therefore, we can categorize the dependent stages into three different types with the terms of: *producer stages*, *consumer stages* and *consumer/producer stages*.

The producer computing stages are independent but the output of their computation would be used for the next computing stages. The consumer computing stages are dependent on the previous computing stages and their computation output would not be used by other stages. The consumer/producer computing stages can be dependent on the previous computing stages and the output of their computing stages would be used by the next computing stages.

Figure 3.3 shows three possibilities for the computing stages dependencies. As shown, the *producer stages* have the possibilities of n output ($n \geq 1$) and the *consumer stages* have the possibilities of m input ($m \geq 1$). Also, the *producer/consumer stages* with multi-inputs/multi-outputs, have the possibilities of m input and n output ($m, n \geq 1$).

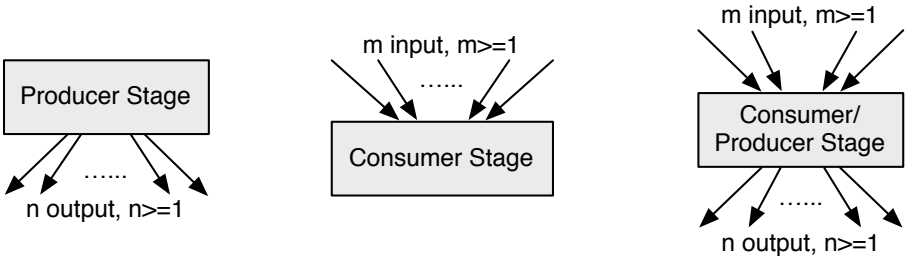


Figure 3.3: The possibilities for computing stages stages.

Figure 3.4 shows the an example of a sequential program with five dependent computing stages. In this example, the *Stage 2* and *Stage 3* are dependent to the *Stage 1*, and the *Stage 4* is only dependent to the *Stage 3*. The *Stage 5* is also dependent to the *Stage 2* and *Stage 4*. Therefore, *Stage 1* is a *producer stage* for the *Stage 2* and *Stage 3*. The *Stage 2*, *Stage 3* and *Stage 4* are identified as *consumer/producer stages* and the *Stage 5* is also a *consumer stage* for the previous stages (*Stage 2* and *Stage 4*).

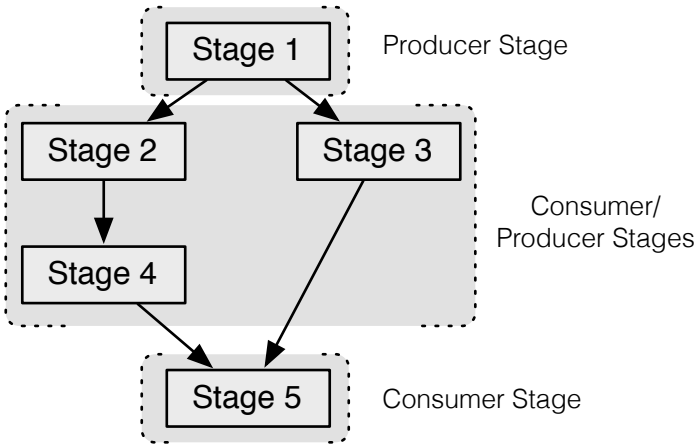


Figure 3.4: The dependency graph of a sequential program with five computing stages.

In the following sections, we describe producer/consumer (P/C) pairs and different types of data communication and synchronization between P/C pairs.

3.2.1 Loops and Dependencies

As previously mentioned, computing stages can consist of a loop, nested loops or a set of loops. To partition the computing stages and to implement them using a P/C pair model, we need to determine the data dependencies between the sequence of loops or nested loops in the code. Computing stages may have one or multiple dependencies. However, here we present an example of two nested loops with one data dependency for simplicity (see Figure 3.5a).

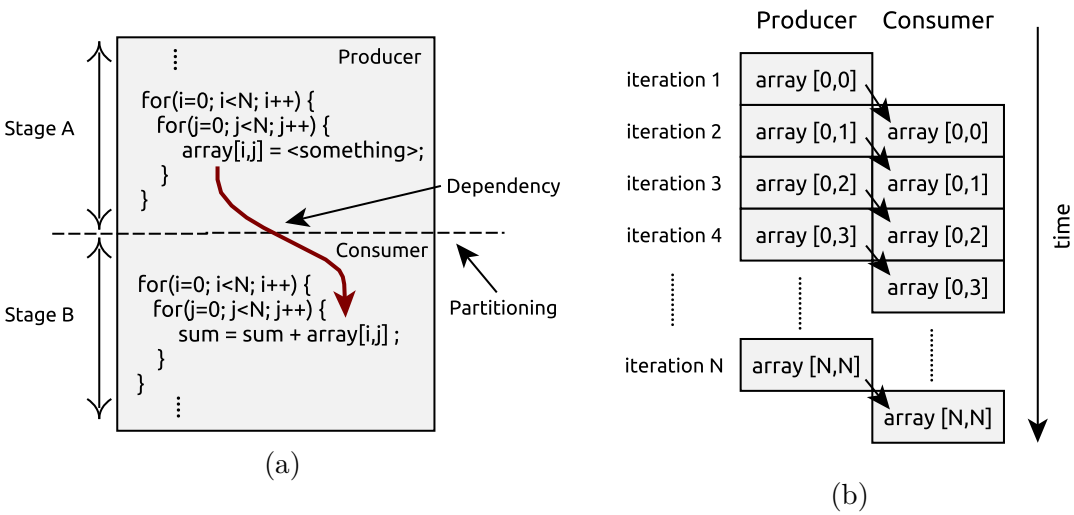


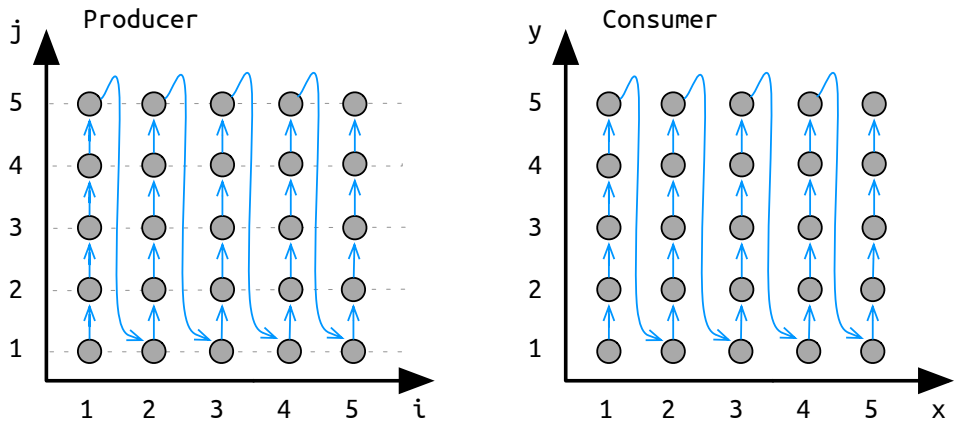
Figure 3.5: An example of pipelining data dependent nested loops using a P/C pair model: (a) Partitioning; (b) Pipelining stages and identification of data communicated between P/C pairs.

Figure 3.5 presents an example of pipelining the sequence of data-dependent loops using a P/C pair model. As shown in Figure 3.5a, the program code is partitioned into two computing stages (A and B). Considering the dependency between these stages, stage B is the consumer of stage A. Figure 3.5b shows the data dependencies of each iteration of the loop between the producer and the consumer using pipelining the execution of both stages which is partially overlapped. For example, as soon as data element $array\ [0,0]$ is output the consumer processes it and waits for the next data element, and after producing data element $array\ [0,0]$ the producer continues producing other data elements such as $array\ [0,1]$ and $array\ [0,2]$.

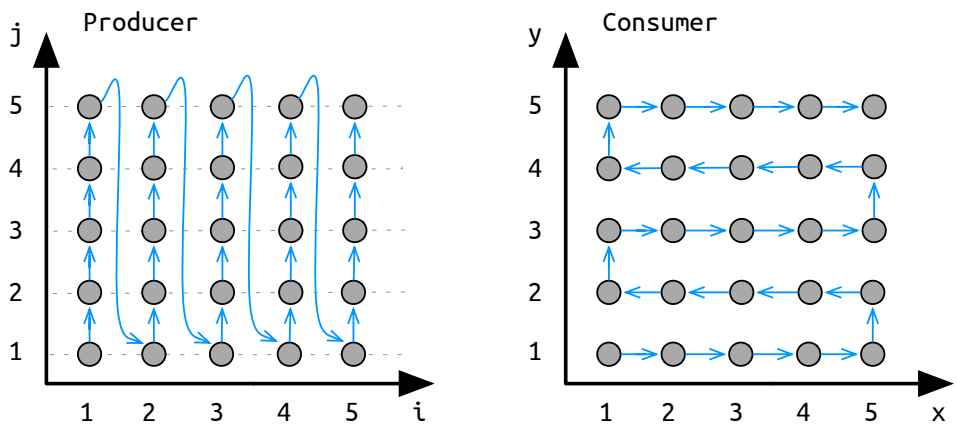
3.2.2 Data Communication Patterns

Producers and consumers can have different data communication patterns. In general, data communication patterns between the producer and the consumer can be classified into two different categories: *in-order* and *out-of-order*.

The data communication pattern between P/C pairs is *in-order* when the sequence of producing data is the same as the sequence of consuming data. Figure 3.5 is an example of an *in-order* data communication pattern between P/C pairs. As shown in this example, the sequences of data (e.g., $array[0,0]$, $array[0,1]$, $array[0,2]$, $array[0,3]$,...) are produced and the same sequences of data are consumed by the consumer. Figure 3.6a shows an example of a generic *in-order* data communication pattern between P/C pairs.



(a)



(b)

Figure 3.6: Examples of different data communication patterns between P/C pairs: (a) *in-order*; (b) *out-of-order*.

On the other hand, when the sequence of producing data is different of the sequence of consuming data, the data communication pattern between P/C pairs is *out-of-order*. We believe that most of the image/video processing applications have *out-of-order* data communication patterns between P/C pairs.

Figure 3.6b presents an example of a generic out-of-order P/C pairs. As shown in this example, the sequences of data (e.g., $[1,1]$, $[1,2]$, $[1,3]$,...) is produced in a different order from the one requested by the consumer (e.g., $[1,1]$, $[2,1]$, $[3,1]$,...). In this case, the consumer needs to wait until the requested data are available. For instance, the consumer can consume the second requested data ($[2,1]$) only when 5 additional sequences of data ($[1,2]$, $[1,3]$, $[1,4]$, $[1,5]$) have been produced by the producer. Note that the distance (herein 5 sequences of data) between the data requested by the consumer and sequences of data have been produced by the producer is depending on the data communication pattern between P/C pairs. For instance, when considering TaLP, this example represents a highly unsuitable case of out-of-order data communication patterns between P/C pairs.

3.2.3 P/C Data Communication Ratio

The ratios of the producer and consumer define the number of times each data element is produced/consumed. If each data element is produced only once, the ratio of the producer is 1. Similarly, if the consumer requests an element just once, the ratio of the consumer is 1. For instance, a P/C pair ratio of $(1:2)$ means that data elements are produced one time and there is at least one element requested/read twice by the consumer. A P/C pair ratio of $(2:1)$ means that there is at least one element produced twice (a maximum) and elements are consumed at most one time. In general, a P/C pair ratio of $(M:N)$ means that there is at least one element produced M times and there is at least one element consumed N times. M and N are represent the maximum ratios of the producer and the consumer respectively.

Figure 3.7 presents an example of in-order and out-of-order P/C pairs with different ratios between the producer and the consumer. The data communication pattern between the producer and the consumer can be grouped in four categories:

$(1:1)$, $(1:C_r)$, $(P_r:1)$ and $(P_r:C_r)$ where $(P_r:C_r; 2 \leq P_r \leq M; 2 \leq C_r \leq N)$, and $(P_r$ and $(1:C_r)$ are the ratios of the producer and the consumer.

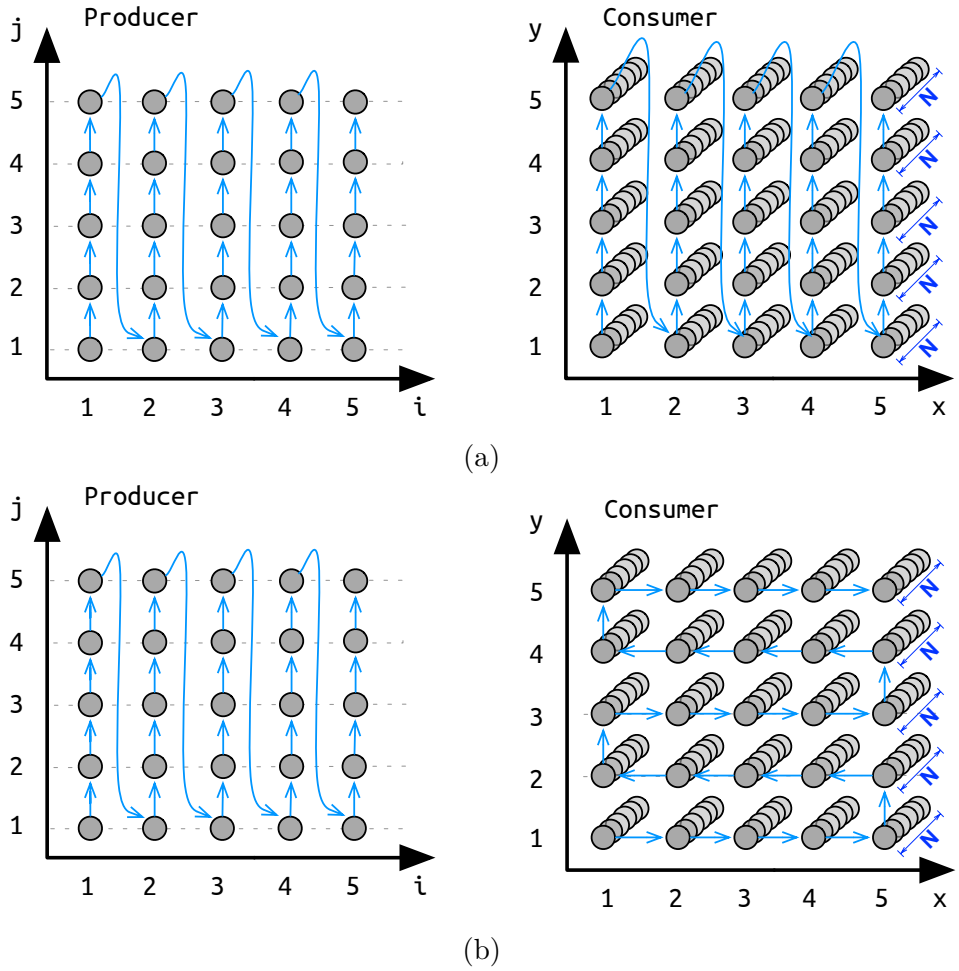


Figure 3.7: Examples of different data communication patterns with the ratio of $(1:N)$ between P/C pairs: (a) in-order; (b) out-of-order.

Figure 3.8 presents the general forms of P/C pair ratios $(M:N)$. Although the producer can produce data elements more than once, in many image/video processing computing stages the P/C pairs ratios are $(1:1)$ or $(1:N)$. Therefore, the approach presented in this thesis is focused on P/C pairs with the ratios such as $(1:1)$ and $(1:N)$.

3.2.4 P/C Communication Schemes

Considering the data dependencies between computing stages, data synchronization between P/C pairs is an important key to provide the correctness of the concurrent

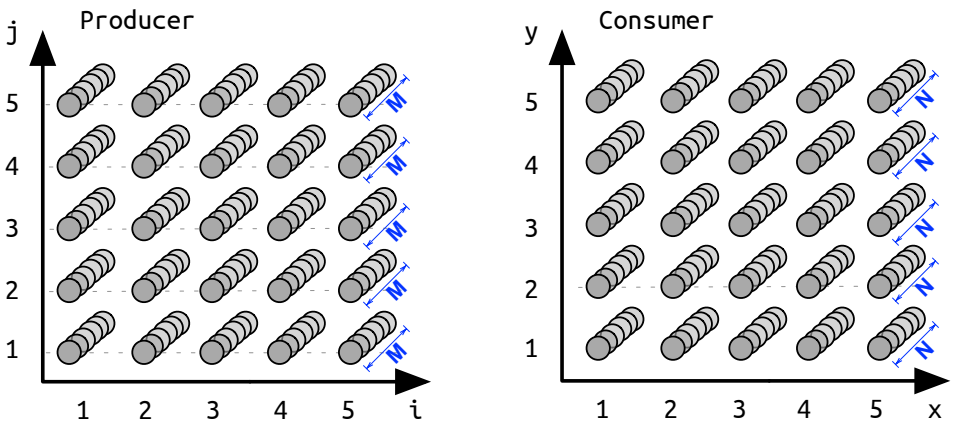


Figure 3.8: A general example of a data communication pattern between the producer and the consumer with the ratio of $(M : N)$.

execution of the computing stages, the validity of the data being communicated between P/C pairs, and to properly allocate limited resources between cores.

There are different ways to synchronize data communication between P/C pairs. According to the granularity, we can classify two types of data synchronization between stages: *fine-grained* and *coarse-grained*. In *fine-grained* data synchronization, each data element is used to synchronize the computing stages. In *coarse-grained* data synchronization, instead of each data element, chunks of data elements or an entire array of elements (e.g., an image) is considered to synchronize the computing stages. The following subsection describes the most common schemes for data synchronization to provide TaLP between P/C pairs in multicore architectures.

FIFO-based

The communication component between P/C pairs can be a simple FIFO. Figure 3.9 presents a fine-grained data synchronization scheme using a FIFO between P/C pairs. The architecture presented here considers a memory shared between cores. Other architectures can be based on the distributed memories (e.g., one per cores). A mix of shared and distributed memories can be also present.

In the architecture presented here, reading and writing from/to the FIFO can be blocked. When the FIFO is full, the producer waits to write into the FIFO. Similarly, when the FIFO is empty, the consumer waits until a data element is written to the FIFO. In this scheme, the producer sends data elements into the FIFO and the

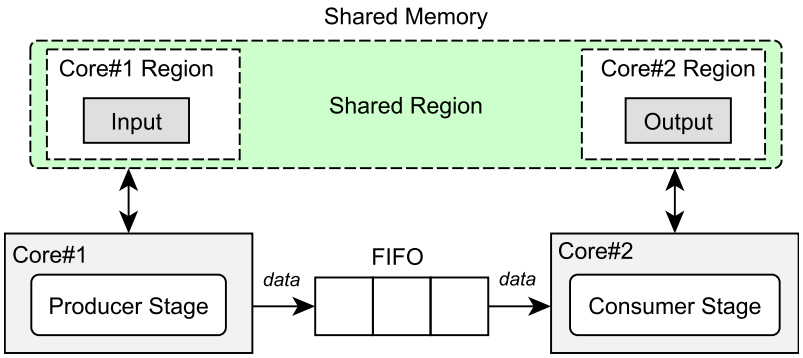


Figure 3.9: Fine-grained data synchronization scheme using a FIFO between P/C pairs and a shared memory multicore architecture.

consumer reads data from the FIFO as soon as it is available. The bottleneck of the communication over FIFO channels is that if the order of consumption is different from the order of the produced data (i.e., when in presence of out-of-order data communication), the producer and the consumer stall and a deadlock can be expected.

Figure 3.10 presents the kernel of a Gray-Histogram code which consists of two computing stages. The first stage transforms an RGB image to a gray image with 256 levels and stores the output into an array. The second stage reads the gray image and determines its histogram. These two stages can be split into producer and consumer tasks. The producer transforms the input image from RGB to gray and the consumer computes the histogram of the gray image.

This kernel has an in-order communication pattern between the producer and the consumer and a ratio of $(1:1)$. In this case, a simple FIFO can be used to communicate data between the two stages and to achieve the fine-grained synchronization between P/C pairs. In this model, the one-dimensional FIFO channel is being accessed by means of a blocking write (*put*) from the producer side and a blocking read (*get*) from the consumer side.

There are some considerations take into account such as the FIFO size and the synchronization delay between the producer and the consumer when using FIFOs. For instance, in the case of out-of-order P/C pairs, using FIFOs requires a conservative determination of the size of the FIFOs to ensure the correctness of data communication between the producer and the consumer. In this communication and

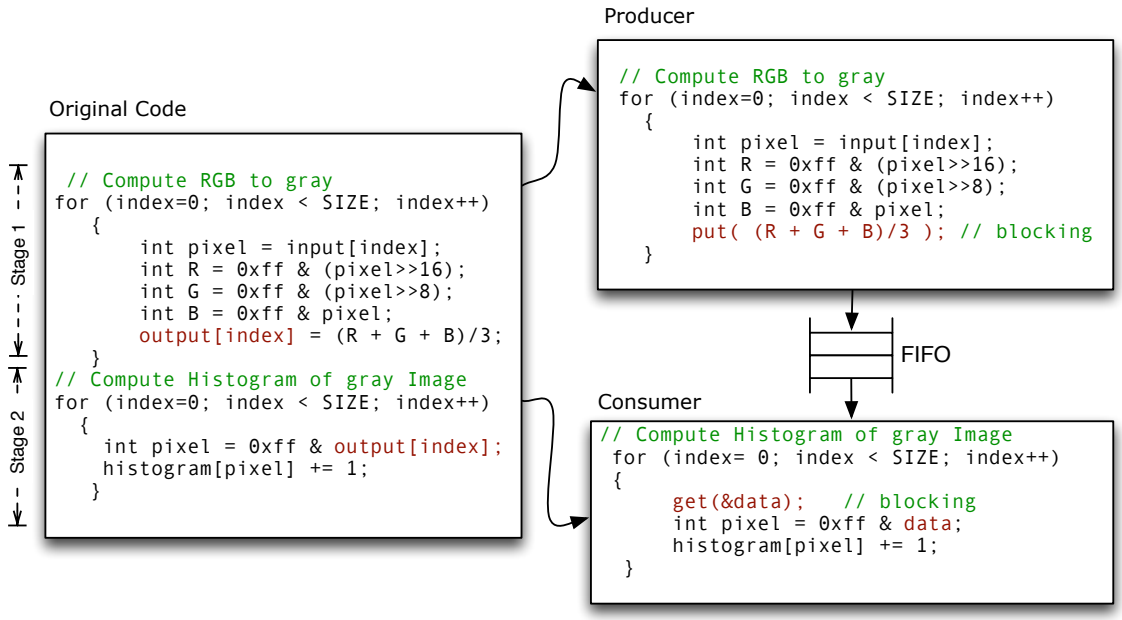


Figure 3.10: Example of a producer/consumer pair using a FIFO channel.

synchronization scheme, all the array elements produced/consumed on each pipelining stage, and thus corresponding to a given FIFO, must be consumed before the data corresponding to another stage is considered. This may require FIFOs with a very large width (or depth), making this approach unsuitable in many cases, as large buffers must be implemented as out-of-chip storage. Note that in some cases of out-of-order producer/consumer pairs the entire data set to be communicated may have to be stored in one FIFO stage, effectively disabling the pipelining of stages. This approach may also need the consumer stores locally the elements of each FIFO stages and process them before getting another set of elements from the FIFO. Another option to implement this approach is the use of multiple FIFO channels with the number given by the size needed in each stage.

Figure 3.11 presents two out-of-order producer/consumer examples of accesses to array elements and the required number of elements for each FIFO stage. In this example, considering the size of each FIFO stage, a set of array element can be stored in each FIFO stages and the consumer can request for the next array elements only when the previous set have been consumed by the consumer.

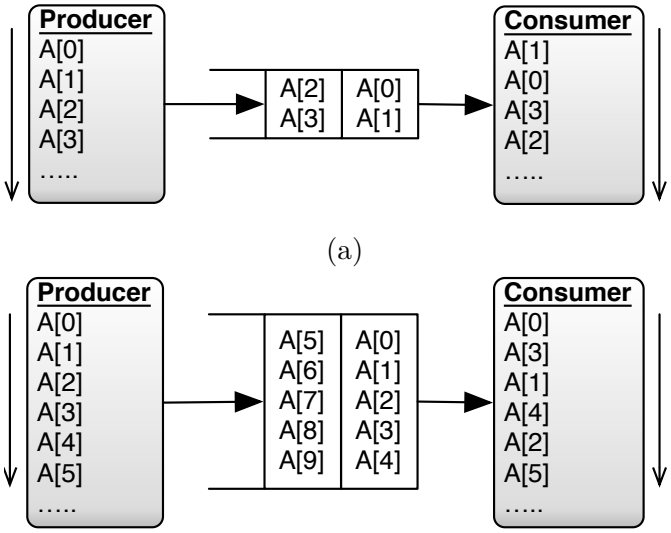


Figure 3.11: Examples of inter-stage scheme based on FIFOs and the number of elements in each stage for out-of-order producer/consumer pairs (a, b).

Shared-Memory Based

Figure 3.12 presents a shared memory based pipelining TaLP scheme. As shown, this scheme is based on a synchronization mechanism which uses shared memory as a storage location for the data being communicated between stages.

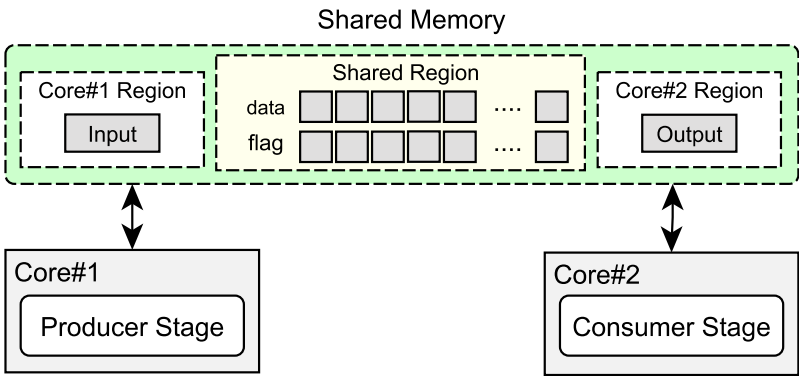


Figure 3.12: Fine-grained data synchronization scheme between P/C pairs using shared memory.

The shared memory stores a produced element or a set of elements using an empty/full bit to establish data synchronization between the producer and the consumer. The producer stores data elements directly in the shared memory and sets to *full*. The consumer checks the flag for each read from the shared memory. If the

requested data from the consumer are not available in the shared memory (flag set to *empty*), the consumer waits until data are available (flag is set to *full*). Figure 3.13 shows the previous example of a simple producer/consumer pair using a flag-based shared memory instead of a FIFO channel.

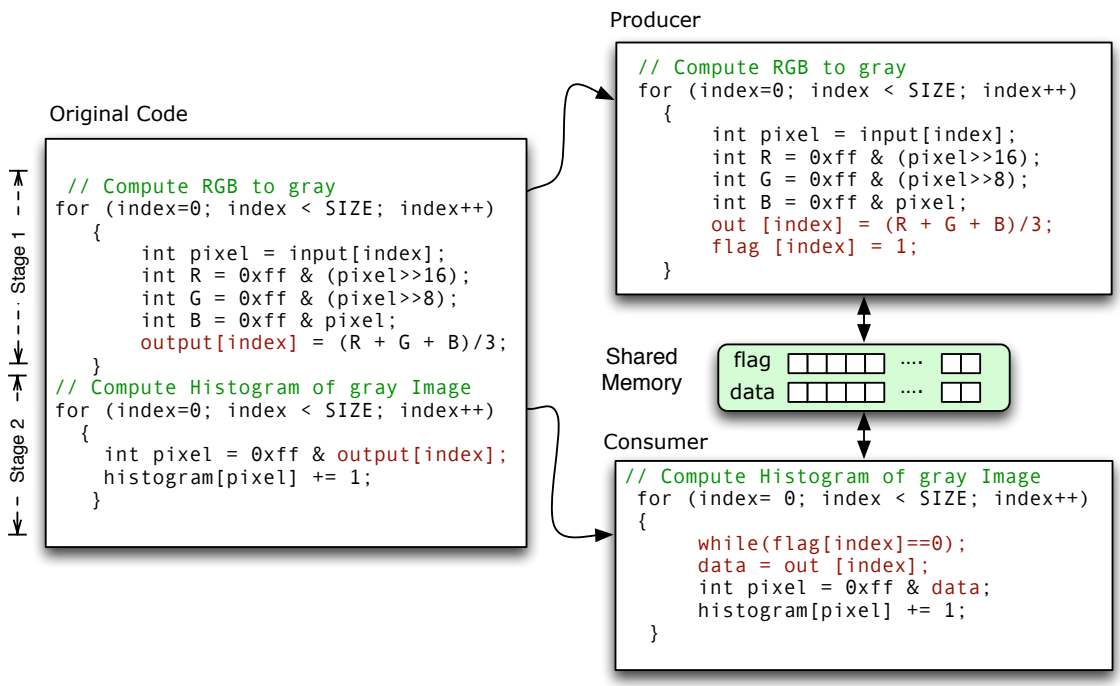


Figure 3.13: Example of producer/consumer pair using a flag-based shared memory.

One of the drawbacks of the implementations of TaLP based on this scheme is the use of a shared memory to provide data for the producer, to communicate data between the producer and the consumer, and to store data produced by the consumer. Being the shared memory a single-port memory, the serialization of concurrent memory accesses occurs and there might be periods of time the P/C cores are waiting for data.

The FIFO-based communication/synchronization scheme described in the previous section, can be also implemented in software using a shared memory architecture as the one presented in Figure 3.12.

The following chapter describes our approach which addresses the main disadvantageous of the two schemes for TaLP described in this section.

3.3 Summary

In this chapter, we presented the main concepts of Task-Level Pipelining (TaLP) and the importance of using TaLP to accelerate the execution of tasks (computing stages) in multicore architectures. We described that to use the advantages of multicore architectures, we need to learn how to partition a sequential program to run in parallel. Due to the partitioning, we described the definition of computing stages in a sequential program and also the dependencies between the computing stages. We used the producer-consumer (P/C) paradigm as a model of synchronization and we described the data dependencies between the producer and consumer pairs.

We described two different types of data communication patterns between P/C pairs: *in-order* and *out-of-order* and four types of data communication ratios between P/C pairs: $(1:1)$, $(1:N)$, $(M:1)$, and $(M:N)$. As the P/C pairs ratios of $(1:1)$ and $(1:N)$ are the most common data communication ratios between P/C pairs in the domain of image and signal processing applications, the work of this thesis is mainly focused on these two P/C ratios. However, for the ratio $(M:1)$, there are some possible approaches to change the ratio of the producer from $(M:1)$ to $(1:1)$. For instance, when the producer outputs a data element M times, the ISB can only store one data. By considering the ratio $(M:1)$, the consumer loads each data one time. Therefore, this approach is not dependent on M whether the value of M is constant or variable for each produced data.

Finally, we presented the two most common multicore architectures to implement TaLP between P/C pairs.

In the next chapter, we present our multicore approach to implement and to provide TaLP.

CHAPTER 4

Our TaLP Approach

Chapter Outline

In this chapter, we describe our approach to provide TaLP for in-order and out-of-order data communication between P/C pairs, including different P/C pair communication ratios. Our approaches are classified into two main groups: fine-grained and coarse-grained approaches. Additionally, we propose a top-down design flow to apply TaLP to sequential applications.

The main concept of our approach is based on a synchronization scheme which uses, for each array element being produced-consumed, a data-buffer storage location and an empty/full bit in the synchronization table. These auxiliary data storage structures have the same size in terms of the number of elements to be stored as the number of elements being communicated between stages. Clearly, such a simple approach may require off-chip storage to store all the data ever being generated and produced. Instead, we make the simple observation that the implementation only needs to have enough storage space for the data items that are in-transit as the storage space can be reused as soon as a given item is consumed. This observation leads to the fundamental issue of how to map a large number of elements to a smaller local memory in order to reduce the number of elements stored in a higher memory layer (e.g., main memory).

We use a hash-based indexing approach as it is a well-known approach for rapid and efficient access to large numbers of sparsely allocated "keys". In addition, as a fully-associated indexing scheme is prohibitively expensive, we have opted for a hash-based indexing approach where the array index is used to compute a single memory address where the corresponding data and empty/full bits will be mapped to the data-buffer and sync-table, respectively.

Figure 4.1 presents a simple hash-based indexing approach. In this example, the data produced and stored into the memory. As soon as a requested data from the consumer (e.g., $A[2]$, the first requested data from the consumer) is available in the local memory, it can be consumed. Note that in this approach, there is no restriction related to the types of the data elements communicated between stages as long as a calculation of the memory position based on the index is provided when needed.

Given the ability to handle out-of-order production and consuming of data items, the use of a many-to-one mapping function raises, however, another subtle issue. As both producer and consumer engage in a double-sided synchronization protocol over data-buffer locations, the producer may require the consumer to read a data item for a specific location before proceeding with the computations. On the other hand, the consumer might be waiting for the producer to generate another data item before attempting to consume the first item the producer is waiting for. This

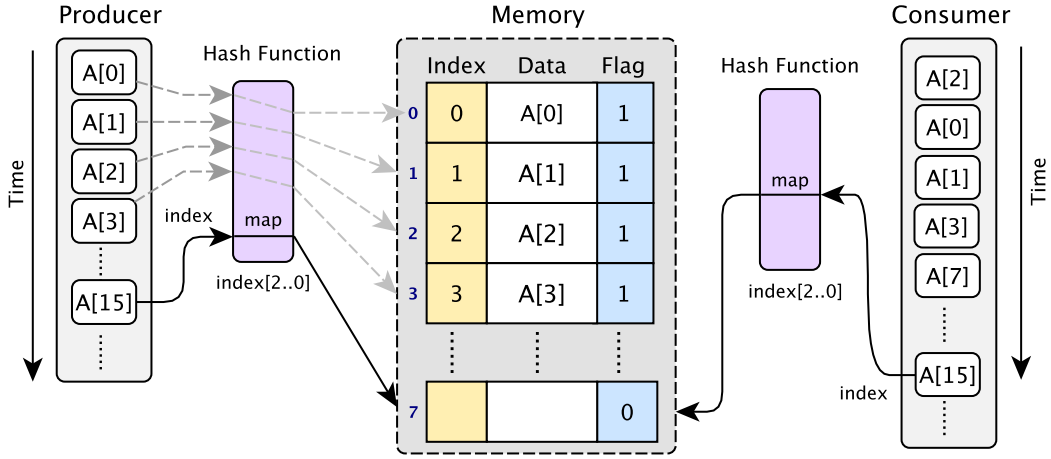


Figure 4.1: An example of hash-based indexing approach with a memory (size=8) and using the 3 least significant bits of index to access the memory.

deadlock situation arises as the mapping function (implemented in our case with hash functions) can create a circular dependency between the items the producer and consumer access.

We address this issue by enforcing the synchronization to be single-sided. We define the mapping function and the data-buffer size so that for the specific computation at hand, there is never a write-conflict at the producer, i.e., the producer will always be able to write data items to the data-buffer. This means that one needs to simultaneously dimension the size of the memory and the indexing function such that in any given window of time throughout the computation where data items need to be stored in the data-buffer, there are no two data items that map to the same entry in the table. The net result is a trade-off between a more sophisticated and more effective hashing or mapping function and the size of the memory as the data-buffer might be larger than a double-sided synchronization protocol would need.

Clearly, the determination of the size of the data buffer must take into account the order and rate at which the producer and consumer interact to understand when a given data buffer location becomes available for the producer to save data. To avoid this problem, we consider a memory hierarchy where at the first level there is a smaller table and at the second level we use the main memory and thus we always ensure that consumer neither stall waiting for storage availability nor data items

produced are stored in already occupied memory positions.

When accessing the data-buffer and the sync-table both pipeline stages translate the address or array index value into a table location using the hash function *Hash* as depicted in Figure 4.2. In order to avoid possible performance degradation, any hashing function of practical value needs to be simple to implement and, if possible, should not impose significant delays when accessing memory.

Although using an empty/full tagged memory scheme is an efficient solution for data synchronization between P/C pairs, it may increase the number of accesses to the main memory when the requested data is not available in local and/or in main memory. For instance, if the requested data from the consumer is not available in the local or in the main memory, the consumer waits until the data is available in the local memory or in the main memory (i.e., when the respective flag is set to one). Considering the overhead for flag checking in the local (on-chip) and in the main memory, the consumer may waste clock cycles and energy.

The *inter-stage Buffer (ISB)* structure with hashing is presented in Figure 4.2. This ISB has the following behavior: (a) a write operation stores the input data in the data buffer position determined by the hash function and flags in the same position the empty/full table (it corresponds to a simple store of 1 when an empty/full table is used); (b) a read operation loads the output data from the buffer position determined by the hash function and checks the flag stored at that position of the empty/full table; (c) once realized, the read operation also updates the corresponding item of the empty/full table (stores a 0 when an empty/full table is used).

As we mentioned previously, in many image/video processing computing stages, the P/C pair ratios are $(1 : N)$. In these cases, instead of a table with empty/full flags in the ISB, we use a table with numbers representing the number of times a data element is requested by the consumer. For each data produced and stored in the local memory of the ISB, the respective table value is set to the number of times it is requested (N). For each request of a data element stored in the local memory of the ISB, the respective table value is decremented by one and when it is 0 means it is empty. However, in some applications, the use of the maximum number of requests

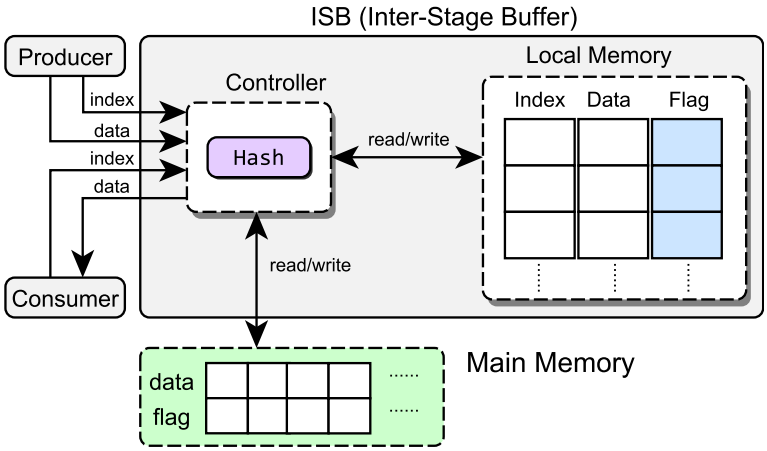


Figure 4.2: Inter-Stage Buffer using local and/or shared main memory with an empty/full bit flag.

per data element for all the elements implies local memory positions never freed. To prevent this situation, one can use a function as *Requests Calculation* in Figure 4.3 to calculate the precise number of requests (from 1 to N) for each requested element by the consumer.

As an example, consider an image processing application with the image size of 8×8 and 3×3 window block depicted in Figure 4.4 (left). The window block is moved horizontally to the next column until it reaches the last column of the image and then continues vertically considering the next row until it reaches the last row of the image. Figure 4.4 presents the situation when the window block moves to the next column of the image. In the second window block, most of the elements are requested in a previous window block which is depicted by the light blue color (see, Fig. 4.4 right). Note that the number of requests for each pixel of the image is not the same, i.e., based on the x, y of the pixel in the image, the number of requests can be different. For example, the four corners of the image are requested once, while the pixels in the center of the image are requested 8 times. Figure 4.5 presents the pseudo-code of the *Requests Calculation* function. The function uses the (x, y) of each produced element as an input argument and returns the number of requests for each element of the consumer stage.

Although this function sets the precise value for the synchronization table, it may not free local memory positions as intended when the number of requests is not

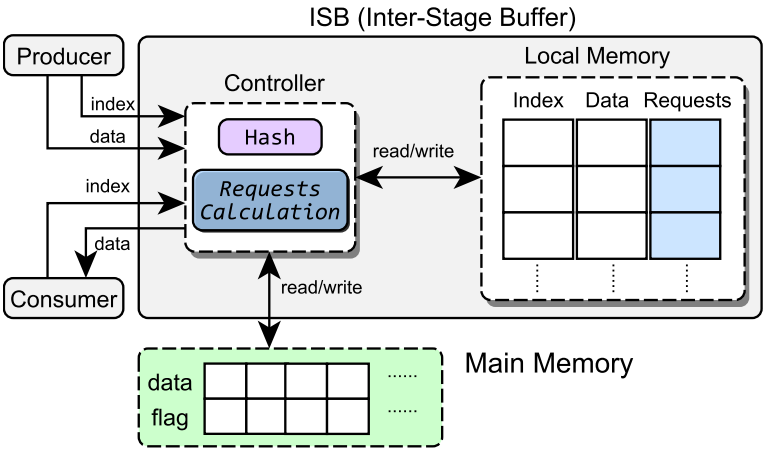


Figure 4.3: Inter-Stage Buffer using local and/or shared main memory and the requests calculation function.

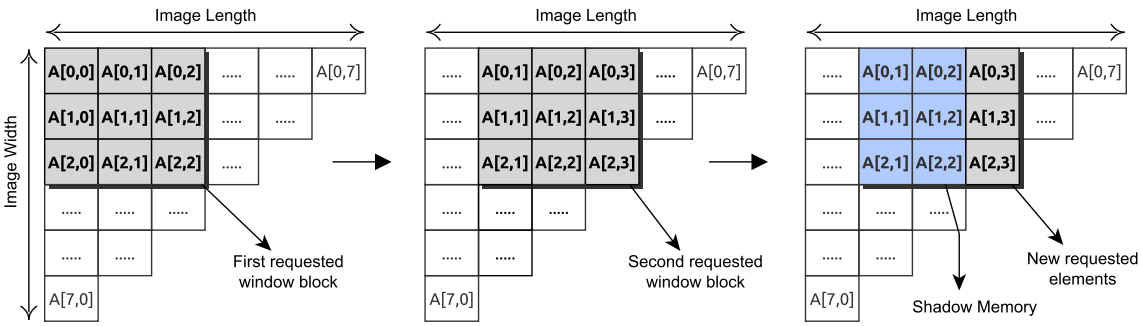


Figure 4.4: Example of a P/C pair with ratio of $(1 : C_{r_{max}})$: window block movement and reducing the number of accesses to the ISB by using a shadow memory.

constant for each pixel of the image. Thus, we propose an optimization technique to decrease the number of requests from the Consumer by storing the previously requested elements into a local *shadow memory*.

The shadow memory is implemented in the Consumer side as a simple one dimensional array to store the previously requested elements as shown in Fig. 4.4. By storing the previous window block in the shadow memory, the Consumer requests only the new elements which are not available in current shadow memory. As shown, the number of requests in the second window block can be decreased to only three requests. As a result of reducing the number of requests from the consumer to the ISB, we can achieve higher speed-ups for $1 : N$ ratio P/C pairs.

Here, we present some image processing applications with out-of-order computing stages and the ratio of $(1 : N)$ between P/C pairs such as FIR-Edge and Edge-

```
#define  I   Window_row
#define  J   Window_column
#define  L   Image_Length        // The columns of the image (length)
#define  W   Image_Width         // The rows of the image (width)

intCalculate_Number_of_Requests( int  x,  int y) {
   intm, n;
   if  (x < I)
       m = x;
   else if  ((x >= I) & (x <= (L - I)))
       m = I;
   else
       m = L - x + 1;
   if (y < J)
       n = y;
   else if  ((y >= J) & (y <= (W - J)))
       n = J;
   else
       n = W - y + 1;
   return(m * n);
   }
```

Figure 4.5: An example of *Request Calculation* function to compute the precise number of requests when the P/C pairs ratios are $(1 : N)$.

Detection kernels. For instance, the FIR-Edge kernel is based on a calculation of the window blocks which performs specific computations on each pixel of the image (e.g., average, sum). Based on the size of the window block and the coordinates of the pixel, the number of requests from the consumer is computed for each produced pixel using the *Requests Calculation* function in the ISB.

Figure 4.6 presents the original code of FIR-Edge, here with the identification of the producer (stage 1) and consumer (stage 2). As shown in this figure, the output of the first stage (herein *out*) is the input of the second stage.

By considering the data dependencies between the two stages, the original code of FIR-Edge can be partitioned and mapped into the producer and consumer sections and using an Inter-Stage Buffer (ISB) between the P/C pair as depicted in Figure 4.7. Note that the instructions *put* in the producer and *get* in the consumer side are defined as atomic instructions which provide the blocking read/write from/to the ISB.

In this example, the producer uses *put* instructions to send the index and data to the ISB. In a similar way, the consumer uses *put* instructions to request data elements from the ISB and uses *get* instructions to receive the requested data from

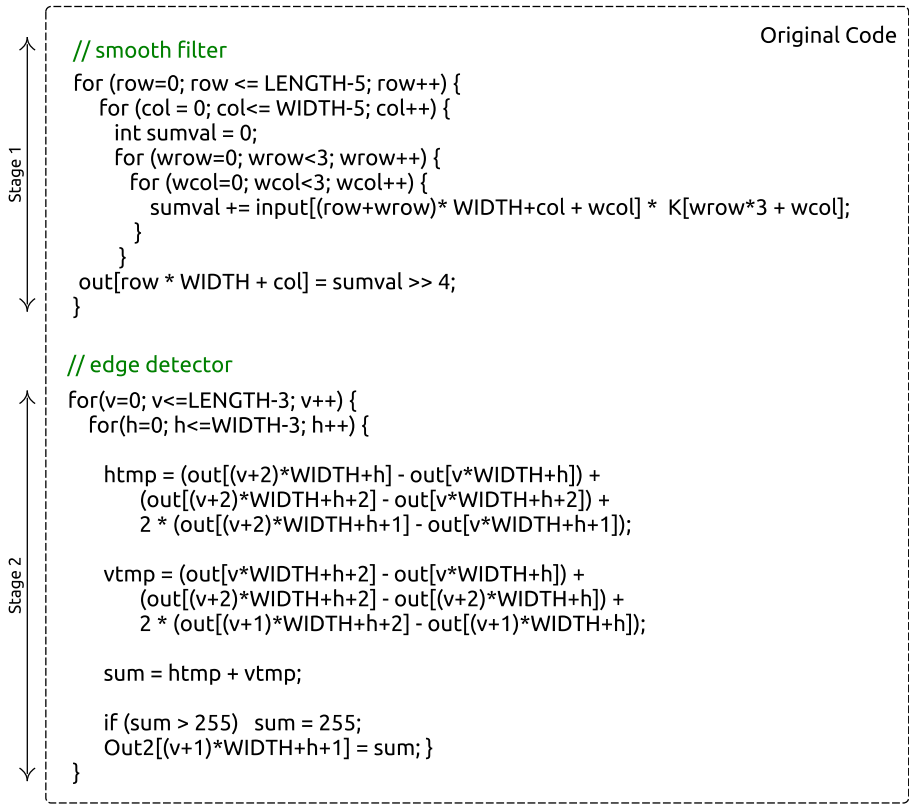


Figure 4.6: The original code of FIR-Edge with out-of-order data communication and (1 : N) ratio between the stages.

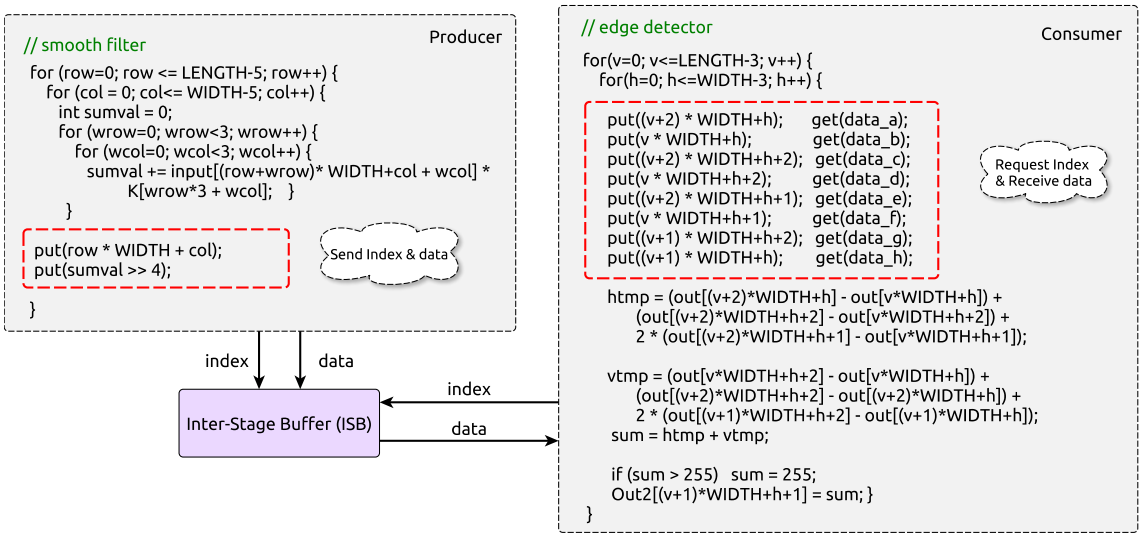


Figure 4.7: The partitioned code of FIR-Edge using an Inter-Stage Buffer (ISB) between the P/C pair.

the ISB. As the window block size in this example is defined as 3×3, the consumer requests 8 data elements in each iteration of the loop.

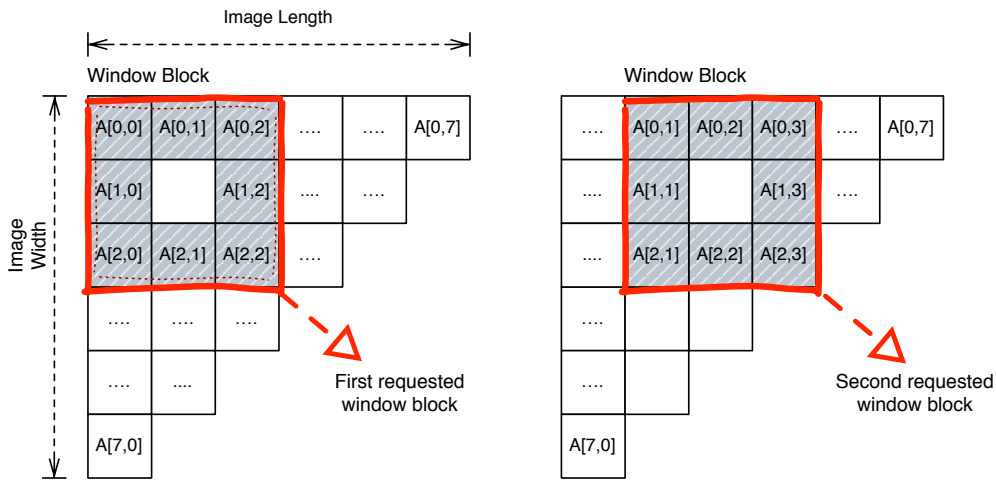


Figure 4.8: The Window block movements in FIR-Edge with out-of-order data communication and (1 : N) ratio.

Figure 4.8 depicts the first and the second iteration of the loop in the FIR-Edge example. As shown, by moving the requested window block in the second iteration of the consumer loop, the requests for four data elements (A[0,1], A[0,2], A[2,1], A[2,2]) are repeated. In addition, the Edge-Detection kernel is another example which the window block size is defined as 3×3. It means that in the consumer stage, all the elements of this window are required to perform a specific computation of the algorithm and stores the results in the output image.

4.1 Fine-grained Approaches

In the context of data communication and synchronization between cores, there are several approaches to overlap execution steps of computing stages (see, [ZSHD02, ZHD03]). In these approaches, functions or loops waiting for data may start computing as soon as the required data items are produced in a previous function or by a certain iteration of a previous loop. Decreasing the overall program execution time is achieved by mapping each stage to a distinct core (processor) and by overlapping the execution of computing stages. In order to apply TaLP, the applications is split into sequences of tasks (computing stages) that represent P/C pairs.

Fine-grained communication in the simple case of a sequence of two data-dependent computing stages (one as a producer and the other as a consumer), might be

achieved by using FIFOs to communicate data between the stages. A FIFO channel with blocking reads/writes is sufficient to synchronize data communications [TKD03b, ZHD03]. Note, however, that the use of FIFO channels is strictly dependent on the order of the communication pattern between P/C pairs.

In this section, we present different fine-grained data synchronization schemes for pipelining computing stages. The baseline architecture in this thesis is a single core with two data-dependent computing stages executing sequentially. The execution time of this scheme provides a criterion to compare the performance impact of different proposed fine- and coarse-grained data synchronization and communication approaches using TaLP.

4.1.1 Fine-grained ISB (Inter-Stage Buffer)

As briefly introduced in the beginning of this section, we explore an alternative inter-stage scheme to provide TaLP between P/C pairs and to overcome the limitations related to inter-stage communications based on FIFOs. In this scheme, for each data element being communicated between the producer and consumer, there is an empty/full flag. The empty/full tagged memories have been used in [Smi82], in the context of shared memory multi-threaded/multi-processor architectures. With our approach, we provide an extension to the empty/full tag memory model [Smi82] that considers a memory hierarchy approach. Figure 4.9 presents the block diagram of a fine-grained data synchronization scheme using an ISB between P/C pairs.

The producer is connected to the ISB using one channel responsible for communication between the producer and the ISB. The consumer is connected to the ISB by using two channels: *sending (requesting index)* and *receiving (reading data)* (identified by arrows between cores in Figure 4.9). Our current approach uses blocking write over the sending channel of the ISB and blocking read from the ISB over the receiving channel. The consumer gets data from the ISB using the receiving channel. The sending channel transmits the requests to the ISB concurrently. The producer and the consumer are both connected to the shared main memory. Note that in other architectures, one may have dedicated memories for producer/consumer and for the ISB. This would improve the overall performance of TaLP and

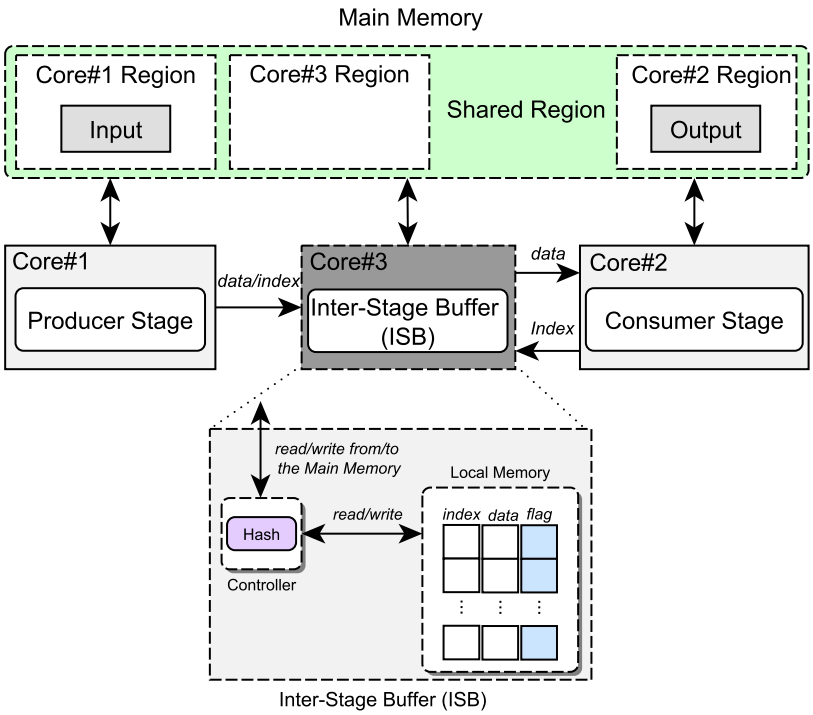


Figure 4.9: Fine-grained data synchronization scheme using an Inter-Stage Buffer (ISB) between P/C pairs and a shared main memory.

consequently there would not be memory access contentions. For instance, instead of the shared main memory, our approach can be implemented using distributed memory. Figure 4.10 presents the block diagram of a fine-grained data synchronization scheme using an ISB between P/C pairs and using distributed memory. Although using distributed memory may improve the overall performance of TaLP, it is a more expensive approach and may require additional programmer's efforts on data distribution and replication.

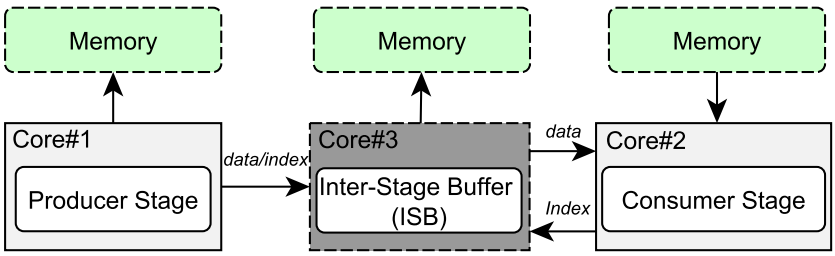


Figure 4.10: Fine-grained data synchronization scheme using an Inter-Stage Buffer (ISB) between P/C pairs and distributed memory.

In our approach, the ISB gets a requested index from the consumer side and checks the status of the respective flag addressed by the hash function if the index

matches. If the requested element is present (i.e., if the respective flag bit is full and the index matches) in the ISB local memory, it is sent to the consumer and the respective flag is set to empty. If the consumer requests an index which is not available in the local memory, the ISB checks if it is available in the main memory.

For each produced array element, the producer sends its index and value to the ISB (e.g., i as an index and $A[i]$ as a value). As shown in Figure 4.9, the ISB receives the index from producer side and maps the index into the local memory using the hash function (e.g., using a number of the least significant bits of the binary representation of the index). The index and value produced are then stored in the ISB local memory location defined by the address given by the hash function. Related to the value stored in the ISB, there is a flag that indicates if a data element was produced and thus can be consumed by the consumer.

Although reading/writing from/to local (on-chip) memory of the ISB is fast, the limitation of the size of local memory may prevent to store all produced data in out-of-order P/C pair cases. We may have a deadlock situation as the producer may stop to produce data if the ISB local memory is full or if ISB local memory location addresses are occupied. To avoid deadlock situations, one can determine before system deployment the minimum size of the local memory needed. Such approach was proposed in [RCD07] in the context of TaLP for application-specific architectures, where the buffer size was determined using register-transfer level (RTL) cycle accurate simulation. Thus, to circumvent this problem, we provide ISB access to the main memory and data is stored in the main memory whenever the flag bit in the local memory is full. In this case, the ISB stores the data value in the main memory without using the hash function. If both flag bits of the local and main memory are empty, the consumer waits until the requested index produces and stores the requested data in local or in main memory.

In summary, the ISB scheme provides TaLP and data communication between P/C pairs for both in-order and out-of-order computing stages. Also, the ISB scheme overcomes the limitations of inter-stage communications based on FIFOs.

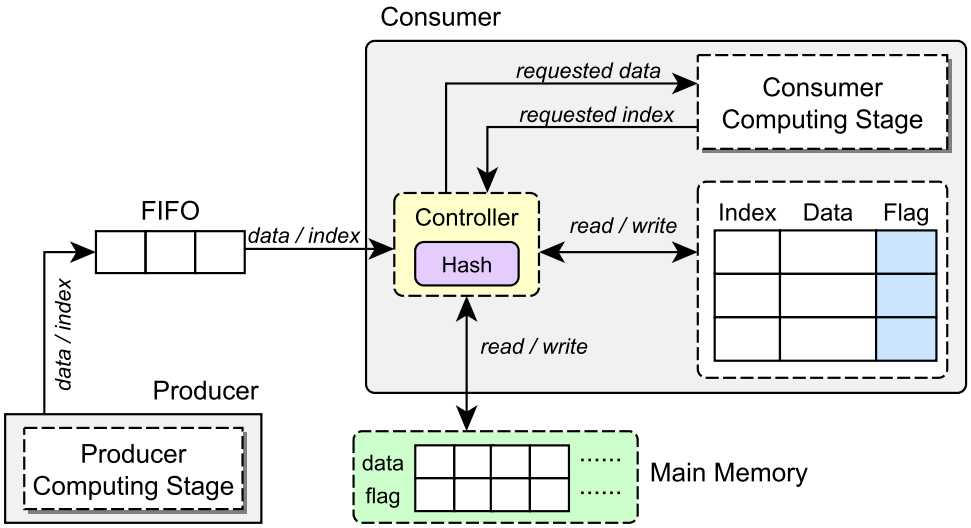


Figure 4.11: Fine-grained data synchronization scheme using a FIFO between P/C pairs and considering the inter-stage buffer (ISB) in the consumer.

4.1.2 Fine-grained ISB within Consumer

Figure 4.11 shows a fine-grained data synchronization scheme which uses a FIFO between P/C pairs and includes an ISB in the consumer. In this scheme, the producer sends the produced indexes and data elements through the FIFO. The controller reads the FIFO and checks if the current read index is equal to the requested index of the consumer. If the indexes are equal, the controller reads data from the FIFO and sends it to the consumer directly. If the indexes are not the same, the controller maps the current read index into the local (on-chip) memory of the consumer ISB. The local memory structure is based on the empty/full flag bit synchronization model previously described. If the controller cannot store the index in the local memory, the controller disables reading from the FIFO. In a similar way, if the requested index from the consumer is not the same as the read index from the FIFO and the consumer is unable to load the requested index from the local memory, the controller turns off reading the next requested index from the consumer until the previous requested index is available in the local or in the main memory.

4.2 Coarse-grained Approaches

We present two different types of coarse-grained data synchronization approaches, using FIFOs between P/C pairs and a shared main memory to synchronize computing stages. In the context of coarse-grained data synchronization approaches, chunks of elements or an entire array of elements (e.g., an image) are considered instead of each data element. To consider frequent communication of data between P/C pairs in these systems, we assume that the producer and the consumer computing stages process N data chunks.

The coarse-grained approaches addressed here consider both data communication and synchronization of entire arrays and in-order P/C pairs at this granularity level. In the case of out-of-order at this granularity level, the approaches presented in the next sections would need that the consumer stores the identifiers of the data chunks provided from the FIFO between P/C pairs and not currently requested. This approach also considers that the data chunks being communicated are too large to be stored in a local buffer (such the one in the ISB) and this is the reason why only a flag signaling a new data chunk is directly communicated through the FIFO between the producer and the consumer. For data chunks able to be communicated via on-chip buffers, the previous ISB approach can be extended to deal with data chunks instead of data elements. In that case, there would be a synchronization flag per data chunk. In the presence of in-order communication patterns, one may opt to the FIFO based communication instead of using an ISB.

4.2.1 Coarse-grained One FIFO

Figure 4.12 presents the block diagram of a coarse-grained data synchronization architecture using a single FIFO as a communication component between P/C pairs. In this scheme, the FIFO contains the *id* of producing data chunks (e.g., an image). The producer stores the produced data chunks in a shared main memory and puts their *ids* (e.g., base address of an image in the main memory) to the FIFO. The consumer gets the *id* from the FIFO and reads the array elements directly from the main memory. Reading/Writing from/to the FIFO is blocking. It means that if the

FIFO is full, the producer stops producing. Similarly, if the FIFO is empty, the consumer waits until the producer puts an *id* into the FIFO. In this scheme, the

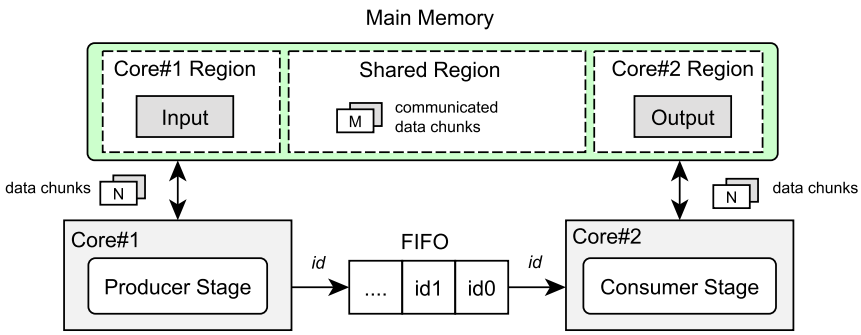


Figure 4.12: Coarse-grained data synchronization block diagram using a single FIFO.

number of communicated temporary data chunks (herein referred as M) stored in main memory is an important key. If $M = 1$, it means that the producer waits for the consumer to consume the entire previously generated data chunk before generating another data chunk. As soon as the *id* of the data chunk is available, the consumer can read the array from main memory and the producer can store the next data chunk in the main memory. Thus, when $M = 1$, the producer and the consumer run sequentially and the execution time is the same as with a single core. Therefore, TaLP is achieved when the minimum number of temporary data chunks is $M > 1$.

4.2.2 Coarse-grained Two FIFOs

Figure 4.13 presents the block diagram of a coarse-grained data synchronization architecture using two FIFO channels between P/C pairs. In this scheme, FIFO 2 stores the *id* of produced data chunks. Similarly, FIFO 1 stores the *id* of consumed data chunks. When the consumer puts the consumed data chunk's *id* into FIFO 1, the producer can reuse the memory by storing the new produced data chunk in the location associated to the *id* received from FIFO 1.

In this scheme, the number of temporary stored data chunks (M) in external memory is less or equal to the number of data chunks being computed (N), while in the previous coarse-grained scheme, the number of temporary stored data chunks in external memory is equal to the number of data chunks being produced/consumed.

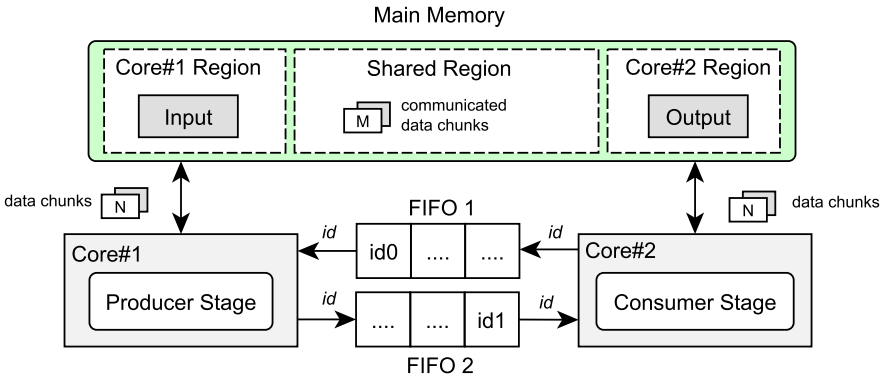


Figure 4.13: An architecture for coarse-grained data synchronization using two FIFOs.

Therefore, in this scheme, the producer can store the new data chunk in external memory as soon as there is space. In a similar way, the consumer reads the id from FIFO 2, consumes the data chunk and sends the *id* to the producer using FIFO 1. Note that depending on the number of data chunks being communicated between stages and the size of the main memory for accommodating data chunks, FIFO 1 may not be needed.

4.3 The TaLP Design Flow

This section describes our proposed design-flow for applying TaLP on FPGA-based multicore architectures (specially targeting the architectures proposed in this thesis). We describe the flow from a sequential program to the pipelining of the execution of computing stages using TaLP. The TaLP design flow requires the steps illustrated in the design-flow block diagram depicted in Figure 4.14. The steps for the TaLP design-flow are the following:

- Identifying the computing stages which includes:
  - Identifying the dependencies between stages;
  - Determining the communication pattern and ratio;
- Measuring the execution time of the stages when considering a single core architecture;
- Deciding the granularity and TaLP scheme;
- Mapping and scheduling computing stages;

- TaLP performance impact evaluation and estimating the speedup;
- Applying TaLP and measuring the real speedup.

In the following subsections, we describe in detail each of the design flow steps.

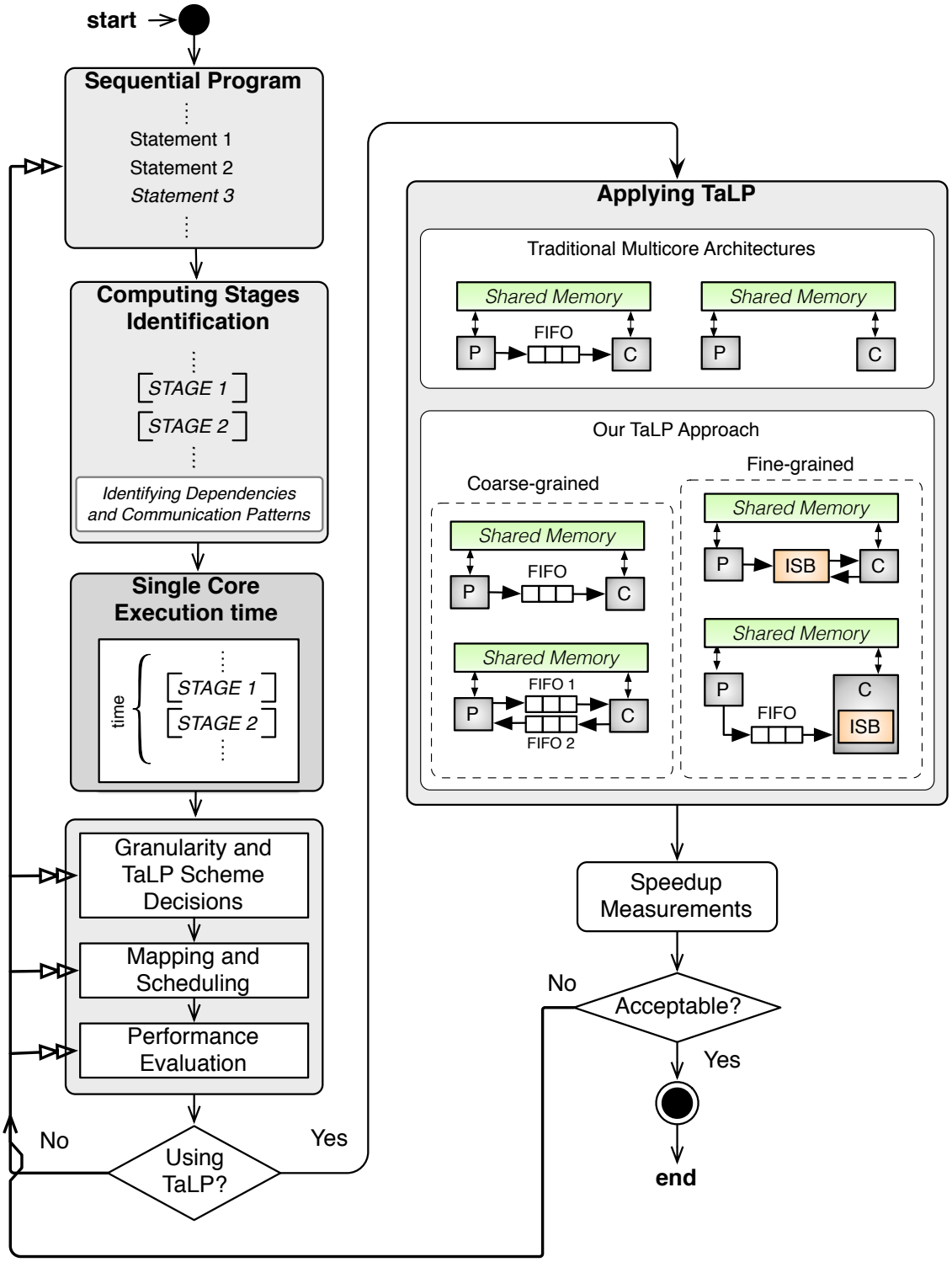


Figure 4.14: Full view of the TaLP design flow.

4.3.1 Computing Stage Identification

The input of our approach is a sequential program written in standard C programming language (see, Figure 4.14). Identifying the computing stages of a sequential program is the beginning and the most important step to provide TaLP. As we mentioned in Section 3.1, in order to identify the computing stages, the hotspot section of the program needs to be chosen. The computing stages can be identified by a user (possibly with the help of tools) or by specific tools.

In manual analysis, first we need to identify the loops, nested loops and function calls inside the sequential program that consume a significant amount of time. One of the metrics that can help to determine the complexity of the loops might be the number of iterations and/or the complexity of the inner loop computations. The second step is to determine if the hotspots of the program can be partitioned and map into the producer-consumer model.

The manual analysis can be suitable when the number of stages is small, and the computing stage operations are simple and easy to identify. In some applications, the number of computing stages in the sequential program is high and thus identifying the computing stages can be difficult. In such cases, we suggest to use the help of specific tools.

4.3.2 Identifying the Dependencies

The second step after determining the computing stages in the sequential program is to determine the dependencies between the computing stages and also whether the dependencies between the computing stages can be handled.

We described the producer and consumer stages and different types of computing stages dependencies in Chapter 3. As an example, we can consider the example of the FIR-Edge which it was previously presented in Figure 4.7. This example is naturally consists of two dependent computing stages (Stage 1 and Stage 2), one as a producer stage and another one as a consumer stage. Figure 4.15 presents the original code of the FIR-Edge with the dependencies between computing stages. The first stage (producer) is responsible to smooth the input image and output the

result image (i.e., out array). The second stage (consumer) also reads the smoothed image from the previous stage as an input, calculates the edge of the smoothed image and output the result (i.e., Out2 array).

Stage 1

```
// smooth filter
for (row=0; row <= LENGTH-5; row++) {
    for (col = 0; col<= WIDTH-5; col++) {
        int sumval = 0;

        for (wrow=0; wrow<3; wrow++) {
            for (wcol=0; wcol<3; wcol++) {
                sumval += input[(row+wrow)* WIDTH+col + wcol] *
                          K[wrow*3 + wcol];
            }
        }
    out[row * WIDTH + col] = sumval >> 4;
    }
}
```

Stage 2

```
// edge detector
for( v=0; v<=LENGTH-3; v++) {
    for( h=0; h<=WIDTH-3; h++) {

        htmp = (out[(v+2)*WIDTH+h] - out[v*WIDTH+h]) +
               (out[(v+2)*WIDTH+h+2] - out[v*WIDTH+h+2]) +
               2 * (out[(v+2)*WIDTH+h+1] - out[v*WIDTH+h+1]);

        vtmp = (out[v*WIDTH+h+2] - out[v*WIDTH+h]) +
               (out[(v+2)*WIDTH+h+2] - out[(v+2)*WIDTH+h]) +
               2 * (out[(v+1)*WIDTH+h+2] - out[(v+1)*WIDTH+h]);

        sum = htmp + vtmp;
        if (sum>255) sum = 255;
           Out2[(v+1)*WIDTH+h+1] = sum;
    }
}
```

Figure 4.15: Identifying the dependencies in the original code of FIR-Edge.

4.3.3 Determining the Communication Patterns and Ratios

One of the essential steps in the computing stage identification is to determine the communication pattern of the producer and the consumer (in-order or out-of-order). Also, we need to determine the ratio of the produced and consumed data elements between the P/C pairs. In this step, we evaluate the data communication patterns and ratios between the P/C pairs to determine the proper data communication and synchronization scheme to provide TaLP.

In practice, the approach is to execute the sequential program on a local machine and analyze the produced and consumed indexes. By analyzing the indexes, we can

determine the data communication pattern and the ratio of the P/C pairs. For instance, for the P/C pairs with the ratio of $(1 : N)$, we may need to use the ISB scheme with the *Requests Calculation* function to determine the precise number of requests from the consumer as presented in Figure 4.3.

4.3.4 Granularity and TaLP Scheme Decision

Identifying the granularity of the data communication and synchronization between P/C pair is an important key which it can help to choose suitable TaLP scheme.

In order to determine the granularity (fine-grained/coarse-grained), we consider the data dependencies, data communication patterns, and ratios between P/C pairs. For instance, if instead of each data element communicating between P/C pairs, chunks of elements or an entire array of elements (e.g., an image) is considered, the granularity of P/C pair is coarse-grained and thus, we choose coarse-grained TaLP scheme. In contrast, if each data element communicates between P/C pairs, the granularity of P/C pair is fine-grained.

We previously described the use of traditional architectures (using FIFOs or shared flag-based main memory between P/C pairs). If we use our fine-grained ISB scheme, depending on the ratio of the P/C pairs, we need to choose the proper ISB as an inter-stage buffer between P/C pairs. For instance, when the ratio of the P/C pair is $(1 : N)$, the ISB may need to use the function *Request Calculation* to determine the precise number of requests of each data element in the consumer and sets the flag into the local memory of the ISB.

4.3.5 Mapping and Scheduling Computing Stages

This step is responsible for mapping computing stages into the producer and the consumer cores. To apply TaLP using a multicore architecture, we partition the original C code into separated codes to map them into separated cores.

The second step is to modify the producer and the consumer code to execute in the target multicore architectures. In our approach, we use two atomic instructions (*get* and *put*) to provide blocking writes and blocking reads to/from the cores. However, the use of these instructions might be different depending on the target

architecture used. For instance, when using FIFO-based synchronization schemes, the *put* primitive can be used to store data into the FIFO, and the *get* primitive to load data from the FIFO. When using the ISB-based synchronization schemes, the *put* instruction can be used when the producer sends data to the ISB. In a similar way, the consumer requests data using the *put* instruction and receives data using the *get* instruction.

Although mapping two computing stages into the producer and consumer cores might be an easy task, mapping and scheduling the applications with more than two computing stages can be a challenge. The pipelining of applications consisting of more than two computing stages may require an architecture with more cores than the ones previously used. One of the possible solutions is to have multiple cores, which can behave as a producer; consumer; or producer-consumer cores. In one extreme of the design space, we may have one core per stage, and thus, each computing stage is mapped to a distinct core. However, this solution may not be a feasible when the application has many computing stages and/or the number of stages is higher than number of cores possible to implement in the FPGA used.

Another possibility is to use only two cores and two ISBs to implement different types of the computing stages. In this case, in order to provide TaLP, we need to partition the stages properly between two cores. The way we split computations in stages can have an impact on the communication structure as we may need to have more than one ISB or an ISB with multiple local tables (e.g., for dealing with more than one array variable being communicated).

Consider the example of a sequential program with dependent computing stages which previously presented in Figure 3.4. In this example, the *Stage 2* and *Stage 3* are dependent to the *Stage 1* while the *Stage 2* and *Stage 3* are independent to each other and thus they can be executed concurrently. The *Stage 4* also is only dependent to the *Stage 3*.

Figure 4.16 shows the block diagram of the suggested solution for multi-computing stages. One of the possibilities to map computing stages to cores can be to map the Stage 1 and the Stage 4 into the first core and map the Stage 2, 3 and the Stage 5 into the second core. Note that the way we split computations in stages can have

an impact on the communication structure as we may need to have more than one ISB or an ISB with multiple local tables (e.g., for dealing with more than one array variable being communicated). Therefore, instead of one ISB with only one local memory, we can use two ISBs with two local memories to provide the communication and synchronization between the stages. The ISB can also be implemented as an IP (Intellectual Property) core. In the Appendix of this thesis, we implement the ISB in hardware.

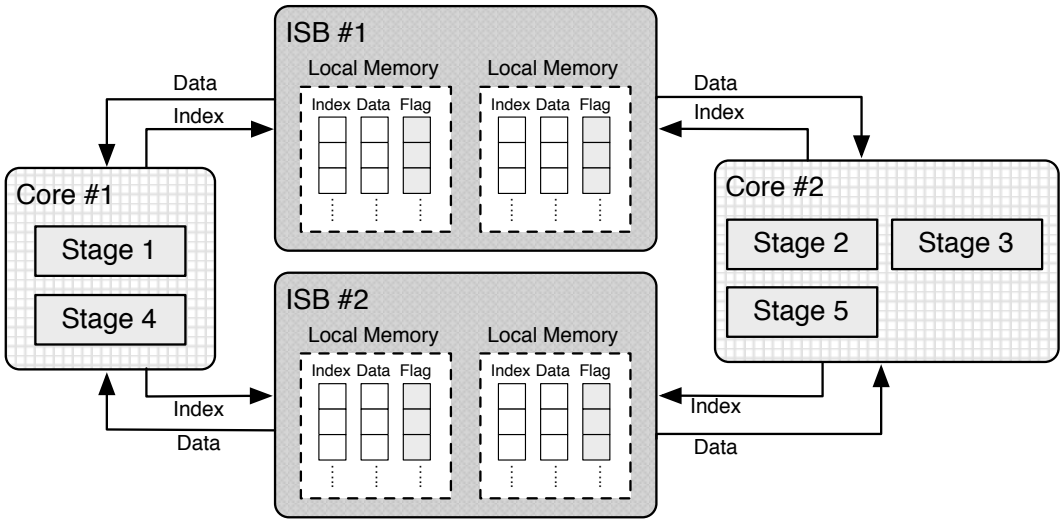


Figure 4.16: A Block diagram of a possible solution to provide TaLP for sequential programs with more than two computing stages.

Although our approach supports multi-input/multi-output pipelining and sequence of P/C pairs, we consider only one P/C pair for simplicity and the ease of implementation in hardware. The impact of an architecture such as the one presented in Figure 4.16 on performance for the applications with more than two computing stages is considered as future work.

4.3.6 TaLP Performance Impact Evaluation

This step is responsible for estimating or measuring the execution time of the sequential program and analyzing the impact of TaLP on the overall performance of the application giving an identification of computing stages. One possibility is to use theoretical models and upperbounds in order to analyze if the system will possibly profit by using TaLP with the partitioning being considered.

In our case, we use the highly optimistic theoretical speedup bounds (herein: Upperbound) for each application as calculated with Equation 4.1 for two stages. This upperbound reflects the distribution of the execution time over the two tasks (computing stages). For the application with two computing stages, the maximum possible value for this upperbound is 2 and would correspond to the execution time equally split over the two tasks (well balanced) and an optimistic fully overlapping of the execution of the tasks. We note that however, the promotion of the data communication between stages to on-chip hardware structures may have an additional effect.

$$Theoretical\ Speedup\ bound = \frac{(T_{S_1} + T_{S_2})}{Max(T_{S_1}, T_{S_2})} \tag{4.1}$$

Where T_{S_i} $(i = 1,2)$ represents the execution time of stage S_i. In general, for benchmarks with more than two computing stages and TaLP support for only two stages (one P/C pair), we compute the upperbound speedup by considering Equation 4.2, where T_{S_k}, T_{S_j} represent the execution time of the two pipelining stages and n is the number of computing stages.

$$Theoretical\ Speedup\ bound = \frac{\sum_{i=1}^{n} T_{S_i}}{\sum_{i=1..n \notin \{k,j\}} T_{S_i} + Max(T_{S_k}, T_{S_j})} \tag{4.2}$$

For instance, for the Wavelet Transform with four stages ($S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$) used, we schedule the execution of tasks as $S_1 \rightarrow S_2$ in one core and $S_3 \rightarrow S_4$ in another core. Therefore, only the execution of S_2 and S_3 is partially overlapped. In this case, the theoretical speedup upperbound is given by the equation below:

$$(T_{S_1} + T_{S_2} + T_{S_3} + T_{S_4}) \ / \ (\ (T_{S_1} + T_{S_4}) + Max(T_{S_2}, T_{S_3})\ )$$

Note that if the execution time of the producer stage is approximately equal to the execution time of the consumer, we can consider it as a well-balanced P/C pair.

Figure 4.17 shows an example of a sequential program which consists of two stages (Stage 1 and Stage 2) and the data dependency between the stages. As shown, Stage 1 is the producer for the second stage and the execution time of the

first stage is less than the execution time of the second stage. This means that the execution time of the stages is not balanced and the idle time of the producer is considerable. Thus, if the idle time of the producer or consumer side is considerable, the performance impact by applying TaLP might not be noticeable.

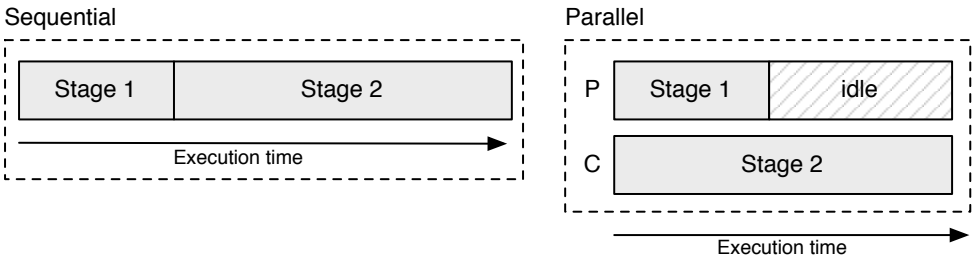


Figure 4.17: An example of a sequential program with two stages and the unbalancing of the execution time of the stages.

Note that if this step indicates that the use of TaLP does not suits the objectives, we can go back in the design flow and repeat the previous steps (i.e., such as mapping and scheduling of computing stages).

4.3.7 Applying TaLP and Measuring the Speedup

This step is responsible for applying TaLP and measuring the speedup of our approach. In this step we measure the real overall execution time of the selected TaLP approach in previous steps and evaluate the performance impact of the approach by determining the speedup. If the achieved speedup is not considerable, it might not be sufficient to go back in the design flow and we may need to apply code transformation techniques.

There are however optimization techniques for the ISB that shall be always considered. Some of these techniques are proposed in next chapter.

4.4 Summary

In this chapter, we presented our approaches based on fine- and coarse-grained data synchronization schemes to provide TaLP between the stages. Our fine-grained approach is a flag-based synchronization scheme using hash-indexing and empty/full flag memory to provide parallel synchronization and communication between cores.

It relies on an ISB (inter-stage buffer) data synchronization scheme which provides TaLP between P/C pairs and, as a result, overcomes the limitations related to inter-stage communications based on FIFOs.

An important aspect of our approach is the use of customized inter-stage buffer schemes to communicate data and to synchronize the cores associated with the producer-consumer tasks (computing stages). Our approach provides the ability to pipeline the tasks of in-order and out-of-order communication patterns between P/C pairs without reordering the sequence of producing and/or consuming data. Also, our approach supports different ratios of P/C pairs. For P/C pairs with the ratio of $1 : N$ (the most common ratio in the domain of image/video processing applications), we presented two optimization techniques (*Requests Calculation* function and *Shadow memory* technique) for the ISB to compute the precise number of requests from the consumer and thus to reduce the number of repeated requests from the ISB and reuse the previous requests.

We also presented two coarse-grained data synchronization schemes when the produced and consumed data are data chunks (e.g., an image). In these schemes, we use the traditional FIFO-based synchronization including the main memory as a data communication channel between P/C pairs. Finally, we presented the top-down design flow of our approach for providing TaLP for a given sequential application. We described and introduced the techniques to identify the stages of a sequential program and evaluate the impact of TaLP on performance.

CHAPTER 5

Optimization Techniques

Chapter Outline

In the previous chapter we presented different fine- and coarse-grained synchronization schemes to provide Task-Level Pipelining (TaLP). Reducing the number of accesses to the external (main) memory is an important technique to achieve higher performance in these schemes. In this chapter, we present optimization techniques to reduce the number of memory accesses to the different levels of the memory hierarchy considered in the communication between producers and consumers. The optimization techniques are classified into two groups: optimizations for the FIFO-based schemes and optimizations for the ISB schemes.

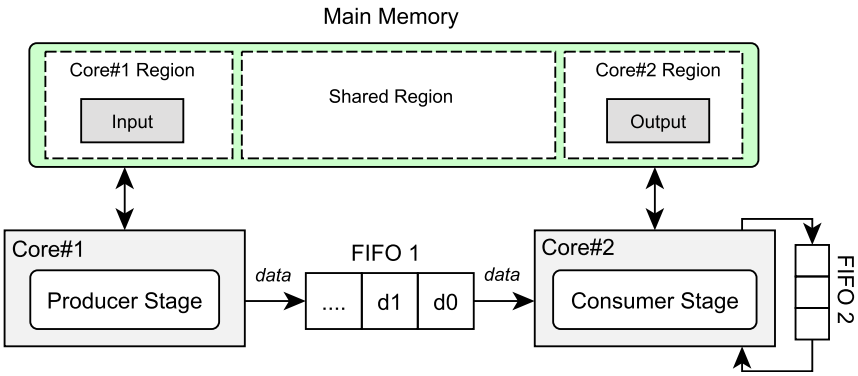


Figure 5.1: Fine-grained data synchronization scheme using a FIFO between P/C pairs and an extra FIFO connected to the consumer to provide a buffer and sequences of accesses to main memory.

5.1 Optimization for Shared Memory Schemes

In the scheme using a standard FIFO between producer/consumer (P/C) pairs (see Figure 3.9), the producer loads the input data from the external memory and puts the produced data into the FIFO. The consumer gets data from the FIFO, process the data and then store the output into the external memory. The consumer writes the output using the specific rate associated with the computations needed before outputting another data element.

The main idea of the optimization scheme presented here is to reduce memory contention by buffering locally the outputs of the consumer stage and then storing them in data chunks (signaling the producer when storing). Figure 5.1 shows our fine-grained data synchronization scheme to provide temporary buffering and blocks of store to external memory in the consumer side. We added an extra FIFO channel (FIFO 2) to the consumer. The consumer reads data from FIFO 1 (producer side), computes, and writes results into FIFO 2 while FIFO 2 is not full or does not have the number of data elements considered for each data chunk. Then the consumer writes all the data in FIFO 2 in external memory. When FIFO 2 gets empty, the consumer restarts reading data from the producer (FIFO 1). This scheme provides the capability to block external memory accesses from the different cores and to avoid possible delays managing simultaneous accesses to the external memory.

Note that this optimization scheme is also suitable for the ISB scheme as it avoids simultaneous accesses to the shared memory.

5.2 Optimizations for ISB-based Schemes

In this section, we present our optimization techniques for the ISB scheme. The goal of these optimization techniques is to maximize the usage of local (on-chip) memory and to reduce the number of accesses to the external memory. These techniques can be categorized into two main groups: the optimizations regarding the hash functions used in the ISB and the techniques to reduce the memory accesses achieved by different hash functions and by other specific optimizations at the ISB.

5.2.1 Hash Functions

Considering the overhead of loads/stores from/to the external memory by the ISB, the performance of the ISB schemes may improve if the data requested from the consumer are as much as possible available in the local (on-chip) memory rather than in the external memory. The hash function is one important component affect if each data element is stored locally or if it needs to be stored in the external memory. Therefore, a hash function able to map the data elements into the local memory with minimum collisions is one possible optimization to maximize the use of local (on-chip) memory.

To evaluate the impact of different hash functions on the use of local (on-chip) memory, we use the 12 different general hash functions presented in Table 5.1. There are two main implementation methodologies for hash algorithms: *Additive/Multiple hashing* and *Rotating hashing*. In Additive/Multiple hashing, the hash value is provided by continually incrementing an initial value by a calculated value corresponding to an element in the data. In this form of hash implementation, the calculation of the element value is a multiplication by a prime number. However, in a rotating hashing implementation, to determine the hash value, we process bitwise shifting operations (left or right or a combination of both) on the value. Note that the shifting amounts are prime numbers as in Additive/Multiple hashing.

Table 5.1: General purpose hash function algorithms.

Hash Functions	Hash Description
H1 (Mod)	Modular mode
H2 (RS)	A simple hash function from Robert Sedgwicks [SW11]
H3 (JS)	A bitwise hash function written by Justin Sobel
H4 (PJW)	Based on work by P. J. Weinberger of AT&T Bell Labs.
H5 (FLV)	Fowler–Noll–Vo hash function [Nol04]
H6 (BKDR)	From B. Kernighan and D. Ritchie's book [Ker88].
H7 (SDBM)	Used in the open source SDBM project.
H8 (DJB)	An algorithm proposed by Daniel J. Bernstein
H9 (DEK)	An algorithm proposed by Donald E. Knuth in [Knu14]
H10 (AP)	AP hash function provided by Arash Partow in [CLRS09]
H11 (Open Addr)	Open Addressing [PH05]
H12 (Comp)	Complement Modular Addressing

Note that the hash functions presented in Table 5.1 are known as simple hash functions and can be representative for a mix of Additive/Multiple and Rotating general purpose hash functions. The source code of the hash functions is available in [Par14]. The hash functions used, need to be simple and not requiring high computing demands as their calculation will be in the critical path to store and write data at the ISB. However, custom hardware cores to implement the hash function can be used if needed.

5.2.2 Main Memory Accesses: Scheme #1

In the previous ISB scheme, i.e., without using optimization techniques, for each miss to the local memory, the ISB keeps checking the flag of the requested index in the external memory. This means the ISB needs to access to the external memory even for the indexes requested not yet available in the external memory. Therefore, we propose a solution to predict if the requested index is available in the external memory.

To reduce the number of accesses to the external memory by the ISB, we present an optimization scheme which consists of a local variable (herein: variable v, initialized to 0) to approximately track the possible data previously produced and stored in the external memory. For each write into the external memory, the ISB calcu-

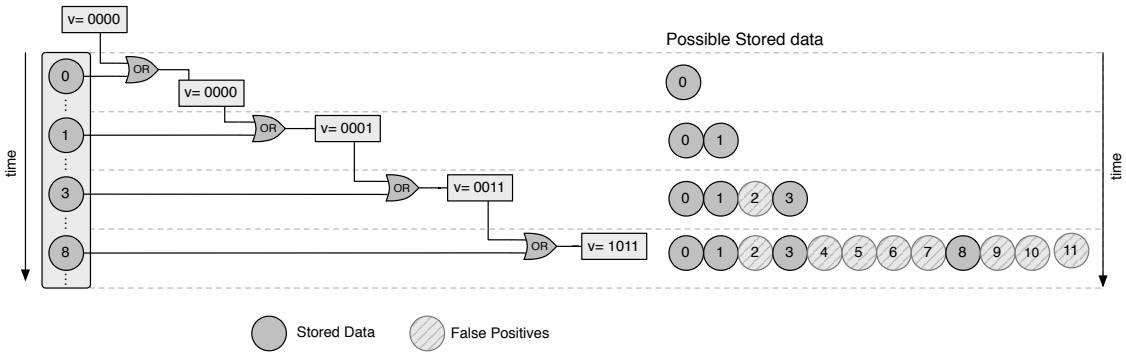


Figure 5.2: An example of optimization scheme #1 to approximately represent the presence in external memory of produced data.

lates the bitwise *OR* of the stored data in the external memory with the current v value. The value of v represents the possible indexes whose data might be available in the external memory. For each write to the external memory, the ISB updates the value of v (i.e., $v = v\ OR\ index$). For instance, as shown in Figure 5.2, if data correspondent to index 3 was stored in the external memory, the current value of v would be 3. This value represents that the data for the indexes $0, 1, 2$ and 3 are possibly available in the external memory.

Before each read from the external memory, the ISB checks the value of v to acquire if the data for the requested index is available in the external memory. The ISB calculates the bitwise *AND* of the requested index from the consumer with the current value of v. If the result is equal to the requested index from the consumer (see Figure 5.3), it means that the data element associated to the requested index might be available in the external memory. Then, the ISB checks the flag of the requested index in the external memory. If the flag is set to one, the ISB reads the index and resets the flag to *empty* (zero). If the flag is *empty*, it means that a false positive has occurred and the consumer needs to wait until the data element is available in the local or in the external memory (see Figure 5.3).

Although in this optimization technique, using a single variable and an OR-ing optimization scheme may not provide an accurate solution to reduce the number of accesses to the external memory (false positives), it reduces many unnecessary accesses for most applications with out-of-order communication patterns.

```
// at this point the ISB has determined that the data element
   related to consumer_index is not stored in the ISB local
   memory and thus it needs to check the external memory
// (1 : 1)
if (consumer_index AND v) == consumer_index then
    if flag [consumer_index] == 1 then
        read from the external memory
    else
        Not in the external memory ; // A false positive
else
    Not in the external memory ; // Correct signaling
```

Figure 5.3: An optimization scheme #1 for loads from the external memory.

5.2.3 Main Memory Accesses: Scheme #2

By considering only one variable in the previous optimization scheme, we may have many false positives as the variable may represent many indexes whose values are not available in the external memory. For example, in Figure 5.2, $v = 1011$ represents 8 values and only 4 (0,1,3,8) values are really stored.

To reduce the number of false positives, we include a new variable (p). p is a vector of n integers (being n the number of bits needed to represent the indexes) where each position represents the number of indexes with one on the associated bit position whose data are currently stored in the external memory (see Figure 5.4). Therefore, in this optimization scheme, the ISB updates two variables v and p for each write/read to/from the external memory. v is used as an auxiliary variable to make sure operations more efficient than applying them to an array of integers such p.

Figure 5.5 shows an example of writes/reads to/from the external memory. As shown in this example, when a value for index 3 is stored into the external memory, v and p become both 0011. As shown in Figure 5.5 (Store), if the ISB stores another value (e.g., with index 7) into the external memory, the value of p is calculated by adding each bit of the index to the associated position of p one by one (e.g., $v = 0111$ and $p = 0122$). For each read from the external memory, the ISB updates the values of v and p. As shown in Figure 5.5 (Load) and in Figure 5.4, if the

```
// Updating the values of v and p for each read from the
   external memory (bits=number of bits for the index).
if (consumer_index AND v) == consumer_index then
    if flag [consumer_index] == 1 then
        read from the external memory
        mask = 1;
        for (i=0; i<bits; i++) do
            j = consumer_index&1;
            consumer_index = consumer_index ≫ 1;
            if (j==1 AND p[i] >0 ) then p[i]= p[i]-1 ; // Update p
            ;
            if (p[i]==0) then
                v = v AND !mask ; // Update v
            mask = mask ≪ 1;
    else
        Not in the external memory
else
    Not in the external memory
```

Figure 5.4: An optimization scheme #2 for loads from the external memory.

consumer requests an index (e.g., 3), the ISB subtracts the bits of this index from the current value of p (0122) and p becomes 0111. In this example, immediately after loading 7, v becomes 0000 which means that there is no data available in the external memory. However, if we have used the scheme #1, v stay equal to 0111, identifying eight false positives. This optimization scheme contributes to a reduction of possible simultaneous loads/stores and to a reduction of accesses to external memory which may decrease both execution time, power dissipation and energy consumption. We note that variable v does not have more information than variable p, but it is important to make the operation verify if a given index has data in external memory.

5.2.4 Main Memory Accesses: Scheme #3

In previous schemes, the ISB receives index/data from the producer and stores them into the local or external memories. Also, the ISB receives the requested indexes from the consumer, reads the local and/or main memories and communicates the data to the consumer. For each data requested by the consumer and neither available

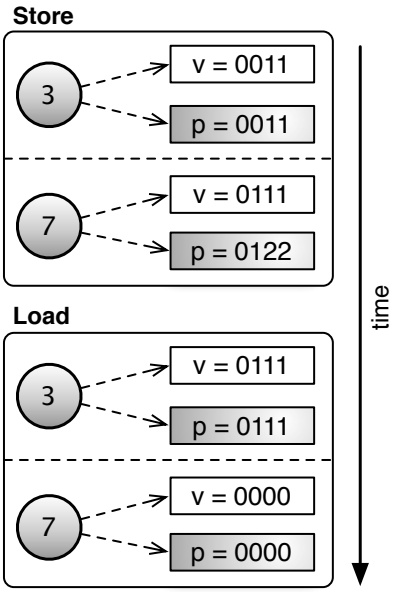


Figure 5.5: An example using a second variable (p) to reduce the number of false positives when estimating the presence of data in external memory.

in the local nor in the external memory, there is no need to continue verifying its availability in these two memory levels. A more efficient scheme is to verify directly the data arriving in the ISB from the producer.

Figure 5.6 depicts the concurrent implementation of the optimization scheme #3. As shown, the ISB consists of two concurrent processes (i.e., store process, and load process). For each store process, the ISB gets the produced index and data from the producer, and gets the new requested index from the consumer. The ISB stores the produced index and produced data into the local or external memory only if the requested index from the consumer is different from the produced index.

In concurrent, for each load process, the ISB first check the if the previous requested index directly loaded from the producer or loaded from memories (i.e., solved =1). If the previous requested index consumed, the ISB gets the requested index from the consumer. When the requested index from the consumer is the same as the current producer index, the ISB deliver the producer data directly to the consumer. For the producer indexes different from the requested ones from the consumer, the ISB checks the availability of the requested index/data in the local memory and external memory once. If the requested index is neither available in local memory or external memory nor is loaded directly, the ISB will only verify

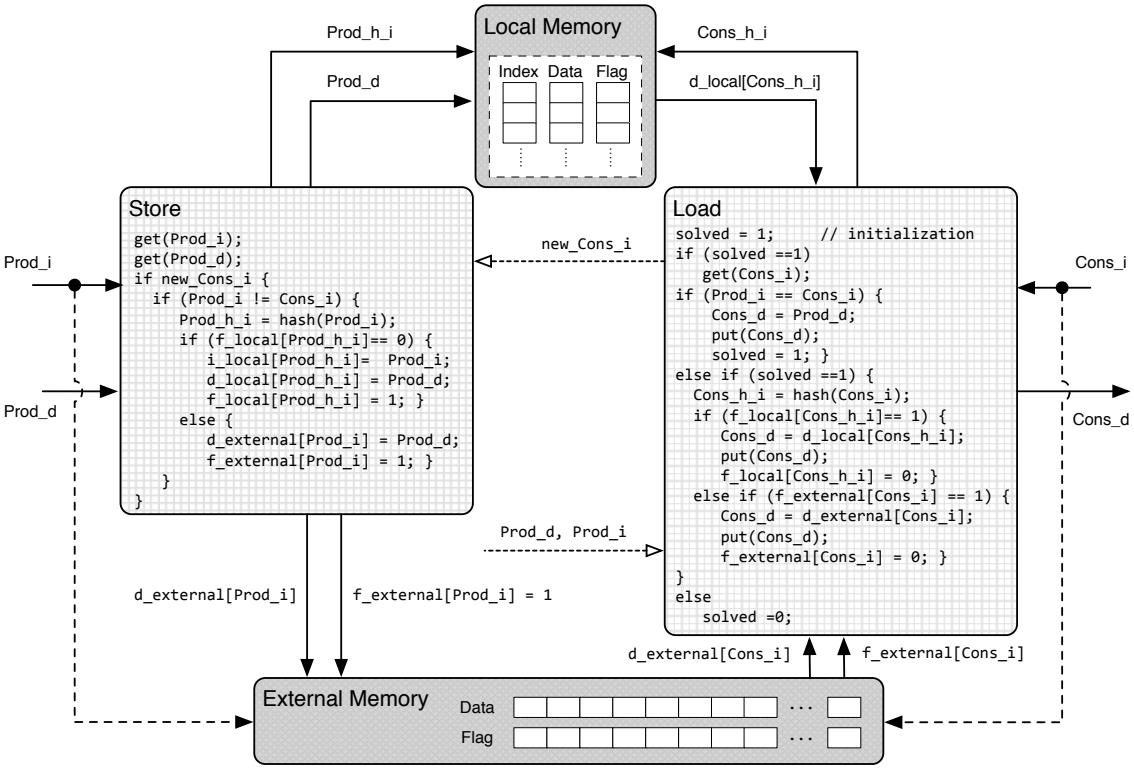


Figure 5.6: The concurrent implementation of the optimization scheme #3.

directly the current requested index with the index arriving from the producer in the ISB.

Figure 5.7 also presents the sequential implementation of the ISB optimization scheme #3. As shown, the ISB first reads the index/data from the producer and the consumer. The ISB stores the producer index/data into the local or external memory when ever is not possible to deliver the producer index/data directly to the consumer. In addition, the ISB accesses to the local memory and/or external memory when the previous requested index from the consumer is not delivered directly and the memory check is allowed. Note that we use *memory_check* flag to control the accesses to the local and the external memories.

Note that comparing the produced index with the requested consumer index without storing the produced index into the local memory might be an efficient solution when the communication pattern between P/C pairs is out-of-order with many stores into the external memory. However, we can also use this optimization scheme for the in-order P/C pairs.

```
// Communicate directly or store the producer index/data into
   the local or the external memory
solved =1 ; // The previous requested data loaded directly
memory_check =1 ; // Memory checking is allowed
get (Producer_Index);
get (Producer_Data);
if Solved == 1 then
  get(Consumer_Index);
Producer_h = hash(Producer_Index);
if (Producer_index == Consumer_index) then
  Consumer_Data = Producer_Data;
  put(Consumer_Data);
  solved = 1; memory_check = 1;
else if flag_local[Producer_h]==0 then
  index_local[Producer_Index] = Producer_Index;
  data_local[Producer_h] = Producer_Data;
  flag_local[Producer_h]=1;
  solved = 0;
else
  data_ext[Producer_Index] = Producer_Data;
  flag_ext[Producer_Index]=1;
  solved = 0;
// Load from local or external memory
if ((solved != 1) & (memory_check ==1)) then
  Consumer_h = hash(Consumer_Index);
  if flag_local[Consumer_h]==1 then
    Consumer_Data=data_local[Consumer_h];
    put(Consumer_Data);
    flag_local[Consumer_h]=0;
    solved = 1;
  else if flag_ext[Consumer_Index]==1 then
    Consumer_Data = data_ext[Consumer_Index];
    put(Consumer_Data);
    flag_ext[Consumer_Index]=0;
    solved = 1;
  else
    Solved = 0; // Neither in local memory nor in external
    memory
    memory_check =0;
```

Figure 5.7: An optimization scheme #3 for each store from the producer into the local or external memory, and for each load from the local or external memory to the consumer.

5.2.5 Main Memory Accesses: Scheme #4

We can use the advantage of both previous optimization schemes to reduce the number of false positives and to use the direct communication technique for the ISB scheme. In this optimization scheme, we use a combination of the optimization scheme #2 and scheme #3.

As mentioned previously in Section 5.2.3 (scheme #2), for each store into the external memory, the ISB calculates the bitwise *OR* of the stored data in the external memory with the current v value (see, Figure 5.2) and uses the second variable (p) to reduce the number of false positives. Each time the consumer requests and the data is neither in the local nor in the external memory, the ISB avoids checking the local and the external memory for this index as it waits for the producer to deliver the data. Note that this is in the case of the (1:1) ratio between P/C pairs.

Figure 5.8 presents the pseudo-code of this optimization scheme when the ISB reads index/data from the producer and stores the index/data into the local or external memory when the producer indexes are different from the requested ones. As shown, the ISB compares the producer and consumer indexes and delivers data from the producer to the consumer directly without storing it into the local or external memory. In addition, for each read from the external memory, the ISB updates the value of v and p and subtracts the bits of the index from the current value of p in a similar way of the scheme #2.

For the case of P/C pairs with the ratio of $(1:N)$, the ISB compares the producer and consumer indexes in a similar way, and communicates the first data from the producer to the consumer directly and sets the flag into the local or external memory to be available for the other $N-1$ requests from the consumer.

```
// Each reads of the ISB from the producer and stores into the
   local or external (main) memory.
solved =1 ;          // The previous requested data loaded directly
memory_check =1 ; // Memory checking is allowed
v=0; p[ ]=0; // v and p initialization
mask = 1;
bits=number of bits for the index;
get (Producer_Index); get (Producer_Data);
if Solved == 1 then
 └ get(Consumer_Index);
Producer_h = hash(Producer_Index);
if (Producer_index == Consumer_index) then
 │ Consumer_Data = Producer_Data;
 │ put(Consumer_Data);
 │ solved = 1; memory_check = 1;
else if flag_local[Producer_h]==0 then
 │ index_local[Producer_Index] = Producer_Index;
 │ data_local[Producer_h] = Producer_Data;
 │ flag_local[Producer_h]=1;
 │ solved = 0;
else
 │ v = Producer_Index OR v;
 │ tmp_Index = Producer_Index;
 │ for (i=0; i<bits; i++) do
 │   │ j = tmp_Index AND 1;
 │   │ tmp_Index = tmp_Index ≫ 1;
 │   │ if (j == mask) then
 │   │   └ p[i]= p[i] + 1 ; // Update p
 │   └
 │ data_ext[Producer_Index] = Producer_Data;
 │ flag_ext[Producer_Index]=1;
 └ solved = 0;
```

Figure 5.8: An optimization scheme #4 when the ISB stores data into local or external memory when the producer indexes are different from the requested ones.

5.3 Summary

The main goal of the optimization techniques presented in this chapter is to maximize the usage of local (on-chip) memory and also to reduce the number of accesses to the main (off-chip) memory. We presented two groups of optimization techniques for the FIFO-based synchronization and ISB-based synchronization schemes. The main idea of FIFO-based optimization scheme is to reduce memory contention by

buffering locally the outputs of the consumer stage and then storing them in data chunks. In this scheme, we provided a temporary buffer for the consumer to reduce memory contention by buffering locally the outputs of the consumer stage and then storing them in data chunks into the external memory at once.

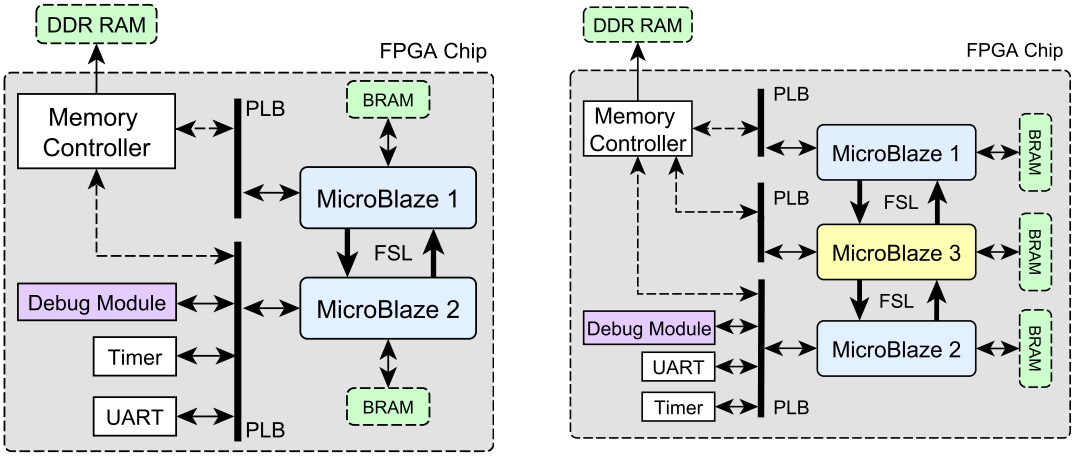
In addition, the main contribution of our ISB-based optimization technique is to maximize the usage of local (on-chip) memory. In previous ISB schemes without optimization, for each miss to the local memory, the ISB reads the flag of the requested index in the external memory. Therefore, the ISB always needed to read the flag even when the data for the requested index is not still available in the external memory. Thus, we need a solution to indicate if the data related to the requested index can be available or not. We presented different optimization techniques such as an OR-ing scheme which can be one of the reasonable solutions to solve this problem. Note that the OR-ing optimization technique may not provide an accurate solution and may result many false positive cases. In order to reduce the number of false positives, we extended the OR-ing optimization technique. In addition, in order to maximize the usage of local (on-chip) memory, we studied the impact of 12 different general hash functions on performance and usage of local memory.

CHAPTER 6

Experimental Results

Chapter Outline

In this chapter, we describe the hardware and software platforms to evaluate our Task-Level Pipelining (TaLP) approach. The experimental results address both fine- and coarse-grained approaches and the impact of the optimization techniques. The evaluation of TaLP focuses on seven benchmarks including common image processing tasks and uses FPGA (Field-Programmable Gate Arrays)-based multicore architectures to implement our approach. Furthermore, the results using our TaLP approach and a multicore architecture reveal noticeable performance improvements for a number of benchmarks over a single core implementation without using TaLP.



(a) (b)

Figure 6.1: FPGA prototype system block diagram with: (a) two MicroBlazes; (b) three MicroBlazes.

6.1 Hardware and Software Platforms

For evaluating our TaLP approach and the optimization schemes, we use a Genesys Virtex-5 XC5LX50T FPGA Development Board [Dig13]. This board contains a Xilinx Virtex-5 FPGA [Xil15b] used with 1.7 Mbits of fast block RAM (BRAMs) connected to a 256 Mbyte 64-bit DDR2 memory.

Figure 6.1 shows two target architectures which were implemented using Xilinx Embedded Development Kit (EDK) v.12.3 tools [Xil10a]. We use Xilinx MicroBlaze (MB) softcore processors [Xil10c] as cores. The MicroBlaze embedded soft core is a 32-bit RISC processor optimized for implementation in Xilinx FPGAs. This soft core processor provides a set of features such as thirty-two 32-bit general purpose registers, 32-bit address bus, and it is highly configurable, allowing us to select a specific set of features required in the target architecture.

Table 6.1 shows the configuration of MicroBlaze softcore processors in our target architectures. As shown, the hardware integer divider, integer multiplier and the barrel shifter are enabled and the MicroBlaze caches are disabled. The size of BRAMs of MicroBlaze processors for instruction and for data is 32 KB.

Each MicroBlaze is connected to on-chip local memory (BRAMs), and also to peripherals such as Debug Module, Timer, UART and the memory controller through

Table 6.1: The configuration of MicroBlaze processors used in our target architecture.

Parameter Name	Description	EDK Tool Assigned
Barrel Shifter	Include barrel shifter	Yes
Floating Point Unit (FPU)	Include hardware floating point unit	No
Integer Multiplier	Include hardware multiplier	Yes
Integer Divider	Include hardware divider	Yes
Instruction Cache	Enable instruction cache	No
Data Cache	Enable data cache	No
Bus Interface	Bus for the peripheral accesses of MicroBlaze	PLB
Stream Interface	Bus for the stream accesses of MicroBlaze	FSL
BRAM Size	The size of BRAMs for instruction and for data	32 KB

the Processor Local Bus (PLB) [Xil10b]. The MicroBlaze processors use the Xilinx Fast Simplex Link (FSL) [Xil11b] to communicate directly with each other. In Figure 6.1, MicroBlaze 1 and MicroBlaze 2 are responsible to execute the codes for the producer and consumer, respectively. Although the ISB can be implemented by custom hardware (see, Appendix), we use an additional MicroBlaze (MicroBlaze 3) to implement the ISB schemes.

This architecture may not provide TaLP solutions with the highest performance, as when using efficient multi-port and/or distributed memories, and simultaneous data accesses requested are not performed at the same time. In the architectures used herein, the three MicroBlazes share the same single-port main memory. The ISB scheme implemented by MicroBlaze provides the flexibility and ease of programmability required to explore and evaluate different data communication and synchronization schemes.

The reference clock frequencies of the FPGA development board and of the MicroBlaze processors are 100 MHz and 125 MHz, respectively.

For software compilation, we use a version of gcc with -o2, mb-gcc 4.8.3, targeted to MicroBlaze processors.

6.1.1 FPGA Resources

At this point it is important to present and analyze the FPGA resources needed to implement each architecture considered in the experiments. What is the overhead

in terms of the hardware resources to implement TaLP? In this section we provide answers to this question.

Table 6.2 presents the hardware resources used for four architectures (Baseline, Standard FIFO, Main Memory with empty/full flag, the ISB, and the ISB within the consumer) when using a Xilinx Virtex-5 LX50T FPGA [Xil15b]. As shown, there are increases of 17% (from 18% to 35%) and 19% (from 15% to 34%) for the number of slice registers and the number of slice LUTs respectively. In a similar way, there are increases of 28% (from 38% to 66%); 54% (from 41% to 95%) and only 6% (from 6% to 12%) for the number of occupied slices, total Memory used and the number of DSP48Es respectively, compared with the baseline architecture. This means that, by increasing the hardware resources of the baseline architecture by 54% of the total memory, 28% of number of occupied slices, 19% of the number of Slice LUTs, 17% of the number of slice registers and only 12% of the number of DSP48Es, we are able to implement the architecture with three MicroBlazes (ISB scheme) and thus achieving the highest performance for all benchmarks (see Table 6.5).

Table 6.2: Hardware FPGA resources usage for each architecture schemes used: the ISB within the consumer core and ISB as a separate core (MB: MicroBlaze).

Device Utilization	**Baseline Architecture (1MB)**		**Standard FIFO (2MB)**		**Main Memory (2MB)**		**ISB (3MB)**		**ISB in Consumer (2MB)**	
	Used	Utilized (%)	Used	Utilized (%)	Used	Utilized (%)	Used	Utilized (%)	Used	Utilized (%)
Number of Slice Registers	5,277	18%	7,623	26%	7,623	26%	10,161	35%	7,623	26%
Number of Slice LUTs	4,489	15%	6,752	23%	6,752	23%	9,868	34%	6,752	23%
Number of occupied Slices	2,805	38%	3,839	53%	3,839	53%	4,760	66%	3,839	53%
Total Memory used (KB)	900	41%	1,764	81%	1,764	81%	2,052	95%	1,764	81%
Number of DSP48Es	3	6%	6	12%	6	12%	9	18%	6	12%

6.2 Benchmarks

In our experimental results, the selected benchmarks include some typical kernels of embedded applications, such as signal and image processing, etc. The benchmarks selected are representative of different types of data communication between stages. Table 6.3 presents the benchmarks and some of their properties including the number of nested loop sets, computing stages, array sizes, the communication pattern and the communication ratio between P/C pairs. The benchmark set consists of:

- **Gray-Histogram**: Transforms an RGB image to a gray image with 256 gray levels (first stage) and determines an histogram of the pixels in the gray image (second stage);

- **Matrix Addition**: Adds three matrices ($A+B+C$);

- **Wavelet Transform**: Applies 1-D Haar horizontally on an input array, transposes the array, applies 1-D Haar vertically, and finally transposes the output array;

- **Fast DCT (FDCT)**: Applies a Fast 2-D Discrete Cosine Transform algorithm. The first stage performs a vertical 1-D FDCT on columns within each block and the second stage performs a horizontal 1-D FDCT on each 8×8 block;

- **FIR-Edge**: Applies a 2-D finite impulse response (FIR) filter followed by an edge detection task. The first stage performs the FIR filter considering window blocks with size of 3×3 of the image. The second stage applies the edge detection on the image resultant from the FIR stage;

- **Edge-Detection**: Detects the edges in a 256 gray-level image which relies on a 2-D convolution routine to convolve the image with kernels (Sobel operators) that expose horizontal and vertical edge information. The program calculates the normalization factor of the kernel matrix and convolves the input image with the kernel horizontally (first stage) and vertically (second stage). The original code is from the UTDSP benchmark suit repository [UTD98];

- **Gaussian blur**: Blurring an image using a Gaussian function. The program sets up the Gaussian convolution kernel and convolves the image with the smoothing kernel horizontally (first stage) and vertically (second stage).

In order to provide a P/C data communication model, the original sequential code of the benchmarks is partitioned into the separate computing stages (producer and consumer), being each stage a sequence of loops or nested loops. In Table 6.3, we can see that almost all the 7 benchmarks consist of two natural stages (S1 and

Table 6.3: Benchmarks used in the experiments (Comm.: Communication).

Benchmark	Nested Loop Sets	Stages	Input Array Size	Comm. Pattern	Comm. Ratio
Gray-Histogram	2	$S_1 \mid S_2$	800×600	in-order	$(1:1)$
Matrix-Add $(A+B+C)$	2	$S_1 \mid S_2$	256×256	in-order	$(1:1)$
Wavelet Transform	4	$(S_1 - S_2) \mid (S_3 - S_4)$	800×600	out-of-order	$(1:1)$
Fast DCT (FDCT)	2	$S_1 \mid S_2$	800×600	out-of-order	$(1:1)$
FIR-Edge	2	$S_1 \mid S_2$	256×256	out-of-order	$(1:N)$
Edge-Detection	2	$S_1 \mid S_2$	128×128	out-of-order	$(1:N)$
Gaussian blur	2	$S_1 \mid S_2$	64×64	out-of-order	$(1:N)$

S2) based on the number of sets of nested loops. The exception is the Wavelet Transform which has four natural data-dependent computing stages (S_1, S_2, S_3 and S_4). In order to use two stages per benchmark we split the Wavelet Transform in the following two stages ($S_1 - S_2 \mid S_3 - S_4$). Therefore, for the Wavelet Transform, we will have two pairs of tasks without pipelined execution (i.e., $T_1 - T_2$, and $T_3 - T_4$).

6.3 Performance Evaluation

In order to evaluate the performance of our TaLP approach, we first measure the execution clock cycles of each computing stage used in our experiments when using a single MicroBlaze without TaLP (see Table 6.4).

In order to have a first idea about the TaLP performance impact, we calculate the highly optimistic theoretical speedup bounds (herein: Upperbound) for each application. In Table 6.4, A shows the theoretical upperbound speedups as calculated with Equation 4.1 (see Section 4.3.6). In order to have an idea about the possible upperbound (also optimistic) when data are communicated between the two tasks (stages) using local (on-chip) buffers, we include upperbound B speedups. These were calculated with the execution time of each stage considering the unrealistic scenario of communicating inter-stage data using internal FIFOs (as if data communication could be in-order) instead of randomly store/load in/from memory (local or external). The intention of the B upperbound speedups is to first show the impact when data are all directly (on-chip) communicated (even if the model also unrealistic considers that communications can be all in-order). Note that for the benchmarks

with more than two computing stages such as Wavelet Transform, we compute the upperbound speedups using Equation 4.2 (see Section 4.3.6).

Table 6.4: The execution clock cycles of the benchmarks without using TaLP and the theoretical Upperbound speedups when using TaLP.

Benchmark	Clock Cycles			Theoretical Upperbound Speedups	
	S_1	S_2	Overall	A	B
Gray-Histogram	46,848,188	23,571,490	70,419,678	1.86×	2.10×
Matrix-Add $(A+B+C)$	5,814,997	5,814,997	11,629,994	2.00×	2.38×
Wavelet Transform	69,660,426	67,260,579	136,921,005	1.99×	2.42×
Fast DCT (FDCT)	27,450,111	27,816,150	55,266,261	1.61×	1.85×
FIR-Edge	27,546,828	20,245,503	47,792,331	1.72×	1.73×
Edge-Detection	7,891,875	7,152,558	15,044,433	1.91×	1.94×
Gaussian blur	1,544,962	263,459	1,808,421	1.17×	1.28×

Regarding to the execution clock cycles of the benchmarks, we can consider some of the benchmarks as well-balanced P/C pairs. For instance, the execution time of the producer stage for the Matrix-Add, FDCT and Edge-Detection is approximately equal to the execution time of their consumer stage. The theoretical upperbound A for the well-balanced benchmarks is close to the maximum possible value for upperbound (2x). This would correspond to the execution time equally split over the two tasks and an optimistic fully overlapping of execution of the tasks. In contrast, the execution time of the producer stage for Gaussian blur is not equal to the execution time of the consumer stage, i.e., the stages for Gaussian blur are not balanced and the idle time of the consumer is considerable and the performance impact by applying TaLP for Gaussian blur might not be noticeable.

6.4 Fine-grained Schemes Results

We start by evaluating the performance impact of fine-grained schemes for TaLP. Table 6.5 shows the speedups obtained when considering fine-grained data synchronization schemes with TaLP vs. a single core baseline architecture for both in-order and out-of-order benchmarks. The baseline architecture consists of a MicroBlaze with computing stages executing sequentially. The execution time of this scheme provides a criterion to compare the performance impact of different pro-

posed coarse/fine-grained data synchronization and communication approaches using TaLP.

Note that since in fine-grained data synchronization schemes, FIFO channels are not suitable for out-of-order benchmarks, the use of the FIFO scheme is not evaluated in Table 6.5 (herein: N/A). We consider $N = 50$ datasets (e.g., images) being computed in all measurements.

Table 6.5: Speedups obtained when considering fine-grained data synchronization schemes with TaLP vs. a single core baseline architecture, considering $N = 50$ datasets (e.g., images) being computed. The size of the local ISB memory is 1,024 data elements. The "Main Memory" correspond to the empty/full approach without the ISB (the scheme presented in Figure 3.12).

Benchmark	Standard FIFO	Main Memory	Inter-Stage Buffer (ISB)	ISB in Consumer
Gray-Histogram	1.58×	1.03×	1.65×	1.65×
Matrix Addition	1.82×	1.35×	1.91×	1.91×
FDCT	N/A	0.89×	1.38×	1.37×
Wavelet Transform	N/A	1.18×	1.46×	1.27×
FIR-Edge	N/A	1.14×	1.57×	1.21×
Edge-Detection	N/A	0.85×	1.39×	1.21×
Gaussian blur	N/A	0.38×	1.14×	0.55×
Geometric Mean	1.70×	0.92×	1.48×	1.24×

As shown in Table 6.5, the highest achieved speedups for Gray-Histogram and Matrix Addition (in-order benchmarks) are 1.65× and 1.91, respectively, reported when using the ISB between the cores and the ISB in the consumer.

However, when using a simple FIFO (column "Standard FIFO"), when the producer loads from the main memory, the consumer may store into the main memory at the same time and it causes memory access conflicts that are solved by serializing the accesses. Table 6.5 shows that the achieved speedups for the Gray-Histogram and the Matrix Addition for the ISB in the consumer schemes are the same as the achieved speedups considering the ISB between P/C pairs (1.65× for the Gray-Histogram and 1.91× for the Matrix Addition) and are close to the theoretical speedup bounds of 1.86× (Gray-Histogram) and 2× (Matrix-Addition).

We obtained the highest speedup for out-of-order benchmarks in the fine-grained data synchronization model when using the ISB between the producer and the con-

sumer. In addition, the lowest achieved speedups for all benchmarks are obtained when using the empty/full bit approach without using the ISB (direct loads/stores to the main memory), due to the no promotion of data transfers between P/C pairs to on-chip direct communication and the high value of simultaneous memory requests.

With out-of-order applications, most data are stored in external memory. In the ISB within the consumer scheme, data are loaded/stored from/to the external memory if the requested index from the consumer is not equal to the produced index or the data element correspondent to the requested index is not available in the local memory. Therefore, load/store data from/to the main memory in the ISB within the consumer scheme is only considered when it cannot be stored locally. As a result for all out-of-order benchmarks, the performance achieved in the scheme with the ISB in the consumer is lower than the speedup in the ISB scheme.

By considering the theoretical upperbound A of the benchmarks (see Table 6.4), the highest achieved speedup for the FDCT (1.38×), Wavelet Transform (1.46×), FIR-Edge (1.57×), Edge-Detection (1.39×) and Gaussian blur (1.14×) are 90.7% (for the FDCT), 69.3% (for the Wavelet Transform), 91.3% (for the FIR-Edge), 72.8% (for the Edge-Detection) and 97.4% (for the Gaussian blur) close to their theoretical speedup.

6.4.1 Impact of Data Chunks

One of our proposed architectures considers an extra FIFO connected to the ISB to provide temporary buffering chunks of the data output by the consumer side (see Section 5.1). Here we analyze the performance of this architecture.

Figure 6.2 shows the impact of buffering locally the outputs of the consumer stage in the performance when using the FIFO #1 (between P/C pairs) with the size of 1, 8 and 16, and the FIFO #2 (connected to the consumer) with various sizes (from 1 to 2,048). Note that for the FIFO #1 with the size greater than 8 (e.g., 16), we achieved the same results as with size equal to 8. As shown, the highest performance is achieved when the size of FIFO #1 is equal or greater than 8 and the size of FIFO #2 is less than 32 (e.g., 1,4,8 and 16). For the Gray-Histogram

and the Matrix Addition (in-order benchmarks) the highest speedups are 1.70× and 1.89× when the size of FIFO #1 is equal to 8 or 16.

In addition, by considering the highest achieved speedup in Figure 6.2 for the Gray-histogram and the Matrix Addition, we conclude that adding an extra FIFO to the consumer (FIFO #2 with the size of less than 32 for Gray-Histogram and less than 16 for Matrix Addition) can delay the stores to the main memory by the consumer.

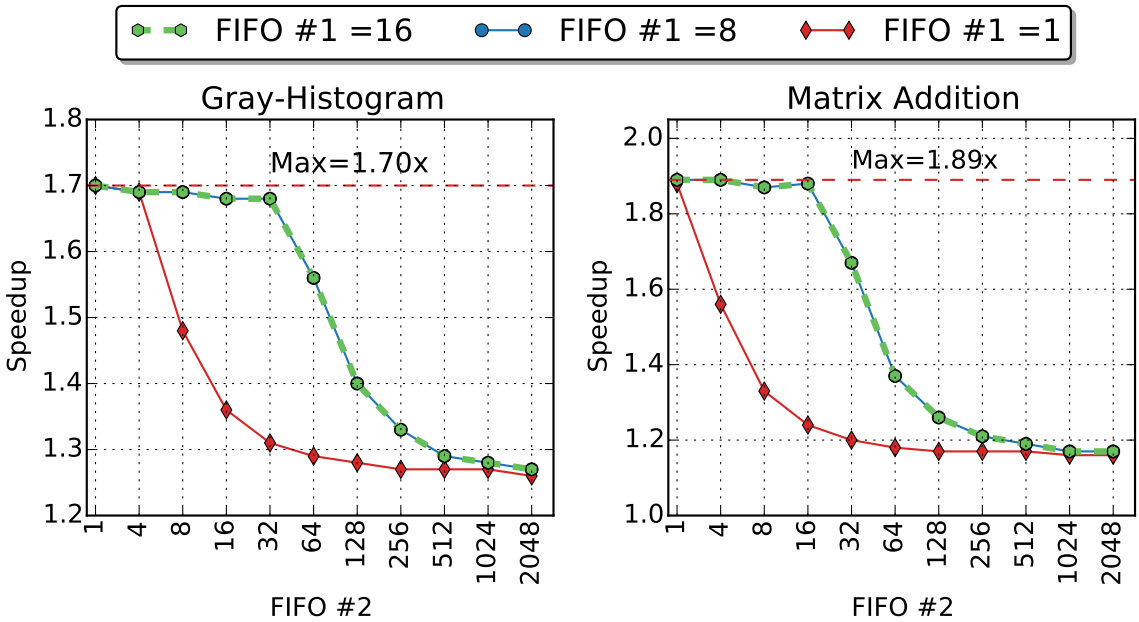


Figure 6.2: The impact of temporary buffering chunks of data by the consumer on performance when using a FIFO scheme and on-chip buffering in the consumer (Figure 5.1) by considering FIFO 1 (between P/C pairs) with the size of 1, 8 and 16 and FIFO 2 with size between 1 and 2,048.

6.4.2 Impact of the Local memory Size

The ISB scheme presented in Section 4.1.1 includes a local buffer. The intention is to store as many as possible data items communicated between P/C pairs in that local buffer. A question that may arise is related to the impact of the size of the local buffer on performance. Here we address this question and provide results when varying the size of the buffer.

Here we evaluate the impact of increasing the size of the local memory of the ISB (up to 4,096 words considering the limitation on the number of BRAMs on our

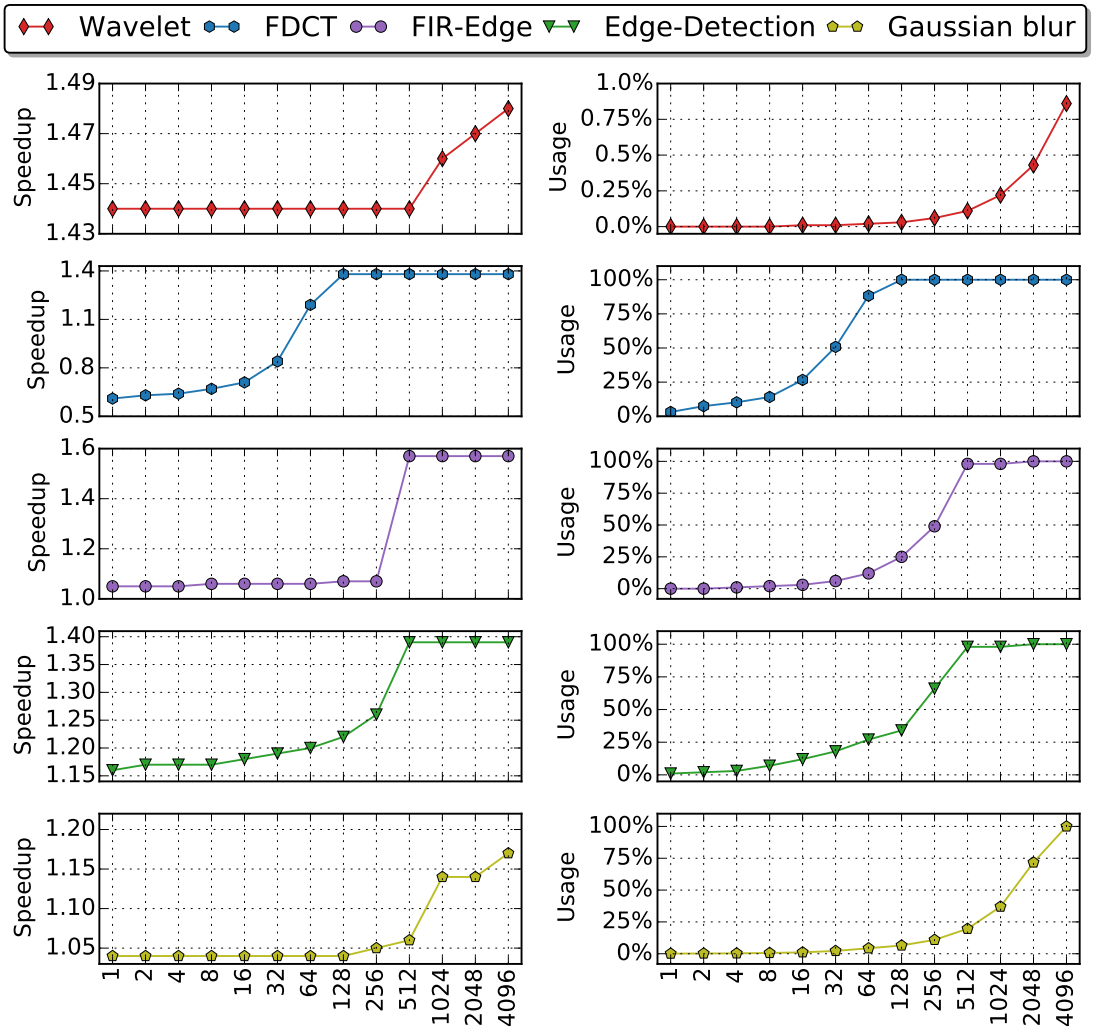


Figure 6.3: The impact of increasing the ISB buffer size on speedup (on left) and the percentage of data communicated between stages using the local memory (Usage) results (on right).

FPGA) on the local memory usage and on the achieved speedups. Figure 6.3 shows the results achieved. The highest speedup for all benchmarks is obtained when the usage of local memory is maximum. For example, in FDCT we obtain the maximum usage of local memory (100%) and the highest speedup (1.38×) with the size of local memory equal to 128. The maximum usage of local memory is obtained when the size of the local memory is greater than 512 for Edge-detection, 2,048 for FIR-Edge and 4,096 for Gaussian blur.

The communication pattern in the Wavelet Transform benchmark allowed only 0.86% use of local memory with the maximum size for the local memory (4,096). Simulation-based experiences indicate that for the Wavelet Transform, we obtain

the maximum usage of local memory when considering 512 KB for the size of the local memory. Note that the total number of data elements to transfer between P/C pairs in Wavelet Transform is 480,000 (800×600).

6.4.3 The Impact of Hash Functions

The hash function used in the ISB can influence significantly the data communicated using the local buffer. Here we evaluate the impact of various hash functions in order to understand if the option used is the one achieving the best results.

Figure 6.4 shows the results of using different hash functions in the ISB scheme and the impact of the hash functions presented in Section 5.2.1 on the usage of local (on-chip) memory considering 1,024 integer data elements for the local memory to maintain the same size used in the previous experiments. We show the number of misses (number of times requesting data not present in the memory accessed) when accessing the local memory for each hash function. As shown in Figure 6.4 (left), the minimum number of misses in all benchmarks is achieved when using the hash functions $H1$ ($Modular$), $H11$ ($Open\ Addr$) and $H12$ ($Comp$). As shown, the $H1, H11$ and $H12$ hash functions use the maximum (100%) of local memory for the local memory size considered. If a hash function uses less than 100% of the local memory, it means that there are local memory locations not being used and the ISB may have to switch to the main memory to store data while the local memory is still not full. The results also show a very low usage of local memory for Wavelet Transform (below 0.22%) followed by a low usage for Gaussian blur (below 40%).

To evaluate the impact on the performance of TaLP when using different hash functions, we measured the minimum required size of the local memory to achieve the maximum usage in the ISB scheme for different hash functions (see Table 6.6). The results illustrate minimum required sizes of the local memory from 128 Bytes to 64 Megabyte. For all benchmarks, the minimum required size of the local memory is achieved when we use $H1, H11$ or $H12$. From the results shown and based on the complexity of the hash functions, we conclude that for these benchmarks, the hash function $H1$ is the best option. Note that $H1$ uses a number of least significant bits

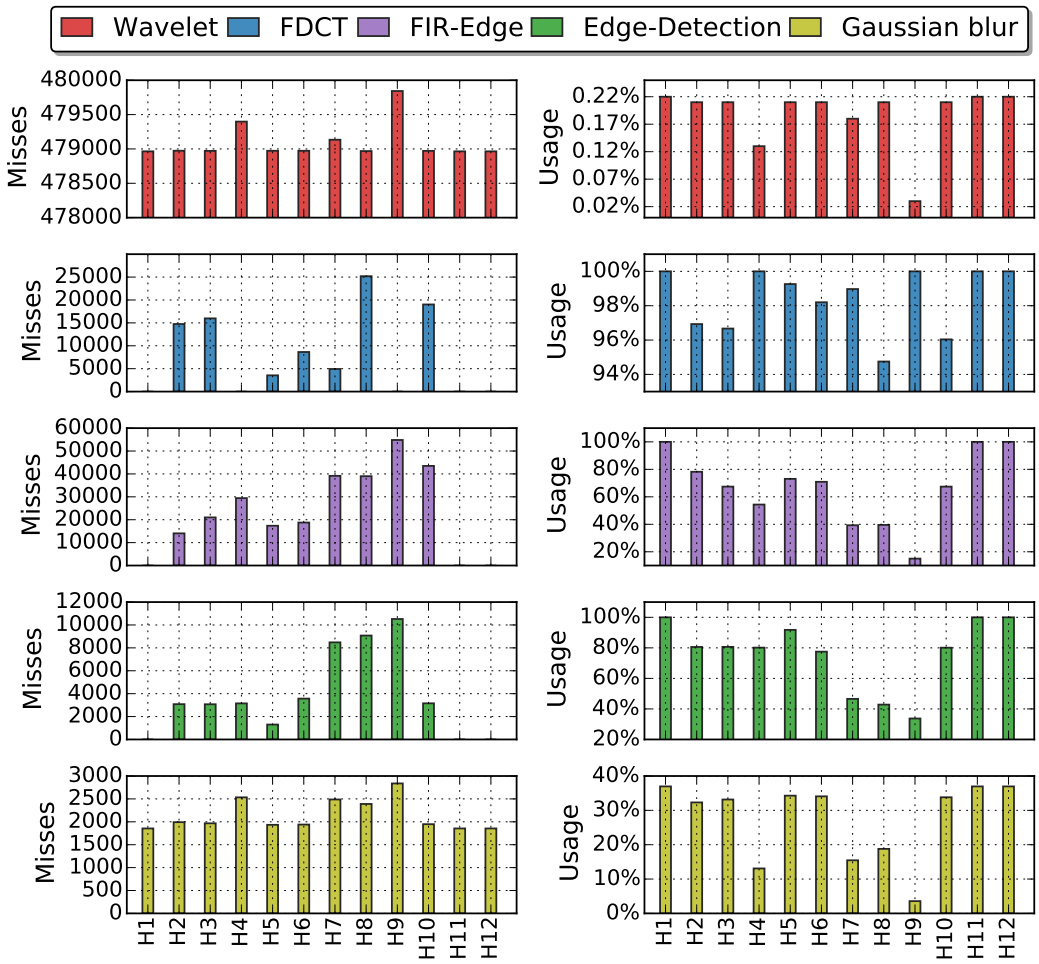


Figure 6.4: The impact of using different hash functions for mapping data into local memory in the ISB Scheme on the percentage of data communicated between stages using the local memory (Usage) and the number of misses when accessing local memory.

(LSBs) of the index and thus does not require heavy operation (a simple AND to a mask or the simple wire connections in the case of custom hardware can be used).

Table 6.6: Minimum sizes required to achieve the maximum usage of local memory for different hash functions (B:bytes; K:KB; M:MB). Sizes in bold are the minimum for each benchmark/hash function.

Benchmarks	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12
Wavelet Transform	**512K**	16M	16M	8M	16M	16M	32M	8M	64M	16M	**512K**	**512K**
FDCT	**128B**	4K	256K	256B	128K	32K	2K	2K	512B	256K	**128B**	**128B**
FIR-Edge	**1K**	128K	256K	8K	128K	128K	64K	64K	32K	256K	**1K**	**1K**
Edge-Detection	**512B**	32K	256K	8K	64K	32K	16K	16K	64K	256K	**512B**	**512B**
Gaussian blur	**4K**	64K	256K	16K	128K	128K	128K	128K	128K	256K	**4K**	**4K**

6.4.4 Optimization Results

The use of the optimization schemes to approximately track the possible data stored in the external memory, can reduce the number of accesses, the number of misses and the number of false-positives to external memory, and may improve the speedups achieved when using TaLP. Here we evaluate the impact of our optimization techniques for the ISB-based scheme (see Section 5.2).

Table 6.7 shows the speedups obtained when considering an ISB synchronization scheme with/without different optimizations vs a single core baseline architecture. For instance, as shown in Table 6.7, the results of the ISB scheme using the optimization schemes illustrate speedups from 1.16× to 1.71× (improvements between 1.75% and 8.92% over the implementation without optimization). The highest performances for all benchmarks are obtained when using the optimization schemes #3 and #4.

Table 6.7: Speedups obtained when using an ISB w/ and w/o optimization schemes vs a single core, and compared with theoretical upperbound speedups (w/:with; w/o: without; Opt: optimizations).

Benchmark	Upperbound A	Upperbound B	w/o Opt.	w/ Scheme #1	w/ Scheme #2	w/ Scheme #3	w/ Scheme #4
Wavelet Transform	1.61×	1.85×	1.44×	1.46×	1.47×	1.48×	1.48×
FDCT	1.99×	2.42×	1.38×	1.43×	1.44×	1.61×	1.55×
Fir-Edge	1.72×	1.73×	1.57×	1.71×	1.71×	1.71×	1.72×
Edge-Detection	1.91×	1.94×	1.39×	1.49×	1.49×	1.50×	1.50×
Gaussian blur	1.17×	1.28×	1.14×	1.16×	1.16×	1.17×	1.16×
Geometric mean	1.65×	1.81×	1.38×	1.44×	1.44×	1.48×	1.47×

Although in the case of Wavelet Transform, FDCT and Edge-Detection benchmarks the speedup of the ISB scheme without optimization over the execution of the benchmark without using TaLP is considerable, the use of the optimized ISB schemes allows further speedup improvements. For these benchmarks, the gap between the real achieved speedup and the theoretical speedup (e.g., 1.48× to 1.85× for Wavelet Transform) possibly indicates potential for further improvements. However, for the case of FIR-Edge and Gaussian blur benchmarks, the results are fairly close to the theoretical speedup upperbounds, 1.71× vs. 1.72× (1.73×) for FIR-Edge and 1.16× vs. 1.17× (1.28×) for Gaussian blur when using the ISB with the optimization schemes.

The number of misses reduces to zero for the FDCT (from 49 misses to 0), FIR-Edge (from 506 misses to 0) and Edge-Detection (from 246 misses to 0) benchmarks by using the optimization scheme which reduces the number of false positives (scheme #2). Also, for Wavelet Transform and Gaussian blur, the number of misses was reduced by 12.35%. This allowed a reduction of the latencies of the benchmarks from 1% to 8.32%. However, these improvements have a minor impact on the speedups already achieved in the previous optimization scheme (scheme #1).

The geometric means of the speedups obtained when using the optimization schemes and when using the ISB scheme without the optimization, we obtained the 4.3% (from 1.38× to 1.44×) speedup increases when using the optimization schemes #1 and #2, the 7.25% (from 1.38× to 1.48×) when using the scheme #3 and 6.52% (from 1.38× to 1.47×) when using the combination of both optimization schemes #2 and #3 (scheme #4).

Note that the reduction of main memory accesses certainly contributes to power reductions and energy savings (future plans include measuring the impact on power and energy).

6.5 Results with Coarse-grained Schemes

In this section, we present the results of the two coarse-grained architectures using FIFOs between P/C pairs and a shared main memory previously presented in Section 4.2. Figure 6.5 and Figure 6.6 show the achieved speedups when using a single FIFO (Coarse-grained One FIFO) and two FIFOs (Coarse-grained Two FIFOs) between P/C pairs, respectively.

In these experiments, we consider a number of data chunks (N) being computed. For instance, in the Coarse-grained Scheme with a single FIFO (Figure 4.12), the number of the temporary data chunks (M) is equal to the number of data chunks being produced/consumed. As expected, if $N = 1$ and $M = 1$, the producer waits for the availability of temporary data chunk in external memory. Thus, the producer and the consumer execute almost sequentially and therefore, no speedups are achieved. When the number of temporary data chunks in external memory is $M > 1$,

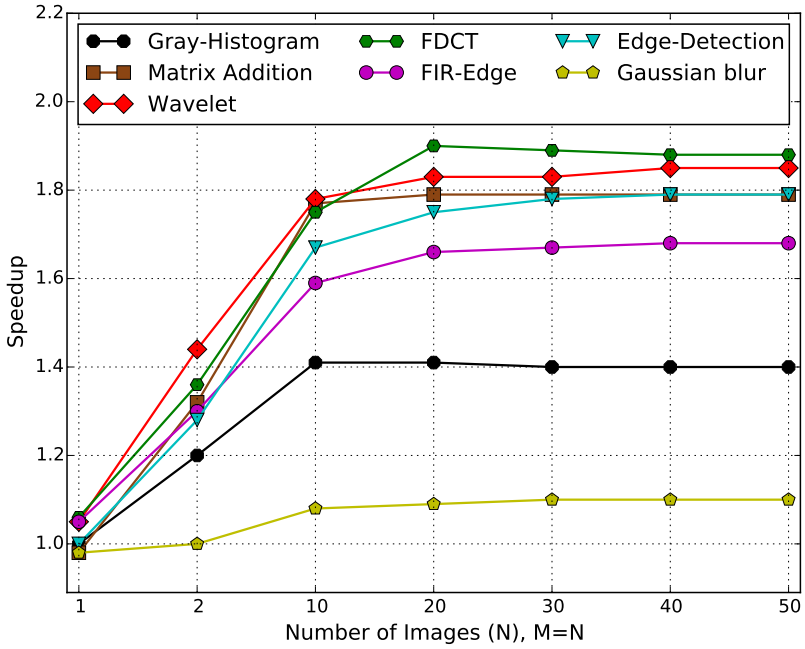


Figure 6.5: Speedups achieved by considering coarse-grained data synchronization schemes for datasets (e.g., images) using a single FIFO.

the producer can process the next data chunk while the consumer is consuming (processing) the previous data chunk. The results in Figure 6.5 show that increasing the number of data chunks being computed (N) and allowing two temporary data chunks being stored ($M = 2$) in external memory significantly increases the performance. The performance significantly increases with values of N between 2 and 20 and stays almost the same for $N > 20$.

Figure 6.6 shows the results when using two FIFOs between P/C pairs. In this scheme, the number of temporary data chunks is $1 < M \leq N$. Based on the limitation of available memory on our experimental board, we consider the range of 1 to 32 for temporary data chunks (M) and $N = 50$. The results show that the performance with the number of temporary data chunks $M = 2$ is very close to the one obtained when increasing the value of M. This is somehow expected as the existence of only one core for the producer and one core for the consumer only allows two temporary data chunks being processed at the same time (one produced and one consumed). In this case, the producing of additional data chunks while the consumer is still consuming the previous one, or the consuming of another data chunk while the

producer is still being producing the next one, seems as expected to have a small impact on performance.

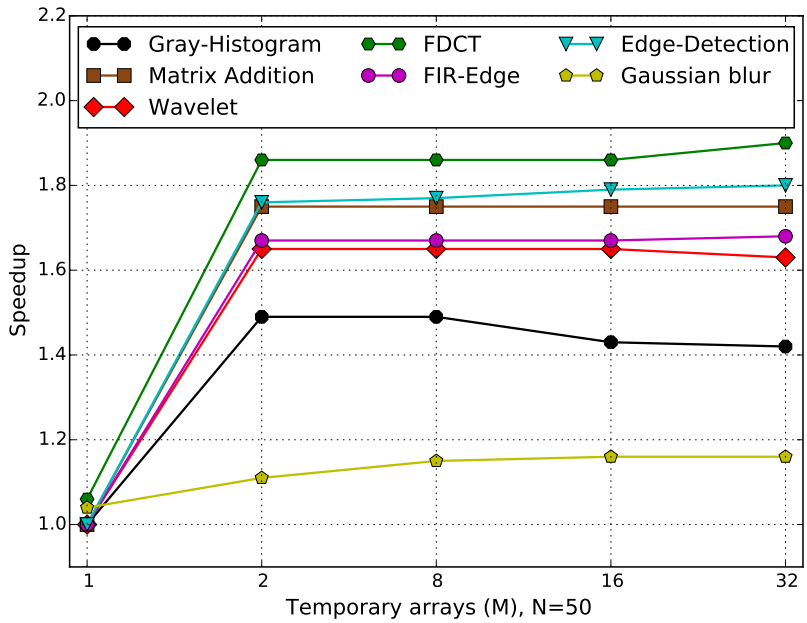


Figure 6.6: Speedups achieved by considering coarse-grained data synchronization schemes for datasets using two FIFOs.

6.6 Summary

This chapter presented the experimental results of our fine- and coarse-grained approaches to provide TaLP for in-order and out-of-order applications.

In order to implement and evaluate our approach, we have used an FPGA development board. In our target architecture, we have used a shared main memory between cores and local memory for each core. Although using shared main memory in our experiments may not provide the highest performance compared with other types of memory systems such as distributed memory, we achieved noticeable performance improvements comparable with the theoretical speedups of the benchmarks.

We evaluated the performance when considering different fine-grained data synchronization schemes with TaLP vs. a single core baseline architecture. The results show considerable speedups when using an ISB scheme for all types of communi-

cation patterns between P/C pairs. Also, we evaluated the impact of increasing the ISB buffer size on speedups and the percentage of data communicated between stages using local memory. The obtained results show the minimum size required of the local memory to achieve the highest performance and to achieve the maximum usage of the local memory. We also measured the impact of using different hash functions for mapping data into the local memory in the ISB scheme on the percentage of data communicated between stages using the local memory (usage) and the number of misses when accessing local memory. The results show that three hash functions (H1, H11, and H12) achieved the minimum size required to achieve the maximum usage of local memory in all benchmarks.

In addition, we provided the results of optimization techniques such as the optimization for FIFO-based and ISB-based schemes. In the FIFO-based optimization schemes, we evaluated the impact on performance when using an on-chip buffering in the consumer. We obtained the maximum performance for in-order applications (1.7× for Gray-Histogram and 1.89× for Matrix Addition) when the size of the FIFO between P/C pairs is equal or greater than 8 and the size of the FIFO used in the consumer is less than 32. For coarse-grained data synchronization, we presented the results when using one FIFO or two FIFOs between the P/C pairs. The results show significant performance improvements for both in-order and out-of-order benchmarks.

In summary, we can conclude that our ISB synchronization scheme is an efficient solution for providing TaLP for both in-order and out-of-order P/C pairs.

CHAPTER 7

Conclusions and Future Work

Chapter Outline

Nowadays, techniques to accelerate the execution of applications in multicore architectures are becoming increasingly popular. Task-level Pipelining (TaLP) is one of the important techniques to speedup the execution of tasks for multicore based systems especially when dealing with applications consisting of producer/consumer *(P/C)* tasks. For providing TaLP, data communication and synchronization mechanisms between producers and consumers are essential. Classic communication and synchronization approaches for multicore architectures are based on FIFOs and shared-memories to synchronize data communication between P/C pairs. For instance, the FIFO-based communication and synchronization mechanism which provides one of the simplest implementation of TaLP is only sufficient when the sequence of producing data is the same than the sequence of consuming data (referred in this thesis as in-order data communication pattern or simply in-order). However, using FIFO channels between producers and consumers may not be efficient or feasible for out-of-order P/C pairs and it might be necessary and/or useful to use other data communication mechanisms. In addition, using only a shared memory scheme as a communication and synchronization channel between

the P/C pairs may cause an overhead increase in multicore architectures especially when most data communications between P/C pairs are through the shared memory.

In this thesis, we have presented an approach for task-level pipelining (TaLP) in the context of FPGA-based multicore architectures to accelerate the execution of sequential applications. The advantages of using FPGAs to implement our multicore architecture is the simplicity of the design cycle and the opportunities of customizing the architecture and field reprogramability provided by FPGAs. Although using other target architectures (e.g., ASICs) may provide higher performance and/or less power dissipation and energy consumptions, designing multicore architectures can be complex and/or expensive. In addition, our approach can also be implemented by fast processor models and platforms such as Open Virtual Platforms (OVP) [ovp16] which may reduce the complexity of the design. However, using OVP cannot provide a clock cycle accurate platform to evaluate TaLP.

We analyzed and compared different implementations of our fine- and coarse-grained data synchronization schemes for a set of both in-order and out-of-order producer/consumer benchmarks. Our fine-grained scheme also supports out-of-order P/C communications when the number of requests to the same produced data element (multiple reads) by the consumer is more than one. Focused on solving the limitations related to inter-stage communications based on FIFOs in traditional approaches, our fine-grained approach uses a flag-based synchronization supported by hash-indexing and empty/full flag memory to provide efficient parallel synchronization and communication between P/C pairs. We propose a fine-grained inter-stage buffer (ISB) data synchronization scheme which provides TaLP between P/C pairs and, as a result, overcomes the limitations related to the FIFOs.

As one of the goals is to use on-chip data communication between P/C pairs, we researched and developed ISB optimizations. Those optimizations are focused on the reduction of external shared memory accesses as they contribute to data communication overhead. Regarding the possible optimizations, we evaluated the impact on performance for TaLP when using different hash mapping functions into on-chip memory. Finally, we presented optimization schemes for the ISB to reduce the number of accesses to the shared external memory and estimate the presence of

data previously produced and stored into the shared external memory.

All solutions proposed in this thesis were implemented using an FPGA board and the results presented consist of measurements using real hardware.

The main contributions of the thesis are summarized in Section 7.1 and the future work is presented in Section 7.2.

7.1 Main Contributions

The major contributions of this thesis can be summarized as follows:

- We presented a technique for pipelining the execution of sequences of data-dependent loops using fine-/coarse-grained synchronization schemes. We introduced a customized fine-grained ISB (inter-stage buffer) data synchronization and communication scheme which enables the TaLP between P/C pairs. We used seven image processing benchmarks in our experimental results. The benchmarks we selected are representative of different types of data communication between stages. The results with an Inter-Stage Buffer (ISB) between producer/consumer cores show speedups from 1.14× to 1.57× for the benchmarks used when using our multicore-based task-level pipelining approaches over the sequential execution of computing stages in a single core. In general, the results showed that the ISB scheme is an efficient solution for both in-order and out-of-order data communication between producers and consumers. We also presented two coarse-grained data communication and synchronization schemes to provide TaLP between P/C pairs. The results showed that two temporary arrays in the shared main memory is sufficient to achieve significant performance improvements.

- We implemented a customized multicore architecture for the inter-stage communication to achieve pipelining execution of P/C pairs. In order to evaluate our TaLP approach, we presented two FPGA-based target architectures (using two and three cores, respectively) using as cores the Xilinx MicroBlaze [Xil10c] embedded 32-bit RISC processor softcore. We used two MicroBlaze

cores for executing the codes for the producer and consumer, respectively and an additional MicroBlaze to implement the ISB schemes.

- We evaluated the impact on the performance of task-level pipelining using different hash functions. We analyzed and compared 12 different hash functions for a set of out-of-order producer/consumer benchmarks. The results showed the minimum number of misses when accessing the local memory and the maximum usage of local (on-chip) memory for each hash function. Also, the results determined the minimum required sizes of the local memory from 128 Bytes to 64 Megabyte. For all benchmarks, the minimum required sizes of the local memory are achieved when we used the modular ($H1$), open addressing ($H11$) and the complement modular addressing ($H12$) hash functions. The results also showed that small sizes of local memory in the ISB are sufficient to achieve high percentages of inter-stage data communication using local on-chip memory and to achieve close to maximum speedups.

- We presented an optimization technique to improve out-of-order P/C pairs when a consumer use more than once a data element produced by a producer (single write, multiple reads). Our optimization technique provided a reduction of requests from the producer by using a scheme based on shadow memory. We used a function to determine the precise number of requests for each requested element of the consumer. This approach provided the ability to support TaLP for out-of-order benchmarks with single write and multiple reads P/C pairs.

- We presented optimization techniques for the ISB to estimate the presence of data previously produced in the shared main memory. We evaluated optimization techniques for the ISB scheme to maximize the usage of local memory and to reduce the number of accesses to the shared main memory (external). The results with an ISB using our optimization approaches show speedups from 1.16× to 1.71× for the benchmarks used when using our multicore-based task-level pipelining approaches over the sequential execution of computing

stages in a single core. The results also show reductions from 12.35% to 100% of misses to external memory.

7.2 Future Work

The work presented in this thesis can be extended for further improvement. We suggest the following directions for future work.

- Continue the research related to the scalability of our approach for more complex computing stages with more than two computing stages. One possibility might be to use more computing cores and inter-stage buffers (ISBs) between each two cores to support multiple producer-consumer pairs. We previously suggested a possible architecture in Section 4.3 to support multiple computing stages which can use only two cores and two ISBs containing multiple local tables. For this proposed approach, we suggest the following tasks:
  - Investigate the impact of using multi-hash functions on performance when mapping the data elements into the multi-tables;
  - Study the techniques to reuse the idle local memories in the ISBs with multi-local tables to reduce the number of accesses to the shared main memory.
- Providing code restructuring techniques for a given application to generate more suitable code for TaLP and then automatically decide and apply TaLP. We need to provide reconstructing techniques which can automatically identify the computing stages of a given application, determine the data dependencies and the communication pattern between the stages, evaluate the balances between the execution time of the stages, and finally decide if the restructured code is suitable and then apply TaLP.
- Runtime techniques to auto-tune the ISB according to the communication pattern requirements. In this case, the ISB can switch to the suitable scheme for both in-order and out-of-order applications by considering the communication pattern between P/C pairs. For instance, In out-of-order P/C pairs with the

ratio of (1:N), the ISB can automatically use the function to determine the precise number of requests for each requested element of the consumer. However, using this function by the ISB, for P/C pairs with the ratio of (1:1) is not required.

- Study the impact of the granularity when using coarse-grained approaches for streaming applications and evaluating the synchronization and communication grains between data chunks representing arrays and partitions of arrays.

- Techniques to predict the speedup achieved by TaLP and according to a partitioning of computing stages and to a local buffer size in the ISB. One possibility might be to start using the knowledge that the shape of local memory usage in Figure 6.3 might be similar to the shape of speedup when varying the size of the local memory. These techniques can be important to help designers and to support Design-Space-Exploration (DSE).

- Implement customized multicore architectures for the inter-stage communication using hardcore processors (e.g., ARM processors). The implementation of the ISB in hardware as an IP core can be an interesting solution in this context.

- Explore the synthesis of hash functions based on the pattern of a given application which is one possible solution to maximize the local (on-chip) communication between P/C pairs. For instance, for each given application, the requested data elements from the consumer and the produced data elements by the producer can be evaluated and thus generate a suitable hash function to maximize the usage of local (on-chip) memory.

- Study the impact of the approaches proposed in this thesis on power and energy consumption. For instance, one possibility might be to evaluate the use of application-specific ISBs on power reductions and energy saving.

Bibliography

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, September 1995.

[Amp16] Intel vtune amplifier, 2016. URL `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982. ISBN 0137010788.

[Ban88] Utpal Banerjee. Dependence. In *The Kluwer International Series in Engineering and Computer Science*, pages 27–43. Springer Science Business Media, Norwell, MA, USA, 1988.

[BBM+12] Siegfried Benkner, Enes Bajrovic, Erich Marth, Martin Sandrieser, Raymond Namyst, and Samuel Thibault. High-level support for pipeline parallelism on many-core architectures. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 614–625. Springer-Verlag, Berlin, Heidelberg, 2012.

[BF99] G.T. Byrd and M.J. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87 (3):456–466, 1999.

[BGW+13] A. Bardizbanyan, P. Gavin, D. Whalley, M. Sjalander, P. Larsson-Edefors, S. McKee, and P. Stenstrom. Improving data access efficiency by using a tagless access buffer (tab). In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*, pages 1–11. IEEE, 2013.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 72–81. ACM, New York, NY, USA, 2008.

[Bol08] Thomas Bollaert. Catapult synthesis: A practical introduction to interactive c synthesis. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 29–52. Springer, 2008.

[BR96] Stephen Brown and Jonathan Rose. FPGA and CPLD Architectures: A tutorial. *IEEE Design Test of Computers*, 13(2):42–57, June 1996.

[Car05] João M. P. Cardoso. Dynamic Loop Pipelining in Data-driven Architectures. In *Proceedings of the 2Nd Conference on Computing Frontiers (CF'05)*, pages 106–115. ACM, New York, USA, 2005.

[CHW00] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, Apr 2000.

[CLRS09] Thomas T. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.

[CM08] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1402085877, 9781402085871.

[CTLA12] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. Communication-aware mapping of kpn applications onto heterogeneous mpsocs. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 1266–1271. ACM, New York, USA, 2012.

[CW00] Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, pages 57–64. ACM, 2000.

[DG07] A. Douillet and G.R. Gao. Software-pipelining on multi-core architectures. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*, pages 39–48. IEEE Computer Society, 2007.

[Dig13] Digilent, Inc. *Genesys Board Reference Manual*, September 2013.

[ECR+10] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*, pages 89–100. IEEE Computer Society, Dec 2010.

[EHM+05] Jos van Eijndhoven, Jan Hoogerbrugge, Jayram M.N., Paul Stravers, and Andrei Terechko. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, chapter Cache-coherent heterogeneous multiprocessing as basis for streaming applications, pages 61–80. Springer Netherlands, 2005.

[FJ08] S. Fide and S. Jenks. Architecture optimizations for synchronization and communication on chip multiprocessors. In *IEEE International*

Symposium on Parallel and Distributed Processing (IPDPS'08), pages 1 –8. IEEE, April 2008.

[Giv06] T. Givargis. Zero cost indexing for improved processor cache performance. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(1):3–25, 2006.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52. ACM, New York, NY, USA, 2008.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162. ACM, 2006.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[Hoa78] C. a. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISBN 0131532715. ISSN 00010782.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.

[IKA99] Kentaro Inenaga, Shigeru Kusakabe, and Makoto Amamiya. Producer-consumer pipelining for structured-data in a fine-grain non-strict dataflow language on commodity machines. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'99)*, pages 77–86. IEEE Computer Society, Washington, DC, USA, 1999.

[Int11] Intel xeon processor e5-2640, 2011. URL `http://ark.intel.com/products/64591/Intel-Xeon-Processor-E5-2640-15M-Cache-2_50-GHz-7_20-GTs-Intel-QPI`.

[JA90] Reese B. Jones and Vicki H. Allan. Software pipelining: A comparison and improvement. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture (MICRO'23)*, pages 46–56. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[JB07] Benjamin James Braun. *Ehrhart theory for lattice polytopes.* PhD thesis, Washington University, USA, 2007.

[Jef93] Kevin Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice (SAC'93)*, pages 796–804. ACM, New York, NY, USA, 1993.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475. Stockholm, Sweden, Aug 1974.

[KC12] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 437–448. IEEE Computer Society, Washington, DC, USA, 2012.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[Ker88] Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

[KKK+09] D. Kim, K. Kim, J.-Y. Kim, S. Lee, and H.-J. Yoo. Memory-centric network-on-chip for power efficient execution of task-level pipeline on a multi-core processor. *IET Computers & Digital Techniques*, 3(5):513, 2009.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'81)*, pages 207–218. ACM, New York, USA, 1981.

[Knu14] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The.* Addison-Wesley Professional, 2014.

[KRD00] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign CODES 2000*, pages 13–17. ACM, San Diego, CA, USA, May 2000.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[LAUH+15] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. Discopop: A profiling tool to identify parallelization opportunities. In Christoph Niethammer, José Gracia, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2014*, pages 37–54. Springer International Publishing, 2015.

[LF13] M. Lattuada and F. Ferrandi. Modeling pipelined application with synchronous data flow graphs. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pages 49–55, July 2013.

[LKM11] Per Larsen, Sven Karlsson, and Jan Madsen. Expressing coarse-grain dependencies among tasks in shared memory programs. *IEEE Transactions on Industrial Informatics*, 7(4):652–660, 2011.

[LM87a] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[LM87b] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987.

[LW05] Bei Li and Pieter Van Der Wolf. TTL inter-task communication implementation on a shared-memory multiprocessor platform. In *16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2005)*, pages 1–8, 2005.

[MKH+13] T. Miyajima, T. Kuhara, T. Hanawa, H. Amano, and T. Boku. Task level pipelining with PEACH2: An FPGA switching fabric for high performance computing. In *International Conference on Field-Programmable Technology (FPT'13)*, pages 466–469, Dec 2013.

[MKH+14] Takaaki Miyajima, Takuya Kuhara, Toshihiro Hanawa, Hideharu Amano, and Taisuke Boku. Task Level Pipelining on Multiple Accelerators via FPGA Switch. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'14)*, pages 267–274. Acta Press, Calgary, Canada, 2014.

[MKTdK07] S. Meijer, B. Kienhuis, A. Turjan, and E. de Kock. A process splitting transformation for kahn process networks. In *Design, Automation Test in Europe Conference Exhibition(DATE '07)*, pages 1–6, April 2007.

[MKWS07] Sjoerd Meijer, Bart Kienhuis, Johan Walters, and David Snuijf. Automatic partitioning and mapping of stream-based applications onto the intel ixp network processor. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES'07)*, pages 23–30. ACM, April 2007.

[Moo65] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114 – 117, 1965.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[NAS15] NAS parallel benchmarks (NBP), 2015. URL `https://www.nas.nasa.gov/publications/npb.html`.

[NMSD09] Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed Deprettere. Realizing FIFO Communication when Mapping Kahn Process Networks onto the Cell. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 308–317. Springer, 2009.

[Nol04] Landon Curt Noll. Fowler/noll/vo (fnv) hash, 2004. URL `http://www.isthe.com/chongo/tech/comp/fnv/`.

[OPE05] *MPC7450 RISC Microprocessor Family Reference Manual*, 5 edition, 2005.

[OPE16] OpenCV: Open Source Computer Vision, 2016. URL `http://opencv.org/`.

[ovp16] Open Virtual Platforms (OVP), 2016. URL `http://www.ovpworld.org/`.

[Par14] Arash Partow. General hash function source code (c), 2014. URL `http://www.partow.net/programming/hashfunctions/`.

[PH05] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In Pierre Fraigniaud, editor, *Distributed Computing (LNCS)*, volume 3724, pages 108–121. Springer, 2005.

[RCD07] Rui Rodrigues, João M. P. Cardoso, and Pedro C. Diniz. A data-driven approach for pipelining sequences of data-dependent loops. In *Proc. 15th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pages 219–228, 2007.

[RKAP+12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[RVD10] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531–551, 2010. ISSN 01678191.

[SH01] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *International Symposium on VLSI Technology, Systems, and Applications. Proceedings of Technical Papers*, pages 184–187. Institute of Electrical & Electronics Engineers (IEEE), 2001.

[Smi82] Burton J. Smith. Architecture and applications of the hep multiprocessor computer system. *Real-Time signal processing IV*, 298:241–248, 1982.

[Smi86] B.J. Smith. A pipelined, shared resource mimd computer. In *Advanced computer architecture*, pages 39–41. IEEE Computer Society Press, 1986.

[Sni02] Greg Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proceedings of ACM 10th International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 177–186. ACM, New York, NY, USA, 2002.

[SPE15] Standard performance evaluation corporation (spec), 2015. URL `http://www.spec.org/`.

[SRI14] Alexandre Skyrme, Noemi Rodriguez, and Roberto Ierusalimschy. A survey of support for structured communication in concurrency control models. *Journal of Parallel and Distributed Computing*, 74(4):2266–2285, April 2014.

[Sut08] Herb Sutter. Lock-free code: A false sense of security. *Dr. Dobb's Journal*, 2008.

[SW11] R. Sedgewick and K. Wayne. *Algorithms in C*, volume 4th. Addison-Wesley Professional, 2011. ISBN 9780321573513.

[TCA07] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-3047-8.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, volume LNCS 2304, pages 179–196, 2002.

[TKD02] Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. A compile time based approach for solving out-of-order communication in kahn process networks. In *Proceedings of IEEE Int. Conference on Application-specific Systems (ASAP'02)*, pages 17–28, 2002.

[TKD03a] Alexandra Turjan, Bart Kienhuis, and Ed F Deprettere. Realizations of the extended linearization model. In *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, pages 171–190. CRC Press, 2003.

[TKD03b] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A technique to determine inter-process communication in the polyhedral model. In *Proceedings of 10th International Workshop on Compilers for Parallel Computers (CPC'03)*, pages 8–10. Amsterdam, The Netherlands, January 2003.

[TKD05] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Solving out-of-order communication in kahn process networks. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 40 (1):7–18, 2005.

[UTD98] UTDSP benchmark suite, May 1998. URL `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

[vec16] Pareon profile, 2016. URL `http://www.vectorfabrics.com/`.

[VJ04] Srinivas N. Vadlamani and Stephen F. Jenks. The synchronized pipelined parallelism model. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 16: 163–168, 2004.

[VJ07] S. Vadlamani and S. Jenks. Architectural considerations for efficient software execution on parallel microprocessors. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'07).*, pages 1–10, March 2007.

[WL06] M. Weinhardt and W. Luk. Pipeline vectorization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 20(2):234–248, November 2006.

[WTS+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30 (9):86–93, 1997.

[Xil10a] Xilinx, Inc. *EDK Concepts, Tools, and Techniques*, September 2010.

[Xil10b] Xilinx, Inc. *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)*, 2010.

[Xil10c] Xilinx, Inc. *MicroBlaze Processor Reference Guide v12.3*, 2010.

[Xil11a] Xilinx. Ise design suite, 2011. URL `http://www.xilinx.com/products/design-tools/ise-design-suite.html`.

[Xil11b] Xilinx, Inc. *LogiCORE IP Fast Simplex Link (FSL) V20 Bus v2.11c*, April 2011.

[Xil12] Xilinx, Inc. *Vivado Design Suite User Guide, High-Level Synthesis*, July 2012.

[Xil15a] Xilinx, Inc. *VC707 Evaluation Board for the Virtex-7 FPGA (User Guide)*, ug885 (v1.6.1) edition, September 2015. URL `http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html`.

[Xil15b] Xilinx, Inc. *Virtex-5 Family Overview*, August 2015.

[YXY+14] Zhiyi Yu, Ruijin Xiao, Kaidi You, Heng Quan, Peng Ou, Zheng Yu, Maofei He, Jiajie Zhang, Yan Ying, Haofan Yang, Jun Han, Xu Cheng, Zhang Zhang, Ming'e Jing, and Xiaoyang Zeng. A 16-core processor with shared-memory and message-passing communications. *IEEE Transactions on Circuits and Systems*, 61(4):1081–1094, April 2014.

[ZHD03] Heidi E. Ziegler, Mary W. Hall, and Pedro C. Diniz. Compiler-generated communication for pipelined fpga applications. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)*, pages 610–615. ACM, 2003.

[ZHX+15] Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong. Cmost: A system-level fpga compilation framework. In *Proceedings of the 52th Annual Design Automation Conference (DAC'15)*, pages 158:1–158:6. ACM, New York, USA, 2015.

[ZNS13] Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. Mapping of streaming applications considering alternative application specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):34:1–34:21, March 2013.

[ZSHD02] H. Ziegler, B. So, M. Hall, and P.C. Diniz. Coarse-grain pipelining on multiple FPGA architectures. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, pages 77–86. IEEE Computer Society, Washington, DC, USA, 2002.

APPENDIX A

Appendix

A.1 Implementing the ISB in Hardware

Although in Section 6.1 we have used a full softcore processor (MicroBlaze) to implement the Inter-Stage Buffer (ISB) schemes, the ISB can be also implemented as an *Intellectual Property (IP)* core using custom hardware. Implementing the ISB as an IP core may reduce the communication overhead and may decrease the FPGA resources needed. In addition, an ISB IP core would allow its use by hardware designers in the context of other target FPGA-based architectures, including the ones generated by HLS tools. In this appendix, we describe the implementation of an ISB IP core and its integration in a target architecture consisting of two MicroBlaze processors (one acting as producer and the other one acting as consumer). We also provide the preliminary experiment results and also evaluate the impact of using the ISB IP core on the performance for a number of benchmarks.

Figure A.1 shows a possible multicore target architecture considering an ISB scheme without access to external memory. In the architectures used herein, MicroBlaze 1 and MicroBlaze 2 are responsible for executing the codes for the producer and the consumer, respectively. Note that in the proposed architecture herein, the barrel shifter, the hardware multiplier, and the hardware divider are enabled and the MicroBlaze caches and the hardware floating point unit are disabled. The ISB IP core communicates with the two MicroBlaze processors directly through the Xilinx Fast Simplex Link (FSL) [Xil11b].

The ISB IP core was described in behavioral-RTL (Register Transfer Level) VHDL and was synthesized using Xilinx ISE 13.1 [Xil11a]. In the experiments presented herein, an ISB local buffer with the size of 1,024 elements was described in VHDL using a standard one-dimensional array of registers (32 bit).

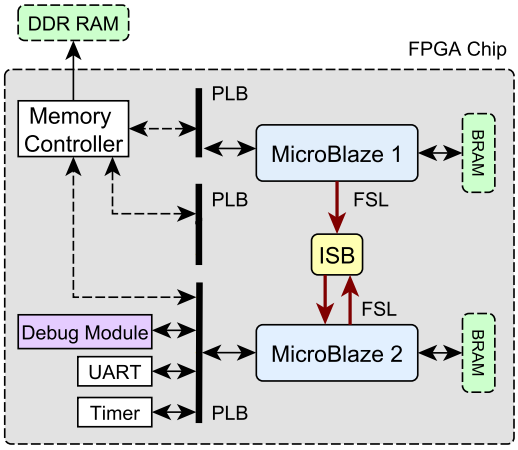


Figure A.1: FPGA prototype system block diagram with two MicroBlazes and including ISB as an IP-core.

We evaluate the impact of implementing the ISB in hardware on communication overhead, performance, and FPGA hardware resources. In order to evaluate the communication overhead between the MicroBlaze cores, we measured the number of clock cycles required to communicate a single data element from the producer's core to the consumer's core without storing in the local ISB memory for both architectures: (a) the original ISB scheme using MicroBlaze to implement the ISB; and (b) the proposed architecture considering the ISB as an IP core. Each measurement includes the request of data from the consumer, the sending of data from the consumer to the ISB, and the ISB verification if this the data requested is available in the local buffer and if so the sending of the data to the consumer. Thus, the number of clock cycles represent the cycles from the consumer request to the availability of data on the consumer side.

Table A.1 shows the overall number of clock cycles for both architectures. As shown, 176 clock cycles are required to communicate a single data element from the producer's core to the consumer's core in the architecture using the ISB IP core. In the original scheme of the ISB (implemented by a third MicroBlaze), the communication overhead is 190 clock cycles. This means that using an ISB IP core can

decrease 7.37% the number of clock cycles required for a single data communication between the cores compared with the original ISB scheme. When increasing the number of data elements communicating between cores (e.g., 64 data elements) and considering the use of the local ISB memory, the number of clock cycles in the proposed architecture compared to the ISB implemented in a MicroBlaze decreased by 95.89% (from 20,799 to 854 clock cycles), a considerable reduction. The reason is that in the original ISB scheme, reads/writes from/to the memory are over a single-port memory connected to the MicroBlaze and thus, the ISB cannot perform more than one memory read/write simultaneously. As in the hardware implementation of the ISB, we use a Dual-port memory (to implement the local table of the ISB), two read and write operations to different memory addresses can be performed simultaneously.

Table A.1: The overall data communication clock cycles between the producer's core and the consumer's core in the ISB scheme

ISB Scheme	**Clock Cycles (#CS)**	
	a single data element	64 data elements
Implemented in hardware	176	854
Implemented in MicroBlaze	190	20799

Note that our custom ISB IP core only uses local memory. By considering the size of input arrays in the benchmarks presented in the Table 6.3 and a local buffer with the size of 1,024 in the ISB IP core, the P/C pairs can communicate using only local (on-chip) memory and thus without the need to store data in the external memory. When considering the dataset sizes and the benchmarks presented in the Table 6.3, and also an ISB local buffer with the size of 1,024 in the ISB scheme implemented as a MicroBlaze, 100% of the P/C communicated data elements can be loaded/stored from/to the local memory for FDCT, FIR-Edge and Edge-Detection benchmarks. However, for Wavelet Transform and Gaussian blur benchmarks, 0.22% and 36.97% of data elements can be loaded/stored from/to the local memory, respectively. This means that for these benchmarks it might be more efficient to disable the access to external memory, as is the case of using the custom ISB IP core presented here. However, using an ISB IP core with only local memory may not be sufficient when

most data elements need to be store into the external memory and thus, the use of a level of external memory would need to be considered for a more generic ISB IP core.

In order to evaluate the impact of using the ISB IP core on the global performance of the benchmarks, we have used the ISB IP core in the context of TaLP. Figure A.2 shows the achieved speedups for the Wavelet Transform, FDCT, FIR-Edge, Edge-Detection, and Gaussian blur when using the ISB scheme implemented with three MicroBlaze processors with and without optimization techniques, and the proposed scheme using two MicroBlaze processors for the producer/consumer cores and the ISB IP core. In these experiments, we considered the ISB optimization schemes (scheme #3 and scheme #4) which achieved the highest achieved speedups presented in Table 6.7.

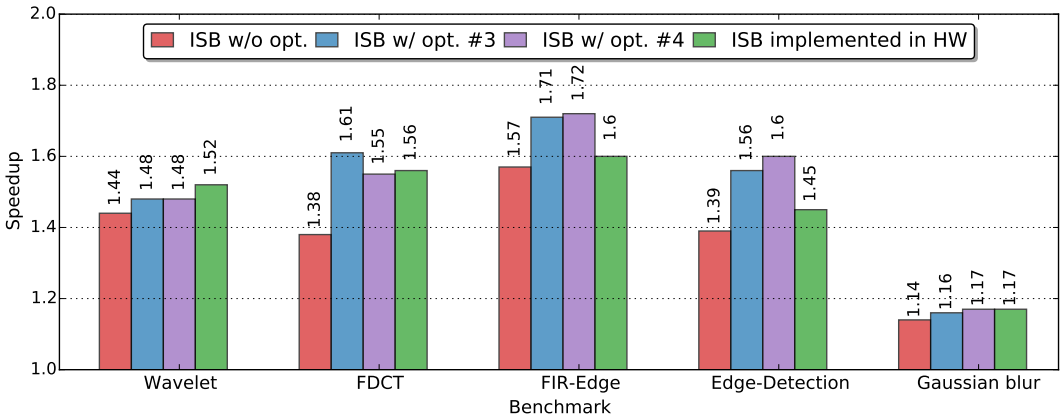


Figure A.2: Speedups when using the ISB IP-core using only local memory over the original ISB scheme without optimization and the ISB with optimizations (scheme #3 and scheme #4).

The results show that using the ISB IP core between producer/consumer cores without access to external memory are very similar to the achieved speedups when using ISB optimization schemes. For instance, for FDCT the achieved speedup when using the ISB IP core (1.56×) is almost equal to the speedup achieved when using the ISB with optimization scheme #4 (1.55×) and 96.7% close to the speedup achieved when using the ISB with optimization scheme #3 (1.61×). However, in the case of Wavelet Transform, the speedup increases 5.56% (from 1.44× to 1.52×) compared with the ISB without optimization, and increases 2.7% compared with the highest

achieved speedup (ISB with optimization schemes #3 and #4). For FIR-Edge, the speedup increased by 1.91% (from 1.57× to 1.60×) when using the ISB IP core and compared with the ISB without optimization. The speedups achieved when using the ISB IP core for FIR-Edge is 93% close to the speedup obtained when using the ISB with optimization scheme #3 and scheme #4. In addition, for Gaussian blur, the speedup achieved when using the ISB IP core increased by 2.63% (from 1.14× to 1.17×) compared with the ISB without optimization and equals the speedup when using the ISB with optimization schemes.

With respect to the theoretical uppderbound A of the benchmarks (see Table 6.4 in Section 6.3), the achieved speedups when using the ISB IP core for the Wavelet Transform (1.52×), FDCT (1.56×), FIR-Edge (1.60×), Edge-Detection (1.45×), and Gaussian blur (1.17×) are 76.38% for Wavelet Transform, 96.89% for FDCT, 93.02% for FIR-Edge, 75.92% for Edge-Detection and 100% for the Gaussian blur, close to their theoretical speedup.

In order to analyze the overhead in terms of hardware resources to implement our custom ISB IP core, we compare the hardware resources when using a MicroBlaze to implement the ISB with the custom ISB IP core separately. Table A.2 presents the hardware resources used for these architectures when using a Xilinx Virtex-5 LX50T FPGA [Xil15b]. Note that in the experiment herein, we have used the same configuration (A) for the MicroBlaze to implement the ISB using the barrel shifter, the hardware multiplier, and the hardware divider and without using MicroBlaze caches and the hardware floating point unit. However, in Table 2, we also considered another configuration (B). In this configuration we considered a MicroBlaze without the barrel shifter, the hardware multiplier, and the hardware divider. As shown, the configuration B has a reduction of 6.45% of the number of slice registers (from 5,227 to 4,890), 10.8% of the number of slice LUTs (from 4,489 to 4,004) and also 15.86% of the number of occupied slices (from 2,805 to 2,360) compared with the ISB implemented using one MicroBlaze with the configuration A.

As shown we achieve a considerable reduction of all FPGA resources when using the ISB IP core compared to the ISB implemented using one MicroBlaze in both configurations A and B. For instance, the total memory used has a reduction of

Table A.2: FPGA hardware resources usage for the ISB implemented using one MicroBlaze and the ISB as an IP core.

Device Utilization	ISB implemented using one MicroBlaze (A)		ISB implemented using one MicroBlaze (B)		ISB (IP Core)	
	Used	Usage Percentage	Used	Usage Percentage	Used	Usage Percentage
Number of Slice Registers	5,227	18%	4,890	16%	114	1%
Number of Slice LUTs	4,489	15%	4,004	13%	1,929	6%
Number of occupied Slices	2,805	38%	2,360	32%	589	8%
Total Memory used (KB)	900	41%	900	41%	18	1%
Number of DSP48Es	3	6%	3	6%	0	0%

98% (from 900 KB to 18 KB). In addition, the number of slice registers and LUTs decreased by 97.82% (from 5227 to 114) and 57.03% (from $4,489$ to $1,929$). In addition, the number of DSP48Es has a reduction of 100% (from 3 to 0) when considering our custom IP core versus the use of a MicroBlaze.

In addition, Table A.3 presents the hardware resources used for the following two architectures: (a) the system with two MicroBlaze (P/C) processors + the ISB (MicroBlaze); and (b) the system with two MicroBlaze processors + ISB (IP core). The MicroBlaze configurations in the system with two MicroBlaze (P/C) processors and the ISB implemented with one MicroBlaze are the same as configuration A. As shown, the total memory used in the system using the ISB IP core is decreased by 34.21% (from $2,052$ to $1,350$). The number of slice registers and the number of DSP48Es in the system with two MicroBlaze cores and the ISB IP core are also reduced by 20.98% (from $10,161$ to $8,029$ and by 33.3% (from 9 to 6), respectively. However, the size of slice LUTs and the number of occupied slices are increased by 8.8% (from 9,868 to 10,820) and 3.39% (from 4,760 to 4,927) compared with the system with two MicroBlaze (P/C) + ISB (MicroBlaze Scheme).

In general, we can conclude that the system with two MicroBlaze cores and the ISB IP core with only local memory uses less memory, less number of slices registers and DSP48Es and almost equal number of slice LUTs and occupied slices compared with the ISB scheme implemented with three MicroBlaze (P/C and the ISB). Note that the maximum clock frequency of the ISB IP core and the MicroBlaze processor for the FPGA used in the experiment are 200 MHZ and 125 MHz.

Table A.3: FPGA hardware resources usage for each systems used to implement the ISB scheme: the system with two MicroBlaze (P/C) + the ISB (MicroBlaze); and the system with two MicroBlaze + ISB (IP core) (MB: MicroBlaze).

Device Utilization	System with two MicroBlaze (P/C) + ISB (MB)		System with two MB + ISB (IP core)	
	Used	Usage Percentage	Used	Usage Percentage
Number of Slice Registers	10,161	35%	8,029	27%
Number of Slice LUTs	9,868	34%	10,820	37%
Number of occupied Slices	4,760	66%	4,927	68%
Total Memory used (KB)	2,052	95%	1,350	62%
Number of DSP48Es	9	18%	6	12%

A.2 ISB with Two Tables

In the scheme presented in Section 4.1.1, we have used only one local table in the ISB. However, the ISB can use multiple tables for loading/storing data locally. The main idea of using the multiple tables in the ISB is to increase the number of local (on-chip) memory accesses and thus reducing the number of accesses to the external memory. Multiple ISB tables provide the opportunity to store locally data for which the hash function returns the same index and the associated position in at least one table is already full.

Figure A.3 shows the ISB scheme using two local tables with hashing and an empty/full flag. This ISB has the following behavior: the ISB reads the index from the producer and maps it into the buffer position of Table 1 which is calculated by the hash function. If the flag of the mapped position in *Table 1* is set to 0 (*empty*), the ISB stores the index and data in *Table 1* and sets the flag of the mapped position to 1 (*full*). If storing the index into the first local table fails (the mapped position is *full*, i.e., is already used), the ISB tries the second local table (*Table 2*) and maps the produced index into the same mapped position of *Table 2* determined by the same hash function. If neither *Table 1* nor *Table 2* are available to store the data locally, the ISB stores the data into the external memory. Also, the ISB reads the requested index from the consumer and maps the index to the buffer position of *Table 1*. If the flag of the mapped position in *Table 1* is set to 1 (*full*), the ISB loads

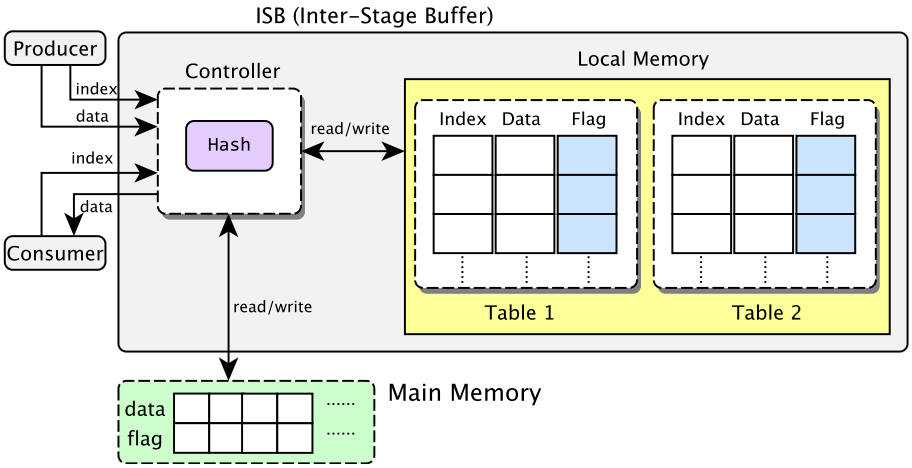


Figure A.3: ISB using two local tables with an empty/full bit flag.

the data and resets the flag to 0 (*empty*). If the requested index is not available in *Table 1*, the ISB tries the mapped position in *Table 2*, and only if data is not there it inspects the external memory.

Table A.4 shows the speedups obtained when using the ISB with one and two local tables as buffers and implemented using a MicroBlaze. In these experiments, the size of each local table is 1,024. As shown, for Gray-Histogram, the achieved speedup using two tables is the same than the achieved speedup in the original ISB scheme using only one local table. However, in the case of FDCT, the speedup increases 15.9% (from 1.38× to 1.60×) when using two local tables and almost equals the theoretical upperbound A (1.61×) for the FDCT.

Table A.4: Speedups achieved when considering an ISB scheme using one and two local tables and implemented using a MicroBlaze core vs. a single core baseline architecture.

Benchmark	ISB w/ One Table	ISB w/ Two Tables	Upperbound A
Gray-Hisogram	1.65	1.65	1.86
Fast DCT (FDCT)	1.38	1.60	1.61
Wavelet Transform	1.46	1.47	1.99

The number of data elements stored externally with one and two local tables is 0. This means that for FDCT, there is no need to the external memory and all data elements can be stored locally. In FDCT, the number of hits to each memory address of the ISB local buffer with the size of 1,024 is 469 times. Thus by considering dual

local buffer for the ISB, the number of hits to the same mapped address of local buffer may reduces and as a result it can improves the performance.

With respect to the Wavelet Transform, the performance improvement is only 0.7% (from 1.46× to 1.47×). In Wavelet Transform, 99.7% of all data elements are loaded/stored from/into the external memory and the maximum number of hits to each local memory address of the ISB local buffer is 2. Thus, the use of two tables does not have the potential to contribute to a significantly higher number of data elements stored locally.

One of the possible solutions to increase the number of data elements stored locally in multiple tables can be the use of a specific hash function for each table.

About the Author

Ali Azarian received his B.Eng and Master degree in computer engineering both from Azad University, Iran in 2002 and 2007, respectively. In 2010 he started his Ph.D. in Informatics Engineering at the Faculty of Engineering of the University of Porto (FEUP), Portugal and became a researcher at INESC TEC Porto. He visited the Karlsruhe Institute of Technology (KIT), Germany, twice and worked in the group of Prof. Jürgen Becker. Ali's research interests include Parallel Computing, Field-Programmable Custom Computing Machines (FCCMs), Reconfigurable Computing and Embedded System Design. He is a student member of IEEE, ACM and HiPEAC.

Publications Related to the PhD:

1. **Ali Azarian**, João M. P. Cardoso, *"Pipelining Data-Dependent Tasks in FPGA-based Multicore Architectures"*, in the Journal of Microprocessors and Microsystems (MICPRO-Elsevier), Vol. 42, pp. 165-179, 2016.
2. **Ali Azarian**, João M. P. Cardoso, *"Reducing Misses to External Memory Accesses in Task-Level Pipelining,"* in IEEE International Symposium on Circuits and Systems (ISCAS'15), pp. 1422-1425, Lisbon, Portugal, 2015.
3. **Ali Azarian**, João M. P. Cardoso, *"Coarse/Fine-grained Approaches for Pipelining Computing Stages in FPGA-based Multicore Architectures,"* in 3th Workshop on On-chip memory hierarchies and interconnects: organization, manage-

ment and implementation (OMHI 2014), workshop co-located with the 20th International Conference of Parallel Processing (Euro-Par 2014), pp. 266-278, Springer LNCS 8806, 2014.

4. **Ali Azarian**, João M. P. Cardoso, *"An FPGA-based Fine-grained Data Synchronization for Pipelining Computing Stages,"* In X Jornadas sobre Sistemas Reconfiguráveis (REC 2014), pp. 57-60, Vilamoura, Portugal, April, 2014.
5. **Ali Azarian**, João M. P. Cardoso, Stephan Werner, Jürgen Becker, *"An FPGA-based Multi-Core Approach for Pipelining Computing Stages,"* In 28th Symposium On Applied Computing (SAC'13), pp. 1533-1540, Coimbra, Portugal, March 18-22, 2013.
6. **Ali Azarian**, *"Pipelining Computing Stages in Configurable Multicore Architectures,"* In 23th International Conference on Field Programmable Logic and Applications (FPL'13), pp. 2-4, Porto, Portugal, 2013.
7. **Ali Azarian** and João M. P. Cardoso, *"Pipelining Producer-Consumer Tasks using Custom Multi-Core Architectures,"* In Proceedings of the 7th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2011), 10-16 July, 2011, Fiuggi, Italy [Poster Abstracts]

Other Publications during the PhD Work:

1. **Ali Azarian**, João Canas Ferreira, Stephan Werner, Zlatko Petrov, João M. P. Cardoso, Michael Huebner, *"Analysis of Error Detection Schemes: Toolchain Support and Hardware/Software Implications,"* In NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2012), pp. 62-69, Nuremberg, Germany, June 25-28, 2012.