Title
Accelerator Synthesis and Integration for CPU+FPGA Systems

Author
Cheng, Shaoyi

Publication Date
2016

Peer reviewed|Thesis/dissertation

Accelerator Synthesis and Integration for CPU+FPGA Systems

by

Shaoyi Cheng

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Vladimir Stojanovic
Professor Alper Atamturk

Fall 2016

Accelerator Synthesis and Integration for CPU+FPGA Systems

Abstract

Accelerator Synthesis and Integration for CPU+FPGA Systems

by

Shaoyi Cheng

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

As the scaling down of transistor size no longer provides boost to processor clock frequency, there has been a move towards parallel computers and more recently, heterogeneous computing platforms. To target the FPGA component in these systems, high-level synthesis (HLS) tools were developed to facilitate hardware generation from higher level algorithmic descriptions. Despite being an effective method for rapid hardware generation, in the context of offloading compute intensive software kernels to FPGA accelerators, current HLS tools do not always take full advantage of the hardware platforms. Processor centric software implementations often have to be rewritten if good quality of results is desired.

In this work, we present a framework to refactor and restructure compute intensive software kernels, making them better suited for FPGA platforms. An algorithm was proposed to decouple memory operations and computation, generating accelerator pipelines composed of independent modules connected through FIFO channels. These decoupled computational pipelines have much better throughput due to their efficient use of the memory bandwidth and improved tolerance towards data access latency. Our methodology complements existing work in high-level synthesis and facilitates the creation of heterogeneous systems with high performance accelerators and general purpose processors. With our approach, for a set of non-regular algorithm kernels written in C, a performance improvement of 3.3 to 9.1x is observed over direct C-to-Hardware mapping using a state-of-the-art HLS tool.

To ensure the absence of artificial deadlocks in the pipelines generated by our framework, we also formulated an analysis scheme examining various dependencies between operations distributed across different pipeline modules. The interactions between the modules' schedules, the capacity of the communication channels and the memory access mechanisms are all incorporated into our model, such that potential artificial deadlocks can be detected and resolved *a priori*. The applicability of our technique is not limited to the computational pipeline generated by our algorithm, but also other networks of communicating processes assuming their interaction with the channels follows a set of simple rules.

To push the limit in usability of FPGA platforms, we also explored the generation and integration of accelerators using only program binaries and execution profiles. Assuming no

user input, the approach is only applied to more regular applications, where the memory access patterns are analyzable and coarse grained parallelism can be extracted. A run time mechanism is also devised to ensure the correctness of the parallelization performed during accelerator synthesis. With the help of binary instrumentation tools, it becomes possible to integrate the FPGA-accelerated parts into the original application in a user transparent way. Neither recompilation of the original program nor the source code is required. This approach is applied to a few benchmarks for which decoupled computational pipelines are synthesized. With memory level and coarse grained parallelization, significant improvement in performance (3.7 to 9x) over general purpose processor was observed, despite the FPGA running at a fraction of the CPU's clock frequency. The run time checking mechanism was also shown to only incur small overhead, especially for loop nests with large number of iterations.

To Zhanglei Cheng and Peizhen Lan,
my amazing parents,
whose sacrifice and love made it possible for me to complete this work

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the support and help from the people I have worked with over the past seven years. My experience in U.C. Berkeley has been a memorable one, thanks to these wonderful individuals.

Foremost among them is my advisor, Professor John Wawrzynek, who has been a great mentor and role model for me. He has given me the freedom to explore many interesting ideas and encouraged me throughout my graduate study. I am indebted to him for his support and advice. I would also like to thank my dissertation committee, Professor Krste Asanovic, Vladimir Stojanovic and Alper Atamturk, who have inspired and motivated the theme of this research effort.

Special thanks to Dr. Mingjie Lin, who I worked with when I just came to Berkeley, for teaching me many things about graduate school and how to be a successful researcher. I would also like to acknowledge Ilia Liebedev and Chris Fletcher, with whom I have collaborated in some of my early projects in Berkeley, for the wonderful conversations and helping hands. I am also grateful to my labmates James Martin, Simon Scott and Chris Yarp for all the interesting discussions and exchange of ideas.

The wonderful staff in Berkeley Wireless Research Center has also given me big help in completing my study in Berkeley. The infrastructure used in my experiments is supported by Brian Richards, who is always there when I encounter problems in using various software tools. Thanks to Olivia Nolan, Leslie Nishiyama, Sarah Jordan, Columba Candy Corpus and Yessica Bravo, for their effort in making BWRC such a great place to work.

Chapter 1

Introduction

1.1 Trend in General Purpose Computing Platforms

The downscaling of transistor size, as first observed by Moore [1], massively increased the capability of VLSI circuits for more than three decades. The number of transistors on a single silicon die doubles approximately every 18 months while the supply voltage and transistor channel length scaled down according to the prediction by Dennard [2]. This advancement in technology not only enabled more integration, it also kept the power density of the new chip the same as its predecessors, taking advantage of the voltage scaling. Bigger caches and various architectural features were implemented using the newly available transistors, which improved single thread performance. At the same time, the voltage scaling also allows for increased clock frequency of processors under the same power budget. Intel 80386 chips released in the 1980s operated at 25 MHz while the Pentium D, released in the middle of the last decade, reached a clock frequency of more than 3.5 GHz. This 140x boost in frequency, coupled with greater instruction issue width, more accurate branch predictor and many other micro-architectural innovations unlocked huge performance increase for software applications during this period. No change in programming methodology was needed to harness the ever increasing power of the new processors. Unfortunately, the leakage current quickly grows as the threshold voltage of transistor decreases. Eventually, further scaling of the supply voltage was no longer possible and with that, the clock rate of the mainstream processors stagnated at approximately 3.8 GHz.

As the growth of processor clock frequency came to a stop in the middle of the last decade, the performance boost provided by the continuous advancement of hardware technology for sequential programs can no longer be sustained. The benefits of user transparent architecture optimizations are also diminishing. As parallel computers and more recently, heterogeneous computing platforms become mainstream, the software developers are finally faced with the challenge of matching their computation specifications with the characteristics of the underlying compute substrate.

During the rise of parallel computer systems, the research community created many

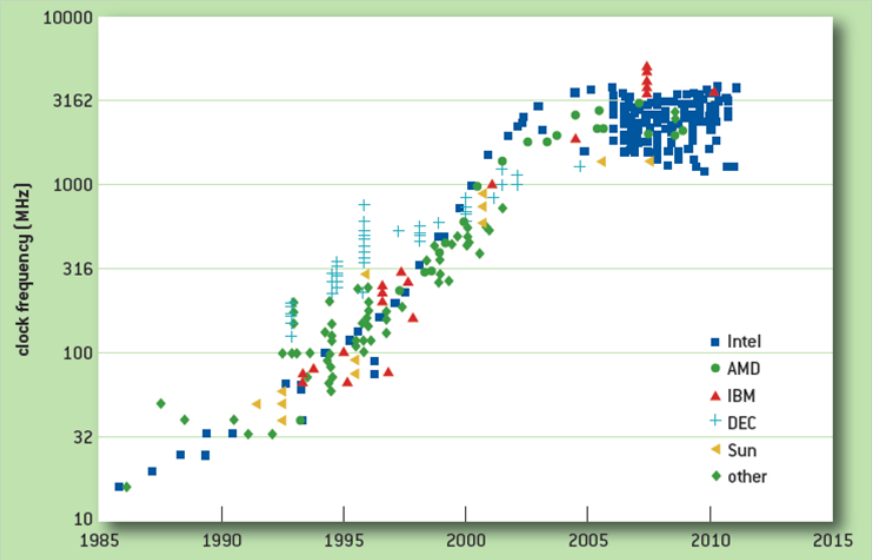


Figure 1.1: Processor Frequency Scaling Over Time [3]

auto parallelization tools [4] [5] [6], to relieve the programmers from the tedious manual parallelization process. As real applications often contain irregular algorithms which are hard to analyze, various mechanisms were also proposed for users to provide hints/interactively guide the parallelizing compiler [7] [8] [9]. A very similar trend, is currently happening in the heterogeneous computing realm as well.

1.2 Heterogeneous Computing with Field Programmable Gate Arrays

Field programmable gate array (FPGA) is a fundamentally different computation device compared to the conventional microprocessors. Figure 1.2 shows a simple FPGA which contains a matrix of logic blocks connected by programmable interconnect. Each logic block can be configured to perform different compute operations while the interconnect links them together to form a fixed function hardware engine. Compare to application specific integrated circuits (ASICs), the FPGA configurations can be easily changed to implement many different functions, accommodating various needs of the users. This flexibility, of course, comes at a cost. As the logic and routing resources are all reprogrammable mechanisms supported by RAM bits and switches, there are significant area, performance and power overheads compared to the ASICs [10]. To alleviate these disadvantages, more specialized circuits such as block RAM and DSP blocks are added to modern FPGAs [11] [12]. Subsequently some of the common functions can be mapped onto these blocks and get closer to ASIC performance/power.

When an application is implemented on an FPGA, arithmetic and logic operations are spatially mapped to the various compute resources in the reconfigurable array, while the dependencies between them are satisfied by physical connections formed with on-chip routing resources. Compare to the CPUs, where the ALUs are heavily time multiplexed, FPGAs can perform orders of magnitude more operations every clock cycle due to its massive parallelism. Many previous research projects [13] [14] [15] have demonstrated the advantage of FPGAs in performance and energy efficiency v.s. other programmable platforms. For the right kind of computation, FPGAs offer an attractive trade-off between flexibility and efficiency compared to ASICs and CPUs, and thus occupy a unique space in the spectrum of computing devices.

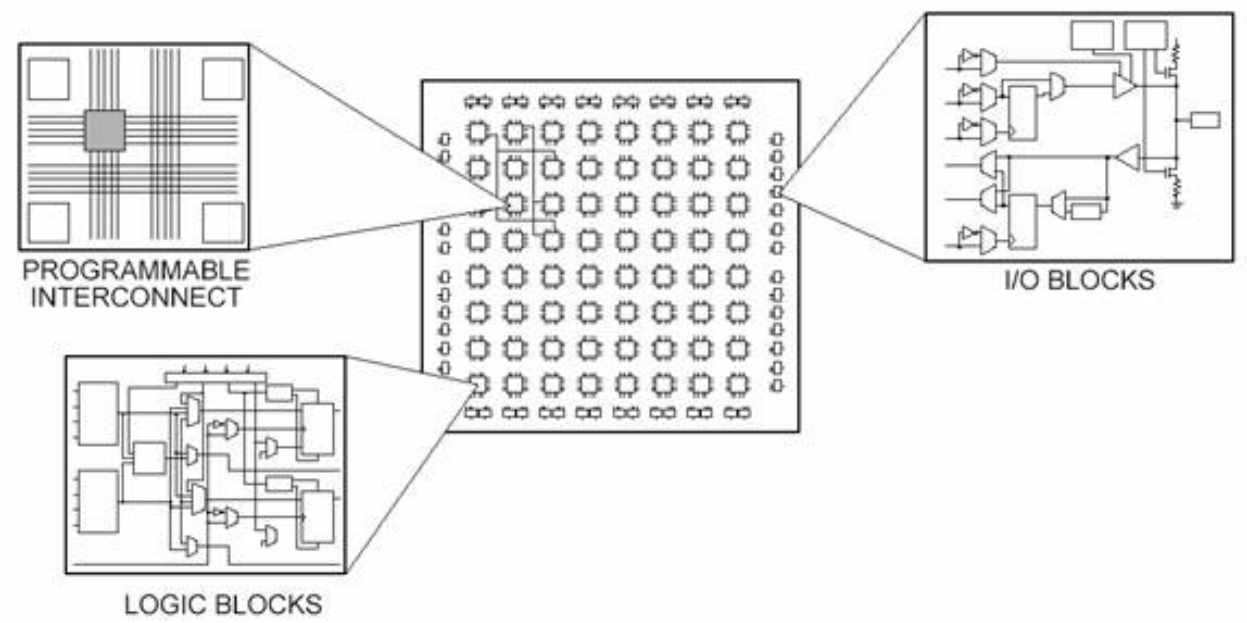


Figure 1.2: Simplified FPGA Architecture [16]

When CPUs are combined with FPGAs, the resulted heterogeneous platforms would allow the users to place workloads onto the most appropriate computing device to increase overall system efficiency. Inherently serial functions may not be able to make use of the FPGAs' abundance of computing resources and can have better speed on the CPUs, which usually run at much higher clock frequencies. Meanwhile, functions with plenty of parallelism can take advantage of the spatial computing paradigm provided by the FPGAs. Application specific accelerators implementing these functions can offer huge performance and energy advantages over general purpose processors. There are many systems where FPGAs are coupled to CPUs through PCIe interconnect, Front Side Bus etc. [17] [18], used in application domains ranging from scientific computing, gas and oil exploration to financial analytics [19] [20] [21]. Also, recent developments in FPGA SoCs, where the reconfigurable arrays are integrated with hard processors and memory interface IPs at the chip level, have created

some highly compact yet versatile computing platforms [22] [23]. However, to make these platforms easily accessible to the application developers, there are still many challenges, as the programming methodology for FPGAs is rather different than that for the general purpose processors.

Traditionally, FPGAs are programmed using register transfer level (RTL) design abstraction. The designers not only have to extract both coarse-grain and fine-grain parallelism in the application, but also need to define the behavior of the final implementation down to every clock cycle. Hardware design principles such as clock management, state machines, pipelining, and device specific memory management must be employed to really unlock the potential in FPGA computing. All these concepts are unfortunately outside the expertise of most application-oriented software developers.

1.3 High Level Synthesis

Given the difficulty in programming FPGAs, there has been a trend towards design synthesis from higher levels of specifications [24], [25]. Being more compact and expressive, high level languages, when used as design input, can greatly increase the productivity of engineers. Just like in the case of parallelizing compilers, the gap between an efficient implementation and a productive programming experience attracted major interest from both industry and academia. Many commercial, [26], [27] [28], [29] and open source [30], [31] tools have been developed over the years to tackle this challenge of generating hardware functional blocks from high level behavioral descriptions. Programming languages such as C/C++, designed for processor-centric execution, are used by these high-level synthesis programs as the medium for input specification.

All these HLS tools attempt to capture parallelism in the control dataflow described by high level languages, often with various forms of guidance provided by the user, and generate hardware in the form of RTL. Compute operations and memory accesses are scheduled according to the dependency constraints extracted from analysis and resource constraints imposed by the target platform. The circuits generated usually follow a Finite State Machine with Datapath (FSMD) paradigm. Activation of a particular operator in the datapath is associated with a certain clock cycle and the execution of the entire control dataflow graph is orchestrated by a synthesized central controller. Depending on the system architecture, the generated hardware can have different mechanisms for accessing the memory. In some systems, direct memory access (DMA) engines are instantiated and data movements are explicitly managed, while in others, the generated hardware is presented with a memory interface rather similar to that used by a processor, with various caching schemes [32] [33] proposed to complement the datapath.

With the HLS tools, it seems the programmers now have an easy path to offloading the compute intensive parts of their software to FPGAs, but the performance boost and energy efficiency of the mapped implementations are often less than ideal. To produce good FPGA designs with HLS, the users still have to visualize and create hardware descriptions,

albeit with the C/C++ syntax. The parallelism in the application still needs to be identified and communicated to the tool with the addition of pragmas and directives. Furthermore, the hardware infrastructure for data movement into and out of on chip buffers also need to be created and managed explicitly. In most scenarios, there is still a trade-off between ease of use and the achieved quality of results. It is thus important to provide methods in parallelism discovery, accelerator optimization and system integration to help the user better take advantage of this unique compute substrate.

1.4 Dissertation Organization

In this dissertation, we present tools and mechanisms for transforming, optimizing and integrating FPGA accelerators using high-level synthesis approach.

Synthesis of computational pipelines with decoupled memory access

The first flow we created transforms a sequential program into a computational pipeline comprising multiple processing stages connected by communication channels. FPGAs are especially efficient in implementing streaming applications where data moves through a cascade of pipeline stages. This is unfortunately not how a typical C/C++ program describes its computation. By utilizing pipeline parallelism in sequential programs, we generate elastic pipelines with much better tolerance towards data access latency. For programs with a memory footprint greater than the on-chip RAM capacity, significant improvement in performance can be achieved. Within the same framework, conventional memory accesses are converted to a streaming model whenever possible, with customized caching and burst accesses to further boost the accelerator efficiency. In chapter 3, the implementation details of this flow are presented. To demonstrate the advantage of our approach, experimental results comparing it against state-of-the-art HLS tools are also presented.

Deadlock prevention in network of statically scheduled accelerators

In synthesizing the decoupled computational pipeline, the original program is essentially converted to a network of processes, each executed by a statically scheduled accelerator. In chapter 4, we create a framework to analyze the property of this network, leveraging past research in Kahn process networks, to determine conditions for liveness of our generated pipeline. Contrary to past simulation-based approaches, we examine the interaction between the static scheduling by HLS and the sizing of the communication channels between the processes to find ways to prevent deadlock *a priori*. We also discuss how our technique may be used in other more generalized contexts.

Accelerator generation and integration using program binaries

To push the limit in usability of FPGA platforms, we explored the generation and integration of accelerators with just program binaries and execution profiles. This approach is only beneficial for more regular applications, where the memory access patterns are analyzable and coarse grained paralellism can be extracted. With program binary analysis and instrumentation, it becomes possible to integrate the FPGA-accelerated parts into the original application in a user transparent way. Neither recompilation of the original program nor the source code is required. Performing all the analysis based on program binaries can introduce more uncertainty than a source code based approach. Thus we have devised a two phase mechanism to have parts of the analysis done during the runtime to ensure correctness. The details are described in chapter 5.

1.5 Thesis Contributions

In summary, the main contributions of my thesis include the following:

- Advancing the state of the art for high level synthesis by devising a systematic method to generate elastic processing pipelines from sequential programs. The pipeline parallelism in the source code is effectively harnessed to create higher performance accelerators.
- Creating an analysis framework for the generated pipeline, which is formulated as a process network. By examining the interaction between scheduling and buffer sizing in this network, we can perform the detection and resolution of deadlocks statically.
- Improving the usability of FPGAs by creating a flow which synthesize accelerators from program binaries. Various aspects of integrating the accelerators with the original program are also discussed. It opens up the possibility of offloading certain types of computation to FPGAs in a user-transparent fashion.
- Quantifying the benefits of the proposed flows using off-the-shelf FPGA SoCs. The performance and area consumption for the benchmarks are presented. Some design space exploration is also performed to demonstrate the trade-off between execution time and resource usage.

Chapter 2

Background and Related Work

In section 2.1, we first briefly discuss the history of high level synthesis and its most recent development. As current generation of HLS tools are heavily leveraging technologies developed in compilers for parallel computers, section 2.2 describes some of the most relevant work in this area. Section 2.3 looks at previous works in integrating conventional processors and reconfigurable fabrics, which provide the target substrate for any software + accelerator compilation flow. Finally, parallelization based on program binaries (section 2.4) is another research area which overlaps with our effort, though the existing work mostly focused on creation of parallel executable for multicore/vector machines.

2.1 Development of High Level Synthesis

The synthesis of hardware accelerators from higher-level specifications has been attracting interest from the CAD research community since the 1970s [34] [35]. The very early works focused on synthesis and simulation at both RTL and algorithmic level, but had little impact outside of academia as they predate the emergence of the EDA software industry. During the 80s and 90s, many fundamental concepts in HLS were explored. The main steps in HLS: allocation, scheduling and binding were all being investigated and influential papers and books were published, laying the foundation for the field [36][37][38][39][40][41][42][43][44]. Meanwhile, for design input, domain specific languages were often used, especially in the DSP-oriented projects [45][46][47]. In terms of commercialization of the research, in an era where RTL synthesis was just getting acceptance, the notion that HLS could further improve productivity using an unfamiliar input language did not appeal to most ASIC designers. As major EDA software vendors ventured into this space [48][49], however, HLS tools started to draw more attention by the early 2000s. But as the input languages for these tools were behavioral HDLs, and the quality of results were not superior to synthesized RTL, they did not achieve much success among the RTL designers.

The most recent generation of HLS tools finally made an attempt to target algorithm and software designers, by adopting C/C++ as the medium for design input [50][51][52].

The rise of FPGAs also offered a perfect target for high level synthesis. The time to market advantage of FPGAs is nicely enhanced with a fast algorithm to hardware mapping flow. The quality of results was also boosted as the newer tools leveraged progress in compiler optimizations and parallel computing. Consequently, we have seen wider adoption of HLS in recent years [53]. On the other hand, the usage model of modern HLS still assumes a certain level of hardware-awareness in the users, be it in the form of following the right coding style [54] or knowing where and when to insert optimization pragmas. Independent studies also suggested the necessity of FPGA expertise in the development of efficient end-to-end solutions [55][56]. There is certainly room for further improvement as the gap between a truly high level design specification and efficient hardware implementation still exists.

2.2 Compilation Techniques in High Level Synthesis

As mentioned in section 1.2, the performance advantage of FPGA accelerators comes from the massive parallelism exploited when computations are spatially mapped. The newest generation of HLS tools has been evolving concurrently with the wide adoption of parallel computers, many of the techniques proposed for parallelizing compilers were borrowed and applied in HLS.

Since the targeted platforms can vary for automatic parallelization, parallelisms of different granularities need to be identified to match of characteristics of the machines (e.g. data-level parallelism for SIMD machines, thread-level parallelism for multicores etc.). All these can be potentially taken advantage of by FPGAs due to their flexibility.

When compiling algorithms written in high level languages to FPGAs, it is possible to exploit instruction level parallelism (ILP) within basic blocks of the program without applying any transformation. Due to the limit of ILP within basic blocks [57], however, the speed up is not enough to provide a significant advantage versus a general purpose processor where the clock frequency can be a lot higher than a typical FPGA. To improve the quality of results in FPGA accelerators, execution of different iterations of loops are often overlapped. This is often performed using software pipelining techniques, proposed for very long instruction word (VLIW) processors [58][59][60][61][62]. The HLS tools start new iterations before previous ones are completed, and the minimum interval with which a new iteration can be initiated is dictated by the latency of the longest circular dependence in the control dataflow graph. This initiation interval (II) ultimately bounds the overall throughput achievable by the accelerators. Other loop transformations like loop unrolling and splitting are sometimes used to complement the loop pipelining performed during HLS. Incurring higher area overhead, they improve the performance by either decreasing the total number of iterations, each with more intra-iteration parallelism, or by simplifying the logic in the critical cycle of dependencies. In commercial HLS tools, these techniques are often applied through the use of pragmas by the users. Thus to find out where to apply these techniques often involves interactive design space exploration using the particular tool.

In certain application domains where memory access patterns can be statically analyzed,

concise mathematical models can be used to represent data dependencies and mathematical transformation [63][64][65][66][67][68]. These techniques were employed in program compilation targeting parallel machines of different kinds[69][70][71], and are now used by the HLS community [72][73][74].

Another body of work explored transformations from imperative descriptions in high level programs to alternative models of computation (MoC), before mapping to hardware accelerators [75][76]. MoCs like Kahn process networks are more suitable for implementation in hardware due to the explicit parallelism laid out in the model. These methodologies are most effective when applied to regular computation kernels like those in DSP applications. There are also some unaddressed issues such as how to close the gap between the memory model of the new MoC and that of a general purpose processor, which is essential in the context of generating closely coupled CPU+accelerator systems. Other tools requires the user to directly construct application in more parallel description of computation to facilitate hardware generation [77][78][79].

An important aspect of the HLS-generated accelerators lies in their interaction with the data stored in the memory. It is a common design practice to have data explicitly moved in and out of chip through instantiation of DMA engines [80], and coordinate the computation with transfer of data using software. There are also tools [81] which hide the complexity of managing memory hierarchy from the user by providing a standard abstraction of data access interfaces. The FPGA designers can use the provided primitives to explicitly manage data movement between the on-board memory and the SRAM on-chip, allowing locally-addressed memory accesses by the computation pipeline. A set of regular kernels are synthesized to this architecture in [82], demonstrating its applicability in the context of high level synthesis.

2.3 Hardware Platforms with Reconfigurable Components

In terms of the actual physical substrates containing reconfigurable elements, there were many different architectures proposed over the years. As the programmable logic portions of the systems vary in size, the models for their use also differ. Several platforms use programmable logic as function units [83] [84] [85] [86], tightly integrated with the processor pipeline. These reconfigurable functional units (RFUs) are given access to the internal states of the processors and can be used to speed up fine grained tasks. The proximity of the programmable logic to the CPU also allows for strong interaction between the accelerated parts of the computation and the software execution flow. To better take advantage of the enhanced architecture, compilers [87] were constructed to map operations expressed in high level languages to configurations for the RFUs.

In contrast with those highly integrated architectures, some platforms have loosely coupled reconfigurable arrays, used in co-processor mode [88] [89] [90]. To effectively leverage the capabilities of the programmable arrays, computation-intensive coarse grained tasks need

to be extracted and offloaded. Even further along this direction of loose coupling, many commercial reconfigurable devices are actually used in stand-alone FPGA boards [91] [92]. Their interaction with the host CPU is even more costly but the capacity of the arrays themselves are usually higher. Traditionally, HDL programming is the preferred way to access these devices, however high level synthesis tools can also be used [93] [94].

2.4 Binary Translation and Optimization

In this work, we also attempt to create accelerators solely based on program binaries, thus another related research area is binary translation. Before a program is actually executed on the processor, its binary can be modified with no additional input from the user or involvement of the compiler. This process can happen statically [95] [96], or dynamically while the application is running. In a lot of cases, the purpose of translating at the program binary level is to bridge the gap between existing software and newer hardware. In the late 90s and early 2000s, for instance, several VLIW machines provide their own dynamic binary translation (DBT) layer to ensure the compatibility of their processor with existing executables [97] [98] [99]. There are also DBT systems where the target ISA and source ISA are the same [100] [101], and the translation is performed to optimize the performance of the program. In the case of just-in-time compilation [102], the source binary is for a virtual machine. By translating Java bytecode to the native ISA, the runtime can achieve performance close to that of a natively compiled binary [103].

With the advent of parallel computers, binary translation also gets used to parallelize sequential programs. In [104], feasibility of dynamic binary parallelization is studied. Traces of instructions are parallelized to run on simulator of multicore machines. The system described in [105] performs both static analysis and dynamic recompilation of "super-blocks" to better take advantage of CMP and vector machines. A more recent project [106] aims to dynamically convert the older Intel SSE instruction set to the newer AVX one. There are also a few projects which perform the parallelization off-line. Vizer [107] provides a framework which statically vectorizes Intel x86 binaries, targeting a very restrictive set of loops. SecondWrite [108] is binary rewriter which generate multithreaded code for affine loops. It also showed there is but a small difference in performance improvement when the binaries, instead of the source code, are used as the starting point for parallelization. All these projects aim to minimize the user effort required to take advantage of the newer, faster but harder to program compute devices, which is an objective we would also like to achieve in this work. Some of their ideas are adopted and modified in our approach described in chapter 5, so they can be applied to FPGA accelerators.

Chapter 3

Synthesis of Decoupled Computational Pipeline on FPGAs

As mentioned in 1.3, the HLS tools statically schedule operations when generating accelerators, whose runtime behavior is therefore rather simple. Different parts of the generated circuit run in lockstep with each other, no dynamic dependency checking mechanisms, such as scoreboarding or load-store queueing, are needed. This rigid scheduling of operators, while producing hardware of simpler structure and smaller area, is also vulnerable to stalls introduced by cache misses or variable latency operations. In this chapter, we present a flow which alleviates this problem by structuring the computations and data accesses in the original program into a series of coarse-grained pipeline stages, through which data can stream. This pattern takes advantage of the FPGAs being throughput-oriented devices, while naturally overlaps computation and communication in the applications.

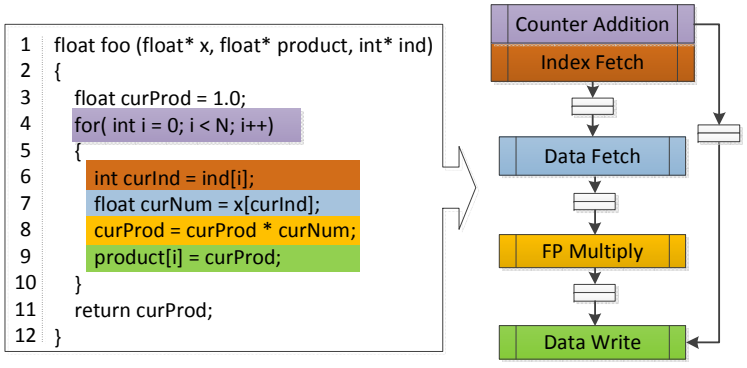


Figure 3.1: Converting a Simple Function to Decoupled Computational Pipeline

3.1 A Motivating Example

Shown in figure 3.1 is a simple example where separating a software kernel into multiple decoupled stages can improve the overall performance. As the shown function is pushed through the conventional HLS flow, opportunities for parallelization are discovered and a static schedule can be generated. The loop counter addition and the loading of $curInd$ can happen simultaneously, and the next iteration of the loop can start before the current iteration finishes. Meanwhile, because the floating-point multiply always uses its result from the previous iteration, the shortest interval with which we can start a new iteration is limited by the latency of the multiplier. The execution schedule of this function is shown in figure 3.2(a).

This execution schedule, unfortunately, assumes the best case latency for all the memory accesses. Since the computation kernel is turned into a monolithic accelerator, the centralized controller would have to stall the entire compute engine when long latency off-chip communication operations are occurring. This is less of an issue when the memory access patterns are known *a priori*, such that the data can be moved on-chip before it is needed, or if the data set is small enough to be entirely buffered on the FPGA. However, in this example, just like in many interesting algorithms, the data access depends on the result of computation and the memory footprint requires the use of off-chip storage. Figure 3.2(b) shows the execution of the generated hardware module in the presence of cache miss stalls. Note how the slowdown reflects the combination of all cache miss latencies. This does not have to be the case though, since there are parts of the computation graph whose progress does not need the memory data to be immediately available. These sections should be able to move forward. This extra bit of freedom in the execution schedule can potentially have a significant effect.

In this example, it is possible to decouple the execution of the floating point multiply and the data accesses from each other. Without a unified schedule, one load operator can keep requesting new data while the other one waits for previous requests to be responded by off-chip storage, and the floating point multiplier works through previously fetched data. This is achieved by having one module responsible for each of the memory accesses and the multiplication, as shown in figure 3.1. The FIFOs allow for the communication between these modules, while buffering the data already produced but not yet consumed. Over a long period of time, the stalls introduced by the memory accesses are shadowed by the long latency of the floating point multiplier, who is always supplied with the backlog of data in the FIFO when cache misses occur. As long as the overall bandwidth provided by the memory subsystem satisfies the need of the computation, the latency of the memory accesses can be tolerated.

Figure 3.2(c) shows the execution schedule when the decoupled pipeline is used. The latencies through the FIFOs are not taken into consideration here, e.g. FP multiply starts immediately after completion of $curNum$ load, but their effect should be minimal when amortized over a large number of iterations.

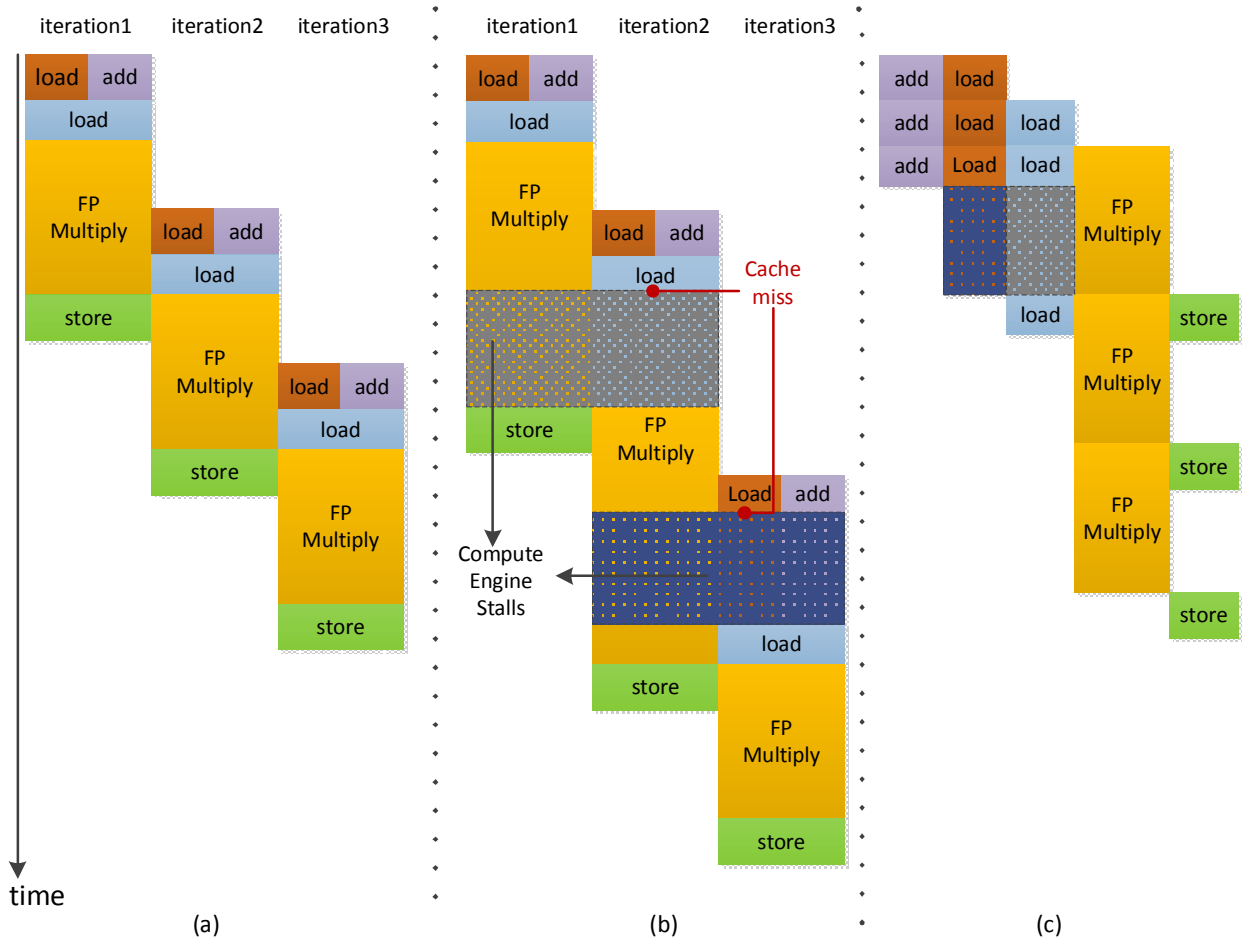


Figure 3.2: (a) Static schedule produced by HLS (b) Actual Execution with Cache Misses (c) Execution of Decoupled Computational Pipeline

3.2 Partitioning the Instructions

To generate modules who are running out of sync from each other, the original control data flow graph (CDFG) needs to be partitioned into multiple sets. Each set is to be converted to a self-contained function and synthesized to a stage in our pipeline. To maximize the performance of the resulted implementation, a few factors should be considered during our partitioning process. First, circular dependencies between nodes of the innermost loop need to be contained within each set. These strongly connected components (SCCs) in CDFG are associated with loop carried dependencies, and are the limiting factors for how aggressively loop iterations can be overlapped. The initiation interval (II) of loops are dictated by the latency of these cycles. As the communication channels will always add latency, having parts of an SCC in CDFG scattered across multiple stages would increase the II of the iterations,

Algorithm 1 Instruction Partitioning

```
1: procedure PartitionCDFG(G)
2:     ▷ SCCs of instructions formed with data/control/memory dependency edges
3:     SCCs ← allStronglyConnComps(G)
4:     DAG ← collapse(SCCs,G)
5:     topoSortedNodes ← topologicalSort(DAG)
6:     longSCCs ← getSCCWithLongOp(SCCs)
7:     memNodes ← findLdStNodes(G)
8:     memLongSCC ← LongSCCs ∪ memNodes
9:     allSets ← {}
10:    curSet ← {}
11:    while topoSortedNodes ≠ ∅ do
12:        curNode ← topoSortedNodes.pop()
13:        curSet ← curSet ∪ curNode
14:        if curNode ∈ memLongSCC then
15:            allSets ← allSets ∪ curSet
16:            curSet ← {}
17:        end if
18:    end while
19:    return allSets
20: end procedure
```

which are now executed in a distributed manner. Secondly, as we have demonstrated in section 3.1, with memory operations separated from dependency cycles involving long latency compute, we can have cache misses completely hidden by the slow rate of data consumption. Thirdly, to localize the effects of stalls introduced by cache misses, the number of memory operations in each set should be minimized, especially when they address different parts of the memory space.

Partitioning Algorithm

The first factor to consider for the partitioning algorithm is one of the observations made in [109], where sequential programs were converted into multithreaded codes running on multicore processors. Their algorithm finds strongly connected components (SCCs) in the original dataflow graph, collapses them into nodes and then heuristically partitions the resulted directed acyclic graph (DAG) into threads with balanced load. In our flow, the search for SCCs is also necessary and its outcome is used for the ensuing partitioning, which centers upon memory operations. In Algorithm 1, the steps taken to perform the partitioning are detailed. The SCCs are collapsed into new nodes, which together with the original nodes in the CDFG, are topologically sorted. The obtained directed acyclic graph is traversed and a new set is created whenever a memory operation or an SCC with long latency computation

is encountered. Here, long latency operations are those which cannot be completed within one clock cycle, and their categorization ultimately depends on the target frequency of the final implementation on the FPGA. Currently, we leverage Xilinxs Vivado HLS to generate latency estimate for various compute operations. With a target clock frequency of 125MHz, for instance, floating point multiply takes four clock cycles while a 32 bit integer addition can be completed within a cycle. As Vivado HLS is eventually used as the backend for our HDL generation, it provides accurate annotations for our flow.

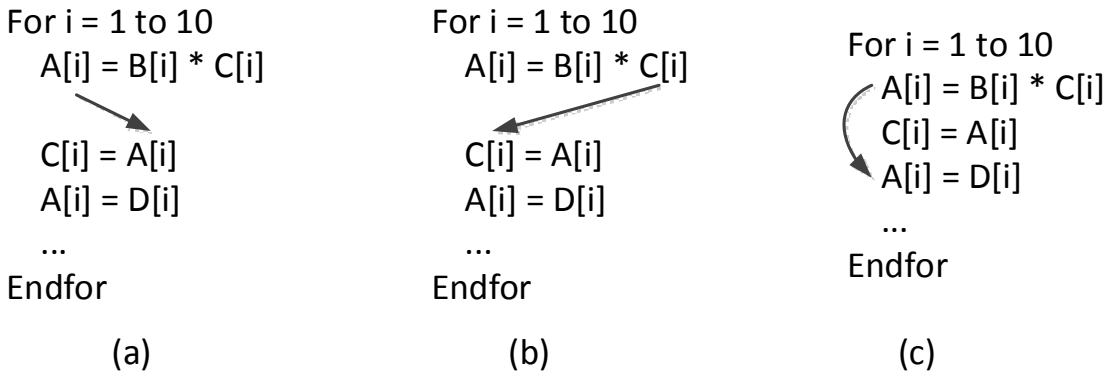


Figure 3.3: Three Types of Memory Dependencies

Preserving Memory Dependency

The semantics of the input high level language often create dependencies implicitly carried by memory accesses. Given two statements in a program, Bernstein's conditions [110] describe when they are independent and can be executed in parallel or out of order. If two memory operations access the same location and one of them is a store, their order in the original program execution must be preserved. More specifically, three types of dependencies need to be observed:

- Read-after-write (RAW) – When the same memory location is written by one statement and read by the other, there is a dataflow dependence between them (figure 3.3a). Performing them out of order results in the outdated operand to be used by the consumer statement.
- Write-after-read (WAR) – When a memory location is read by the first statement and subsequently written by the second, there is an anti-dependence between them (figure 3.3b). Performing them out of order overwrites the correct operand prematurely.
- Write-after-write (WAW) – When both statements write to the same location, there is an output dependence between them (figure 3.3c). Reordering the two writes exposes the wrong value to the subsequent reads.

For every pair of operations whose ordering needs to be preserved, a special dependency edge is added between them. Since the generation of the sets is performed around strongly connected components in the original dataflow, it is important to avoid adding unnecessary memory dependency edges. To achieve this, our flow currently relies on alias analysis to perform partitioning of the memory space. Accesses to disjoint memory regions can be safely reordered. When the source code contains pointer arithmetic or irregular memory accesses, compile time alias analysis may produce overly conservative results, in which case user annotations can be used to provide hints to the tool, similar to [111]. This partitioning of memory space naturally leads to creation of multiple data access interfaces, whose interactions with the memory subsystem can be customized, as will be elaborated in section 3.4.

Under certain circumstances, we do not want to directly add edges between memory operations as they hinder the CDFG partitioning. For the example in figure 3.4, during the execution of one outer loop iteration, the set of memory addresses accessed by the the load does not intersect with that of the store instruction. However, across different outer loop iterations, these two instructions can accesses the same locations. We can conservatively make them dependent on each other, but this creates an SCC that ultimately prevents any partitioning, as can be seen in the figure.

Alternatively, a memory barrier can be inserted after the completion of each outer loop iteration. To implement the barrier in the pipeline, the stage in charges of the last memory operation before the barrier broadcasts "barrier" tokens to pipeline stages who contain memory operations following the barrier. Eventually during the RTL generation, the local schedule of instructions also needs to be constrained such that the sender of the barrier tokens is not reordered to before its memory accessing predecessors, while the receivers of barrier tokens must execute before their memory accessing successors in the original program order as well. In figure 3.4, the store operation (A[i][j] = tmp) is followed by the sender of the barrier token, whereas the load operation (tmp = A[i-1][j]) in the other pipeline stage waits for the reception of barrier token for the previous outer loop iteration.

The insertion of the sender/receiver of the barrier happens after the instruction partitioning. Depending on if a partition is the sender or receiver of the token, an extra push/pop operation is introduced into the set of instructions, associated with the basic block where the barrier occurs.

Control Dependency Edges

Another source of edges in our CDFG – control dependency, also warrants some discussion. The simplest way to add the control dependency edges is to have every instruction dependent on the control transfer instructions of the immediate predecessors of its container basic blocks. For a loop nest, this will necessarily create a single SCC with all the branch instructions of the basic blocks within the loop nest. Essentially, a control flow "backbone" is generated and the branch tag tokens will need to be sent to all other modules whose instructions are predicated on these branches. The pipeline thus generated, while valid, may have higher communication channel count and less opportunities for optimization within each

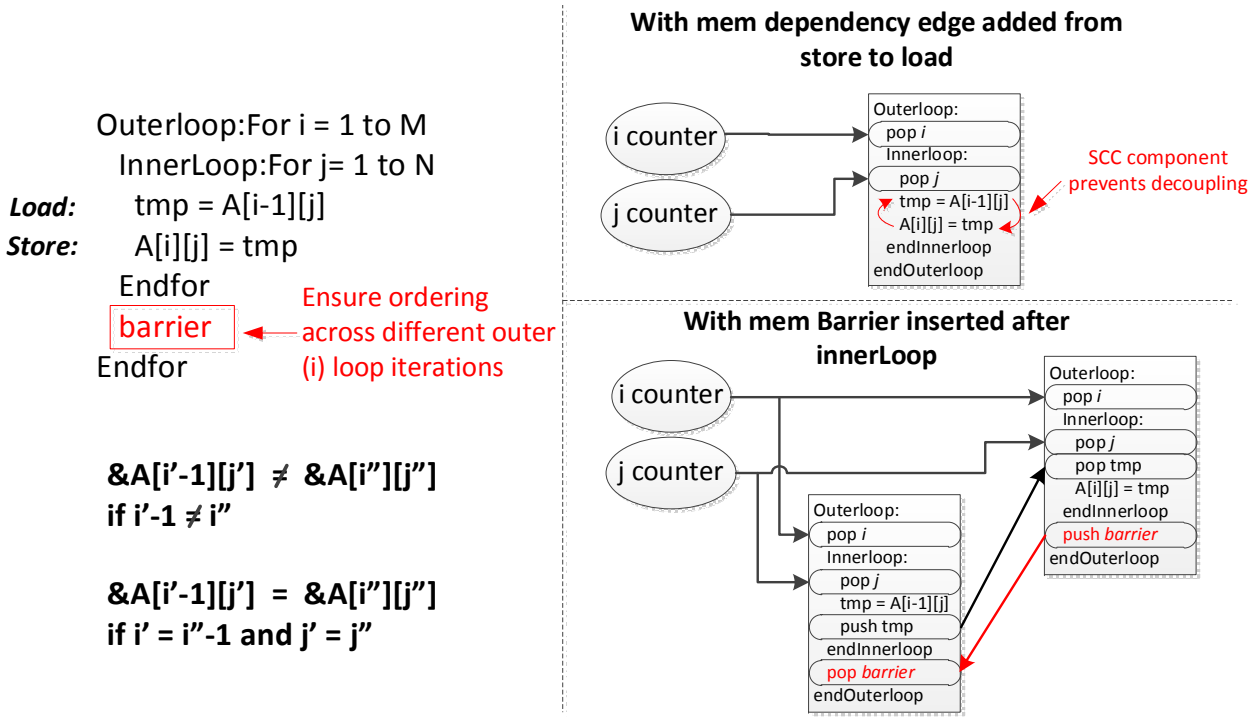


Figure 3.4: Barrier for Enforcing Order of Memory Accesses

module, which will be elaborated more in section 3.4. We thus perform a more aggressive predication/control edge insertion by looking for the earliest branch outcome which necessarily leads to the execution of an instruction. Given an instruction i and its container basic block bb, our flow looks for the nearest set of basic blocks BB' who are not properly post-dominated by bb, and insert the dependency edge between the branch instructions ending each member of BB' and i.

3.3 Construction of Pipeline of Subgraphs

With instructions partitioned into disjoint sets, the flow then reconstructs subgraphs from these sets of instructions. Each subgraph contains a subset of the original control flow/basic blocks and will later be synthesized as a self-contained function. The reconstruction process involves adding relevant basic blocks according to the association of the instructions with the current set.

- For every instruction i in the current set, recreate its container basic block if it's not already in the current subgraph.
- For every instruction j i depends on, recreate its container basic block if it's not already in the current subgraph.

- If j is associated with another subgraph, a "pop" function call is added locally as a placeholder in its container basic block.
- If i is producing operands for instructions assigned to other sets, a "push" function call is inserted after i.

The set of recreated basic blocks B are thus associated with either the instructions assigned to the subgraph, or the placeholder function call supplying them with operands. To mirror the relevant execution path in the original program, a few more steps need to be taken.

- In the control flow of the original program, find the nearest common dominator d of all the recreated basic blocks B, and add d to B. It will be the entry block of our control flow in this subgraph.
- Find all the paths P from d to $b \in B$, add all the basic blocks in each $p \in P$ to B.
- From each $b \in B$, find the paths P_b to every $b' \in B$ without passing through d. Add all the basic blocks in each $p_b \in P_b$ to B.
- Create the branching instructions at the end of each $b \in B$. If the branching instruction was already assigned to the current subgraph, nothing needs to be done, otherwise a "pop" operation is created to accept a branch target token from another subgraph, and then the actual branching is performed according to the received token.
- If d was inside a loop in the original control flow, any branch out of the set B is redirected to d. Otherwise, any branch out of the set B terminates the execution of this subgraph.

In the control flow graph of the original program, the set of paths whose starting points and end points both fall in B can be divided into two groups. Some of these paths never reach basic blocks outside of B, and the rest go out of B and then come back into B via d. The first group is completely contained within our subgraph. If there is any path in the second group, then d must be inside a loop, in which case we have effectively enclosed the subgraph with a *while(true)* loop and the execution of the part of the path within the subgraph will be repetitively activated by the availability of the proper tokens.

The insertion of "pop" and "push" operations is necessitated by various dependencies between the generated subgraphs. Other than data and control (branch) tokens we have mentioned, special tokens are also sent when ordering of memory accesses needs to be enforced. Each of the inserted operations corresponds to a hardware queue between the decoupled modules, the flow of tokens ensures the execution path are synchronized across different subgraphs and the right operands are supplied for computations distributed across the pipeline. Eventually, in the hardware module synthesized from each subgraph, these "pop" and "push" operations are blocking, as they operate on standard FIFO interfaces. Flow control between different pipeline stages are thus naturally introduced. The execution of the entire processing

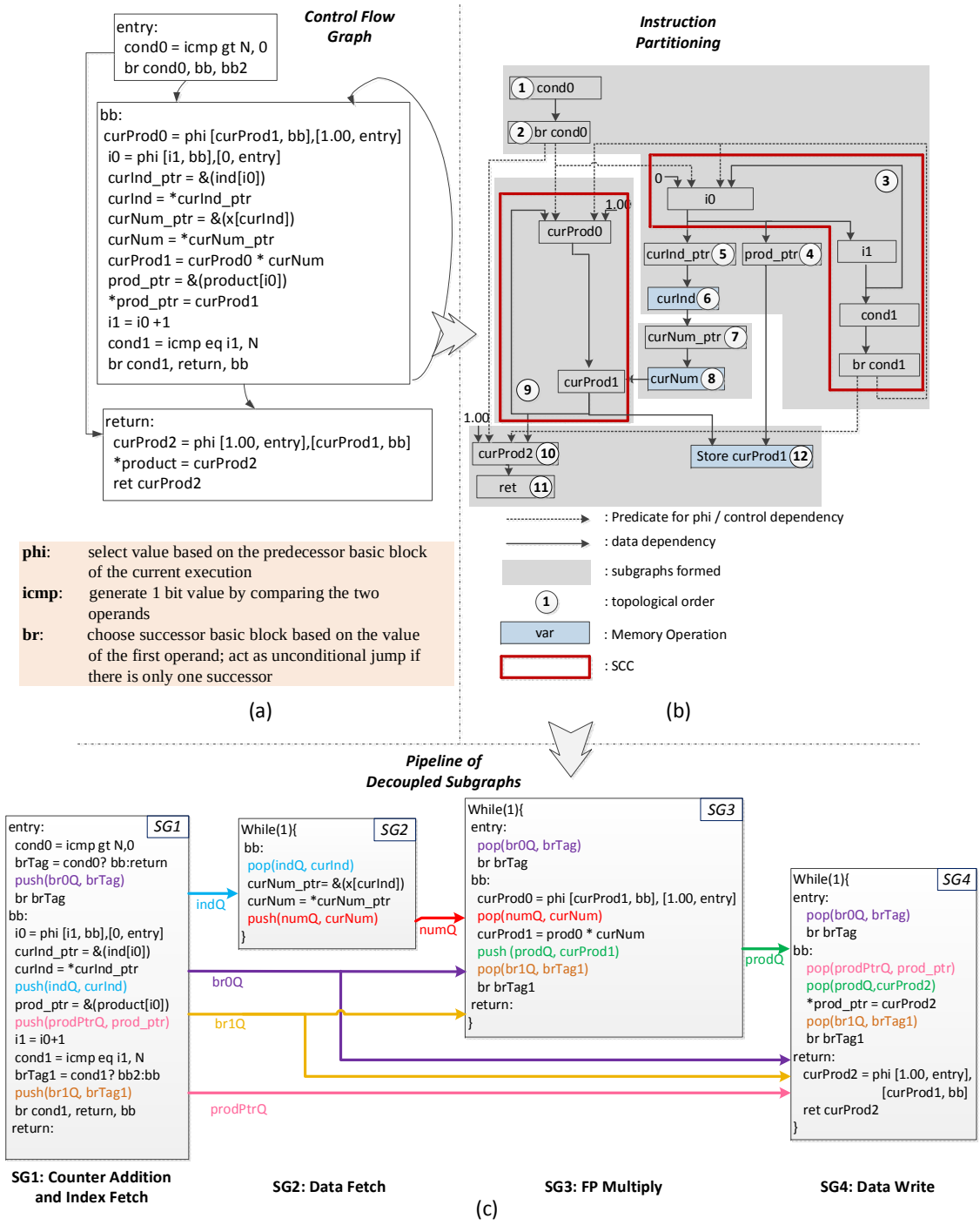


Figure 3.5: (a) The Motivating Example in Single Static Assignment Form (b) Partitioning of Instructions (c) Pipeline of Reconstructed Subgraphs

pipeline is completed when all modules are idling waiting for inputs and all hardware queues are empty. The runtime behavior of this pipeline thus resembles a streaming processing engine, with uncertainties introduced by memory access nodes smoothed out by FIFOs.

The implementation of our partitioning algorithm and the insertion of the communication primitives leverage the LLVM infrastructure [112]. The LLVM front end converts the instructions in the original program to the single static assignment (SSA) form, which makes it easy to track dependencies and thus facilitates all the steps in our algorithm. Figure 3.5 shows the SSA form of our example function and how the instruction partitioning and subgraph reconstruction can be easily performed given the explicit representation of dependencies. For better readability, we have converted the LLVM SSA intermediate representation (IR) to a less verbose, C-like version. Operators not available in C are explained in the figure. The inserted communication primitives are highlighted, the parameters to these "push" and "pop" invocations include the names of the associated queues and the data variables.

It is worth noting that transformations in LLVM framework are organized as "passes". Well-formed LLVM IR should be fed to and obtained from each of these passes. For our flow, a single loop-containing LLVM function is broken into multiple LLVM functions who communicate with each other and collectively perform the same computation as the original subroutine. To ensure each of the generated LLVM functions is well-formed, pointers are added to its argument list to represent the FIFO interfaces the token reads/writes act upon. The original subroutine, on the other hand, has all its computation replaced by invocation of these generated functions, in addition to allocation of communication channels. This is illustrated in figure 3.6. As our pipeline is represented in proper LLVM IR, additional optimization or analysis passes can be applied to it.

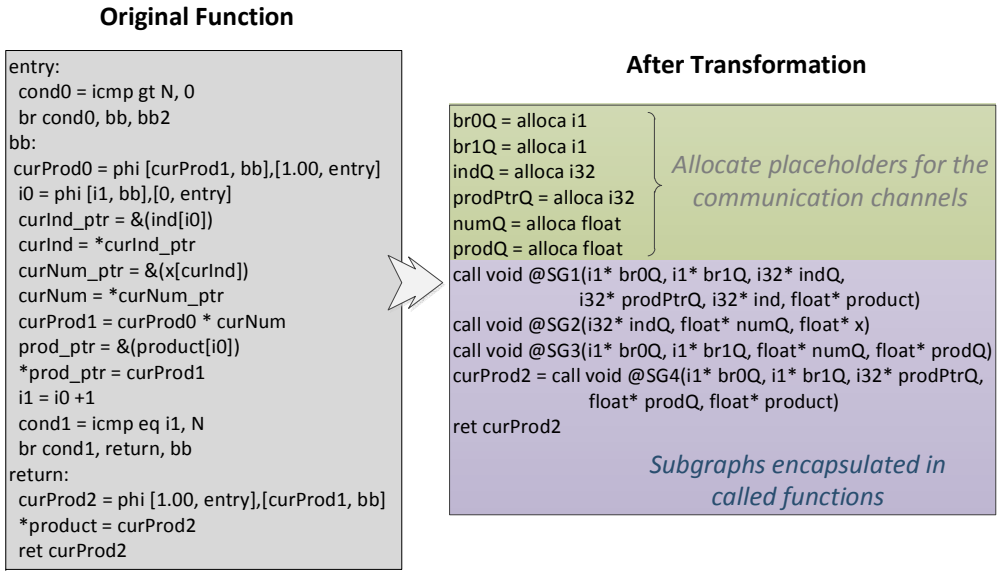


Figure 3.6: Transformation of a Function in LLVM IR

3.4 Optimization of Computational Pipeline

With an initial implementation of the computational pipeline, there are a few simple optimizations we can perform to improve its overall efficiency.

Simplification of Subgraph Control Flow

As mentioned in section 3.3, we generate the control flow of a subgraph by recreating basic blocks to form a self-contained function, in which every path from any member basic block to any other ones are covered. This approach sometimes introduces basic blocks whose only purpose is to have the execution path going through them, instead of doing any computation. Depending on the actual structure of the control flow graph, some of these blocks can be completely removed with proper redirection of branches.

To perform this simplification, we want to eliminate those basic blocks who do not perform any computation/communication, and are not divergent points in the control flow. More specifically, let the set of all basic blocks be BB_{all}, the set of basic blocks containing pop instructions (other than receiver of branch tags) be BB_{comm} and the set of basic blocks performing some computation be BB_{comp}, algorithm 2 outlines the steps for this optimization.

Our algorithm looks for basic blocks who may branch out of the subgraph, or can branch (directly or indirectly) to more than one members among BB_{comm} and BB_{comp}, and change their branches' destinations while deleting unnecessary basic blocks. As illustrated in figure 3.7, this optimization reduces the number of branch tag tokens to be transmitted between different subgraphs and thus the number of communication channels needed. It also reduces each subgraph's complexity, making the generated hardware smaller and faster. In the simplified CFG in figure 3.7, the original outer loop becomes an inner loop, thus can be pipelined independent from the inner loop in the other subgraph.

Duplication over Communication

A major overhead our flow introduces comes from the mechanisms added for different subgraphs to communicate with each other. In hardware accelerators, these are manifested as resource overhead in implementing the extra push and pop operations in each subgraph and more importantly, the FIFOs between different subgraphs. Even a minimal depth FIFO can use a non-trivial amount of FPGA resources. For instance, a 32 bit counter can be mapped onto 14 SLICEs while a 32 bit wide, 16 entry FIFO takes up 24 SLICEs. Considering the area efficiency of the whole pipeline, it can be beneficial to duplicate some of the computations to multiple subgraphs.

More specifically, for each subgraph, any outside node supplying it with operands may be a candidate for duplication. By copying some of these nodes into the subgraph, the edges cut by the subgraph boundaries also change and there can be an overall saving.

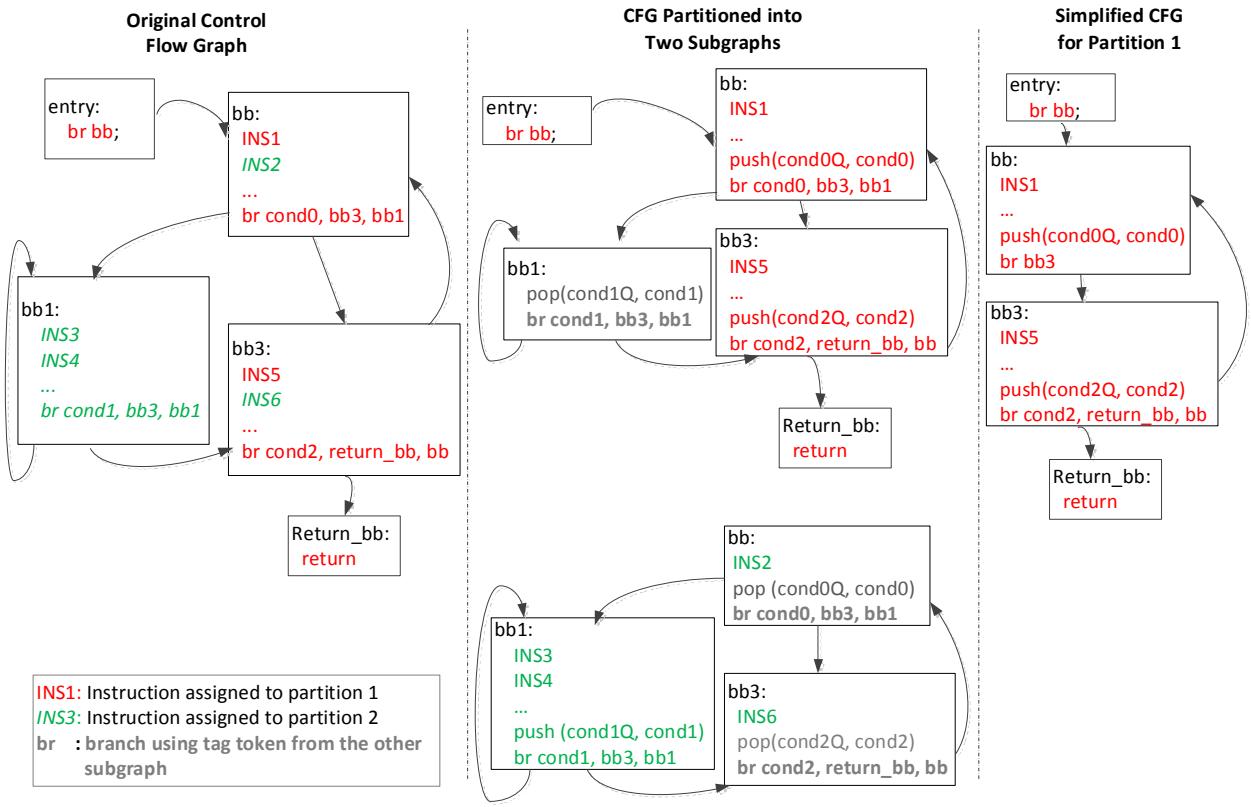


Figure 3.7: Subgraph Reconstruction and Simplification

In this graph cut problem (figure 3.8), we want to minimize the sum of the cost of the cut edges and the duplicated nodes. Since we are duplicating instead of moving, the tokens produced by the nodes involved are used locally and do not need to be sent to other subgraphs. Thus the edges going out of the subgraph from the duplicated node do not need to be included in our cost computation. In the figure, for instance, the initial partition cuts through the edge between node D and $subG$, which carries a cost W_H. If the combined cost of node D and edge AD (W_L) is lower than W_H, we can duplicate node D into $subG$ and obtain a lower cost design.

More formally, we can write this as an integer programming problem, where each node i is associated with a binary variable p_i. $p_i = 0$ if $i \in T$ and $p_i = 1$ if $i \in S$. Similarly, each edge ij is associated with a binary variable d_{ij} which takes the value 1 if $i \in S$ and $j \in T$, and 0 otherwise. Finally, w_{ij} is used to represent the cost of edge ij while c_i represents the cost of the node i.

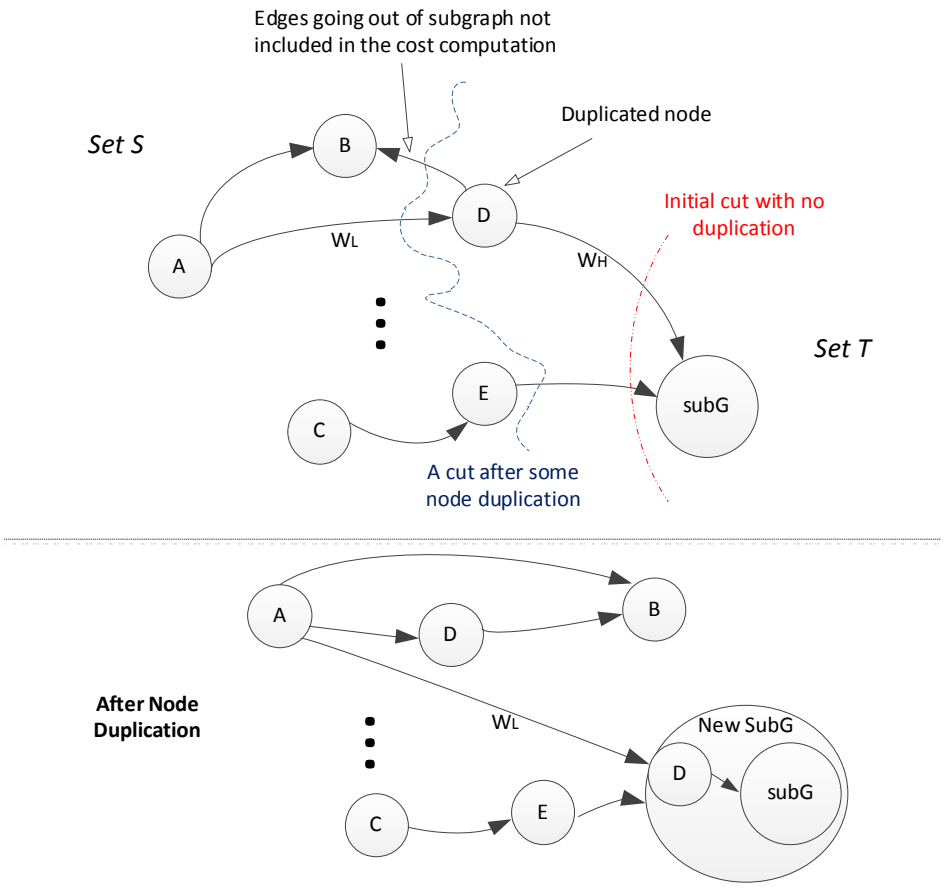


Figure 3.8: Duplicating Nodes into Subgraph

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{ij \in E} w_{ij} d_{ij} + \sum_{j \in T} c_j \\
\text{subject to} \quad & d_{ij} - p_i + p_j \geq 0, \quad ij \in E \\
& p_i \in \{0,1\}, \quad i \in V \\
& d_{ij} \in \{0,1\}, \quad ij \in E \\
& p_{subG} = 0
\end{aligned}
$$

We do not wish to duplicate memory operations, thus a very large cost is associated with each of the memory accesses. For all other nodes, The resource consumed are obtained using the report generated by our backend, Vivado HLS. Similarly, for each of the edges, we synthesize a FIFO of depth 64 using the Xilinx FIFO generator. As the FPGAs contain a variety of resources, we assign a numerical value for each type of primitives used by the node (table 3.1). We derive the percentage silicon area each type of resources occupies from a die photo of an older Xilinx chip [113], and generate the cost of nodes and FIFOs accordingly.

With the CPLEX [114] optimization software, the formulated ILP can be solved for each subgraph obtained. The experimental evaluation presented in section 3.6 has incorporated the outcome of these optimizations.

Table 3.1: FPGA Resource Cost Value for Optimization Formulation

Resource Type	Logic(Slices)	BRAM	DSP
% Chip Area	44.2	12.4	2.0
Num. of Primitives on Die	17280	148	64
Assigned Cost Value	25	840	309

Memory Optimization

Pipelining of Memory Transactions

When creating the decoupled computational pipelines, each memory operation is assigned to a subgraph and the generation of memory requests are synchronized with the execution of the associated module. For instance, in the subgraph shown in figure 3.9, the HLS tool eventually responsible for RTL generation will need to create a unified schedule where the loop counter addition (line 13), load (line 9) and push (line 10) operations are each assigned to a fixed time slot. As the entire module would be stalled when the load misses, no further memory transactions are initiated even though the address needed for the next load can be computed, and the downstream FIFO has enough empty space. Meanwhile, modern memory subsystems usually have the capability to handle many outstanding memory transactions, in fact, their bandwidth has been improving much faster than their latency [115]. It is therefore undesirable for these hardware components to be artificially sensitized towards memory access latencies, resulting in a underutilization of the bandwidth.

To resolve this issue, our flow splits the involved memory access operation into two disjoint portions: *send_req* and *receive_resp*. As shown in figure 3.9, *send_req* takes the place of the load instruction in the original subgraph, and pushes the addresses into the memory subsystem. As our partitioning algorithm creates a new set after adding a memory access, consumers of the returned data are always in other downstream subgraphs. The response port can thus be directly connected to the downstream module. Store instructions can also be dealt with in a similar fashion, though the write response only matters in the presence of memory dependencies. With this special transformation, each memory access node is capable of pipelining many outstanding requests so long as the memory interface is ready. Note not every memory access undergoes the modification. There are cases where the result of a load, or the completion of a store is needed by another operation in the same subgraph. A classic example is the pointer chasing in linked list traversal, where the address for the subsequent memory request would not be available until the current load gets its response (figure 3.10). In general, if a memory access is in a dependency cycle carried by the innermost loop, our flow categorizes it as non-optimizable, and during our partitioning process, it would

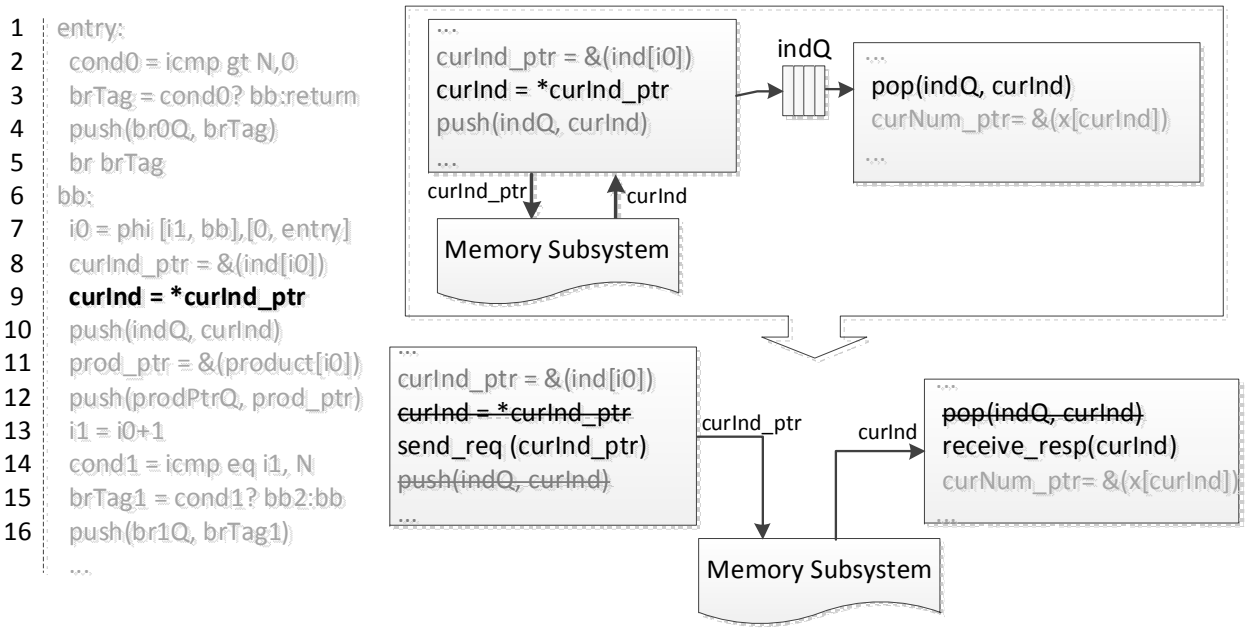


Figure 3.9: Pipelining Memory Transactions

also have been buried inside one of the SCCs. Here we assume that the input to our tool has already undergone potentially helpful high level optimizations. It is well known that for regular applications with statically analyzable memory access patterns, techniques like loop interchange can move the dependency cycle to the outer loops [116]. With the insertion of memory barriers, some of these non-optimizable memory accesses can become pipelinable. However, as the benchmarks we use for this chapter is non-regular, the applicability of these high level transformations is not investigated.

Customization of Data Access Mechanism

Non-regular application kernels often contain a variety of memory access patterns, i.e. streaming, strided or random. General purpose processors use caches as a best effort solution to serve all the different interminglings of these patterns in various applications. The flexibility of the FPGAs, on the other hand, allows for customization of the data access mechanism.

In our flow, partitioning of the memory space has provided an opportunity to create better hardware for memory access on the reconfigurable fabric. Each independent data access interface, corresponding to one memory partition, can be supported differently according to the nature of the address stream it generates. Caches, being rather expensive to implement on FPGA, might not always be the ideal structure connecting the accelerators and the external memory. For instance, there is no reuse of data for streaming type accesses, our

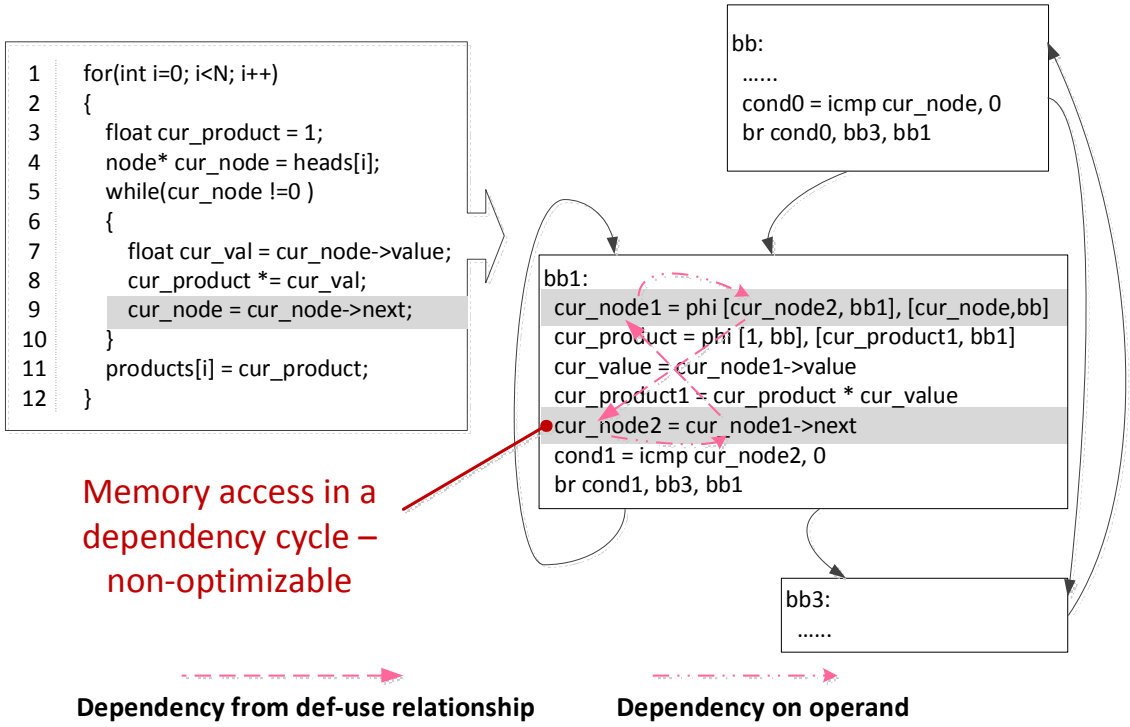


Figure 3.10: Non-optimizable Memory Access

flow therefore does not allocate an on-FPGA buffer. Rather, the *send_req* module shown in figure 3.9 is modified to send burst requests, concatenating multiple load/store in the original program execution. In this mode, a single request sent corresponds with many instances of *receive_resp*, which get executed by a loop. We therefore move the *send_req* operation to outside of the loop, and send the memory address with the total size of the requested data. As the memory interface limits the maximum burst size, an adapter module is created to break the request into ones with appropriate sizes. These are then streamed into the memory, as illustrated in figure 3.11. To automatically generate burst memory accesses, the iteration count needs to be computable within the subgraph. The duplication of SCCs described earlier in this section often copies loop counters into the subgraph firing off the memory requests, enabling this transformation.

When there is a cycle of dependency through memory, an on-FPGA buffer would be beneficial. Our flow currently adds a general purpose cache in this case, but if the particular address stream is analyzable and the reuse distance can be determined statically, structures like smart buffers [117] may be incorporated. Even in the case when the memory accesses are random and a general purpose cache is the only plausible solution, its size and associativity can be adjusted according to a runtime profile.

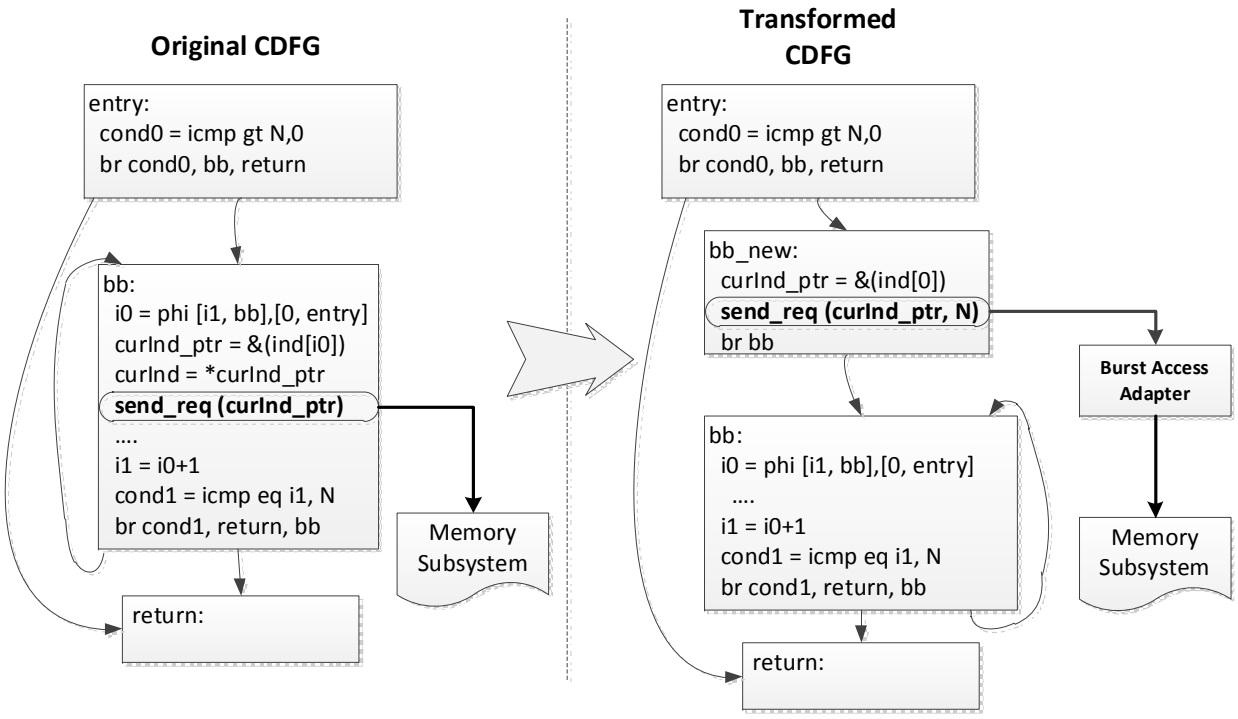


Figure 3.11: Transformation for Burst Memory Access

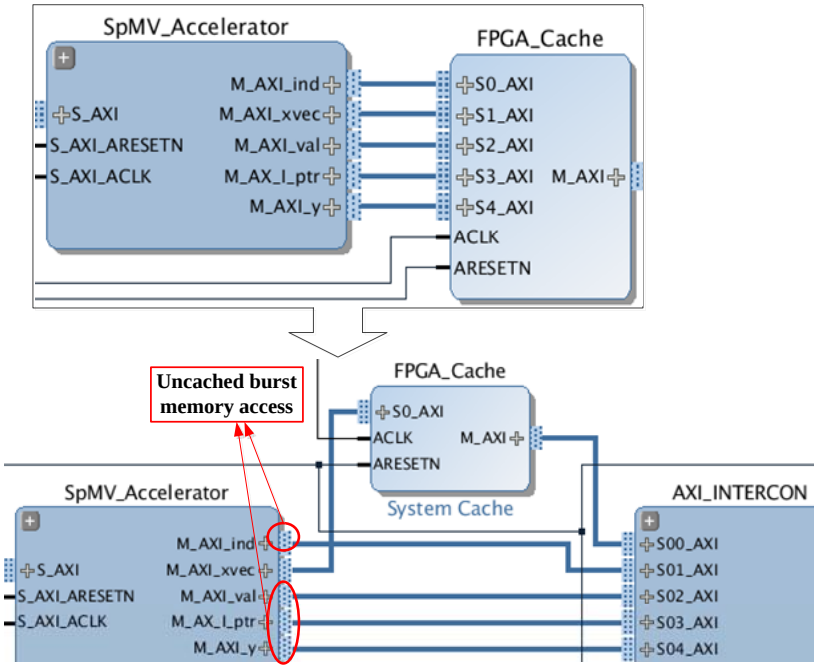


Figure 3.12: Optimized Data Access Mechanism

Shown in figure 3.12 is an example where the described techniques are applied to one of our benchmarks. The cache added between the accelerator and the memory subsystem was originally shared by the five memory interfaces. However, as four of those will load/store data in continuous addresses, they can be converted to burst accesses. The cache is now exclusively used by "xvec" which accesses data randomly. The benefits of these transformations are incorporated into the final performance results shown in section 3.6.

3.5 Hardware Generation

As mentioned in section 3.3, LLVM is used for implementing the synthesis flow for the computational pipeline. To create the final hardware, we translate each of the generated functions back to C syntax and then feed it to an existing HLS tool.

LLVM to C translation

As the computational pipeline is represented as well formed LLVM IR, it's translation to C is just another LLVM pass we write and apply within the LLVM framework.

Values in LLVM IR can take on types not recognized by standard C compiler. For instance, integer types can be of arbitrary width (1-$2^{23} - 1$ bits), most of which does not have equivalence in C language. A vast majority of them do not occur in our flow since the original input is in C. The branch tag tokens and memory dependency tokens, however, generally have very narrow width. These types are supported by some HLS tools, but not standard software compilers. Thus we also generate a separate mapping file containing a list of C macros, rounding up the width of the data to one of the standard ones. Subsequently, for every synthesized LLVM function, i.e. a stage in the computational pipeline, we can also have a software implementation compatible with conventional compilers.

The actual conversion of LLVM to C involves just a few simple steps. The function signature in LLVM is very similar to that of a normal C function. The conversion is thus rather trivial. Meanwhile, at the instruction level, most of the LLVM IR can also be directly mapped to C statements, as illustrated in figure 3.5. One transformation we perform is in dealing with the ϕ operations, which generate values of variables based on the the incoming control edges. In the case where the source instruction for an operand is within the same subgraph, instead of producing to the operand, this source instruction writes directly to the output of the *phi* operator. On the other hand, if the source is in another subgraph, a load operation is inserted to the basic block where the data is produced, but again assigning the result directly to the output variable of ϕ. The ϕ instruction itself can be removed. An example of this conversion is shown in figure 3.13.

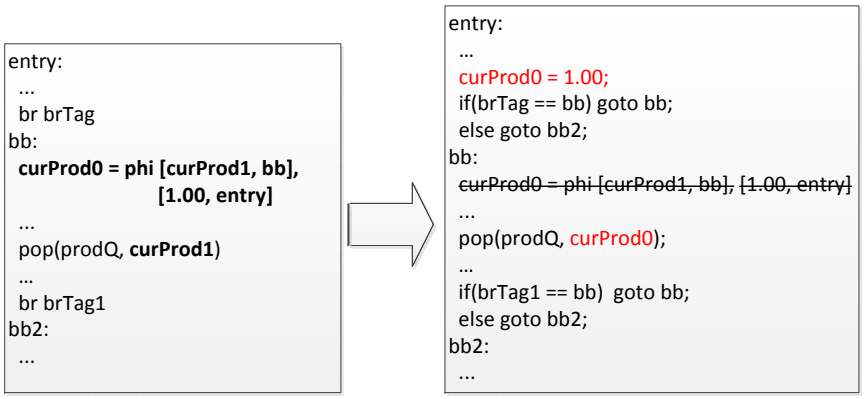


Figure 3.13: Converting ϕ Operator to C

Testing with Multithreaded Software Implementation

We have so far created a C implementation for each stage in the decoupled computational pipeline. To test the functional equivalence of the synthesized pipeline with the original function, our flow also creates a multithreaded software implementation using the *pthread* library. Each of the generated stages is assigned to a separate thread, running concurrently with all other stages. Channels connecting different stages are implemented as instantiations of a special C++ data structure. An templated array is used to represent a FIFO. A channel can contain multiple arrays when it fans out to more than one consumer threads. Each array is protected with a mutex and associated with a conditional variable. Blocking reads and writes are implemented with *pthread_cond_wait* while *pthread_cond_signal* wakes up blocked threads when token/space becomes available.

The original sequential function, which has all its computation replaced by LLVM function invocations, is examined and converted to a top level wrapper. In addition to allocating and initializing the channels, this wrapper also initializes each thread with a pointer to a packaged list of function arguments. After starting each thread, it synchronizes at the end to ensure all threads are completed. With this setup, we can easily compare the outcome from calling the original function v.s. the transformed wrapper function, any differences signify flaws in our pipeline generation flow.

An interesting aspect in the software implementation is the sizing of the FIFOs. In chapter 4, we will discuss how sizes of the channels affect the behavior of the pipeline. For our software implementation, as long as each FIFO contains more than one slot, the execution of the pipeline should produce the same result as the original code. However, a greater number of slots in the FIFOs would allow each thread to run for longer before going into wait state.

Generating System with Xilinx Vivado Tools

When synthesizing each C functions using Xilinx's Vivado HLS, every pointer argument through which actual memory references occur is converted to a independent memory port. These are to be connected to the memory subsystem, as will be described later in section 3.6. Other pointer arguments are used for inter-subgraph communication, they are associated with pragmas directing the creation of FIFO interfaces. In terms of the scheduling computation in each function, we ensure the inner loops are all aggressively pipelined, while the barriers are contained in non-inlinable function such that operations are not reordered with respect to them.

For the next step, where all the components are to be connected together, we rely on the FIFO generators provided by Xilinx. Similarly, on-FPGA cache and the interconnect, which is used to bridge the computational pipeline and the memory subsystem, are also parametrized Xilinx IPs. Within our flow, TCL script is generated according to the communication requirement of the subgraphs. The construction of the whole pipeline is then performed in Xilinx Vivado IP Integrator by invoking this TCL script. All the steps involved in our pipeline generation flow are summarized in figure 3.14.

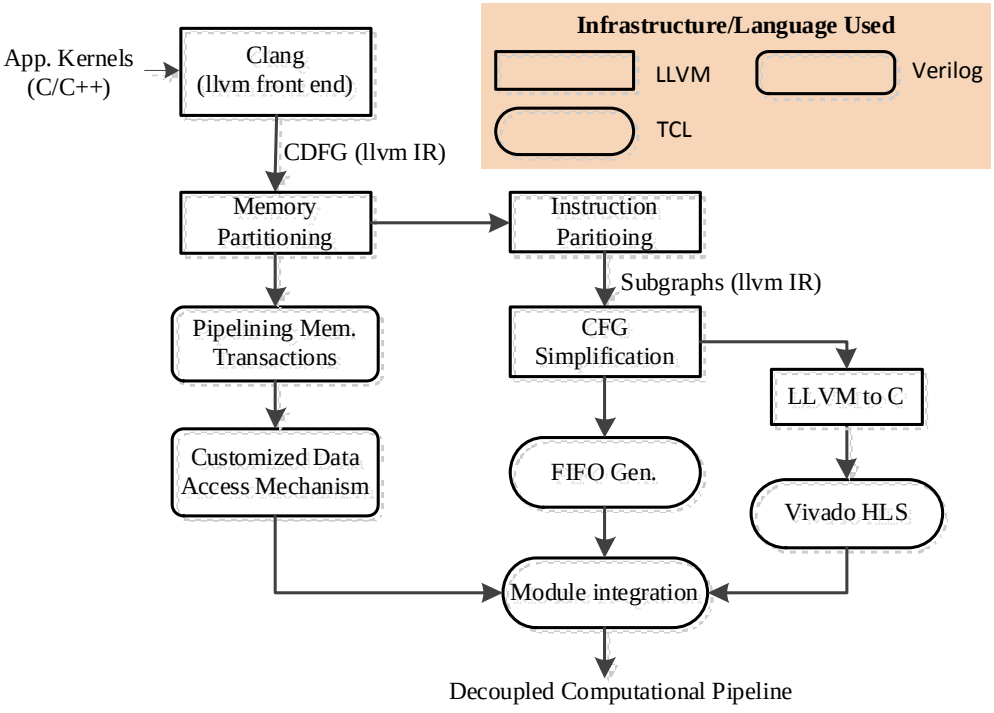


Figure 3.14: Pipeline Generation Flow

3.6 Experimental Evaluation

Experiment Setup

To demonstrate the benefits of our approach, processing pipelines are synthesized and physically implemented on an FPGA. The particular device used for the experiments is the Zynq-7000 XC7Z020 FPGA SoC from Xilinx, installed on the ZedBoard evaluation platform. The SoC is divided into two parts: an ARM-processor based processing system (PS), and the programmable logic (PL). The baseline for our evaluation is the performance of each software kernel running on the ARM core in the SoC. It is an out of order, dual issue hard processor running at 667MHz. The Zynq platform also provides two options for the reconfigurable accelerators to access the main memory subsystem: through the accelerator coherence port (ACP), or the high performance (HP) port. The former connects to the snoop control unit in the processing system and thus uses/modifies the processing system's on chip cache. The HP port connects directly to the memory controller, which necessitates the flushing of cache lines by the processor if a cached data structure is accessed by the accelerator. In either case, if memory states are also buffered in the reconfigurable array with caches, they need to be explicitly pushed to the processing system side after the accelerator finishes running. As both ACP and HP are slave ports, they provide no mechanisms to extract data from the FPGA when the ARM processor is running. The interaction between the generated accelerators and the main pieces of the FPGA SoC is shown in figure 3.15.

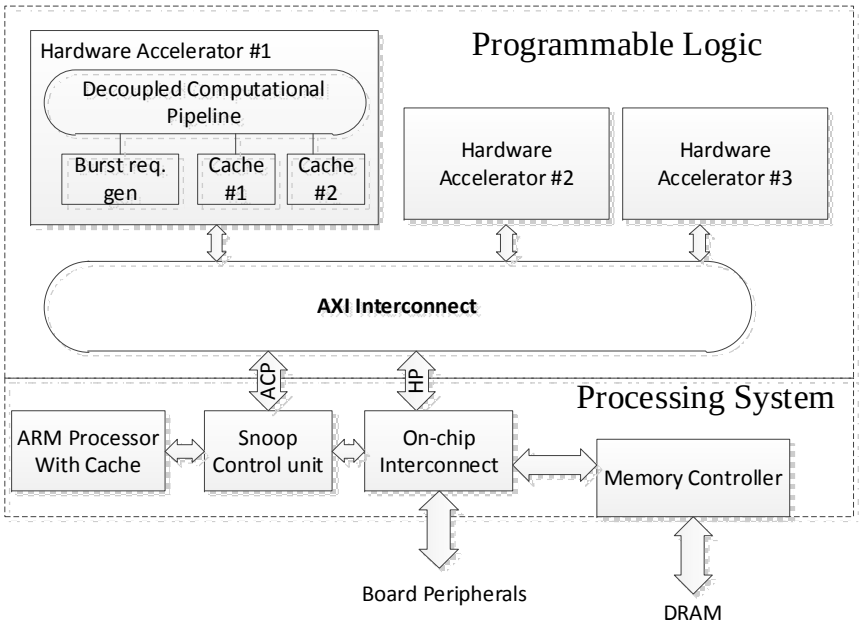


Figure 3.15: Implementation of Computational Pipeline in FPGA SoC

In our study, Vivado HLS, a state-of-the-art high level synthesis tool provided by Xilinx, is used for generating the conventional accelerator (Con.ACC) as well as the individual modules in our decoupled computational pipeline (DCP). With the target clock period set to 8ns during HLS, the tightest timing constraints post place & route implementations managed to meet range from 111 to 150MHz. All design points shown in this section use the highest achievable frequency as the actual operating clock frequency.

Benchmark Descriptions

The target applications of our flow are algorithms where control flow and data access patterns depend on run time data or results of computation. The four irregular kernels listed below are pushed through our flow:

- **Sparse matrix vector (SpMV) multiply** is a computation kernel that has been studied, transformed and benchmarked many different ways in various research projects. Our purpose here is not to produce the best-performing SpMV multiply using special data structure and memory allocation schemes. Rather, we use the most basic and widely used algorithm and storage format to evaluate how much benefit our flow can provide. The input matrix is stored in compressed sparse row (CSR) format. Loads from an index array are performed before numbers can be fetched for the actual floating point multiply.

- **Knapsack** is a problem in combinatorial optimization. Given a collection of items, each with its own weight and value, knapsack tries to select a subset of them such that the total profit is maximized while the weight limit is not violated. It is a classic problem which is often solved using dynamic programming, where the memory addresses accessed come from computation. It is therefore hard to prefetch the needed data unless the entire dataset fits in on chip buffer.

- **Floyd-Warshall** takes a graph as input and computes the shortest distances between any pairs of vertices. Similar to knapsack, it is solved by dynamic programming. Even though all memory references are regular, i.e. simple functions of the loop indices, the control flow depends on the results of computation.

- **Iterative depth first search** is again a widely used graph algorithm. The version used for our experiment makes use of a stack and operates on pointer based data structures.

The irregularity in memory accesses and execution paths in these benchmarks makes it hard for existing HLS tools to directly generate efficient hardware. The conventional accelerators, when implemented on the FPGA, are also very sensitive to the latency of data accesses, due to the high ratio of memory operations to computation.

Table 3.2 describes the characteristics of the input data set for each benchmark. As our approach is primarily used for cases where off-chip communication plays a significant role

Table 3.2: Input Data Set for the Benchmarks

Benchmark	Description of Input Data	Total Size of Input Data
SpMV Multiply	Matrix dimension = 4096 Density of Matrix = 0.25	≈ 16 MB
Knapsack	Weight Limit = 3200 Number of Items = 200	≈ 5 MB
Floyd-Warshall	Number of Nodes = 1024	≈ 8 MB
Depth-First Search	Number of Nodes = 4000 Number of Neighbors per Node = 200	≈ 3 MB

in determining the final performance, the input data size are chosen to be much larger than typical on-FPGA cache. For smaller problems where the entire input data set can be buffered on chip, the conventional DMA+accelerator approach, as described in [80], would not suffer from variable data access latency and our decoupled computational pipelines would offer little advantage.

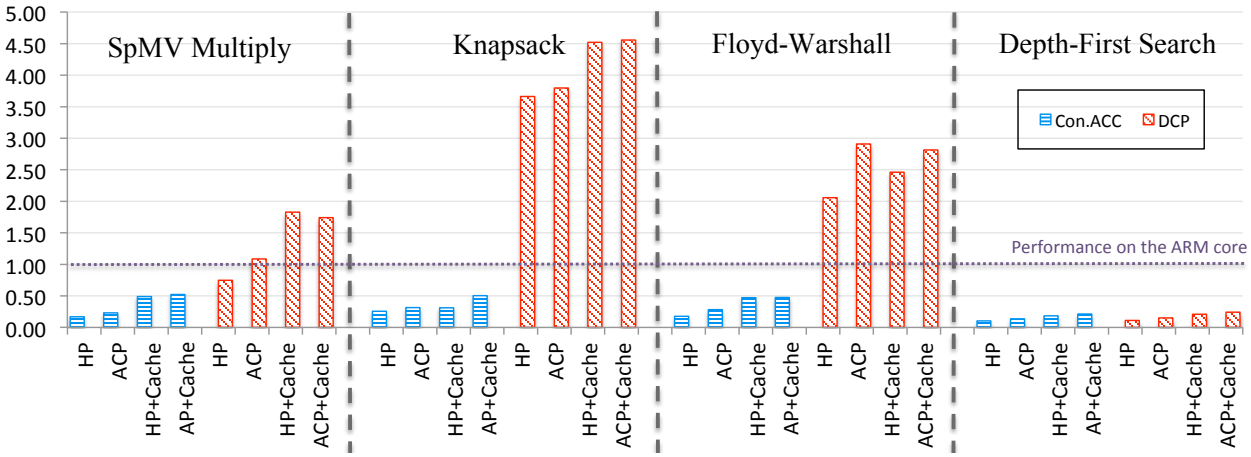


Figure 3.16: Performance of Conventional Accelerators and Decoupled Computational Pipelines

Performance Comparisons

In figure 3.16, performance of the conventional accelerators and decoupled computational pipelines are compared, all the numbers are normalized to the baseline. Both Con.ACC and DCPs are supported with different memory subsystem configurations. Direct connections to

the HP port generally incur the highest memory access latency as the data are not cached in the PS or the PL. This latency is reduced when ACP is used. To bring the data even closer to the accelerator, cache IPs can be instantiated on the PL. Our implementations use Xilinx's system cache IP, configured to be 64KB and 2 way associative.

In all four benchmarks, accelerators generated directly from software kernels using conventional HLS flow actually result in a performance degradation compare to running the kernels on the hard processor. The fastest Con.ACC implementations with on-PL caches, only manage to achieve throughput less than 50% that of the baseline. The superscalar, out-of-order ARM core is capable of exploiting instruction level parallelism to a good extent and also has a high performance on-chip cache. The additional parallelism extracted by the HLS tool is evidently not enough to compensate for the clock frequency advantage the hard processor core has over the programmable logic and the longer data access latency from the reconfigurable array.

With our methodology, the computational pipelines generated are rather competitive against the hard processor, even without a reconfigurable cache. For SpMV multiply, knapsack and Floyd-Warshall, when DCPs are directly connected to the PS through the ACP, the average performance is 2.3 x that of the baseline—representing an 8.4 x gain over the conventional accelerators. Upon the addition of on-PL caches, the average runtime of DCPs was further reduced by 18.7%, making the DCPs about 2.9x faster than the baseline.

It is also apparent that our approach has its limitations, as demonstrated by its ineffectiveness in the benchmark depth first search. The kernel performs very little computing but lots of memory accesses. The use of a stack in DFS also creates a dependence cycle through the memory and consequently, the performance is fundamentally limited by the latency of memory access. Thus there were only small differences between the performance of the conventional accelerator and the decoupled computational pipeline. Besides, the memory access pattern does not provide many opportunities for optimizations. As a result, DCP and Con.ACC achieves performance far below that of the baseline, which has a much higher clock frequency and a faster cache.

The figure also allows us to examine the effect of different memory configurations on the overall performance of the design. There is a general trend that the performance improves as the data get cached closer to the computation engine. For conventional accelerators, the ratio of the fastest to the slowest design points is, on average, 2.4x. For decoupled computational pipeline, this ratio is reduced to 1.7x. The relative insensitivity of the DCPs towards data access latency is rather evident from this difference.

Overall, for kernels suitable for FPGA acceleration, there is a significant performance advantage in using decoupled computational pipelines. If we compare the best results using DCP to conventional accelerators, we see improvement of 3.3 to 9.1 times, with an average of 5.6.

Table 3.3: Resource Usage of Decoupled Computational Pipelines and Conventional Accelerators.

Benchmark		ACP				ACP + 64KB Cache			
		LUT	FFs	BRAM	DSP	LUT	FFs	BRAM	DSP
SpMV Multiply	Con.ACC	9873	9116	10	5	7918	6792	21	5
	DCP	8577	8837	10	5	6718	6788	21	5
	% change	-13.1	-3.1	0	0	-15.2	-0.1	0	0
Knapsack	Con.ACC	7672	7490	8	6	6573	5885	21	6
	DCP	8089	8787	8	6	6970	7256	21	6
	% change	+5.4	+17.3	0	0	+6.0	+23.3	0	0
Floyd-Warshall	Con.ACC	2491	3528	0	10	3806	4629	19	10
	DCP	7659	7210	0	10	8995	8309	19	10
	% change	+207.5	+104.3	0	0	+104.4	+79.5	0	0
DFS	Con.ACC	4810	4929	4	0	4931	4594	21	0
	DCP	8509	7813	4	0	7436	6298	21	0
	% change	+76.9	+58.5	0	0	+50.8	+37.1	0	0

Area comparison

To quantify the impact of our proposed methodology on area, we have compared the FPGA resource usage of conventional accelerators and the decoupled computational pipelines. Table 3.3 shows the results, where each accelerator is complemented with two different memory subsystem configurations.

The difference in area between DCPs and Con.ACCs is effected by two factors. There are additional costs associated with the communication primitives and FIFOs for the DCP implementations. On the other hand, the original programs are partitioned into subgraphs and separately turned into hardware in DCPs, which sometimes can reduce the depth of the internal pipeline in the processing modules, resulting in area savings. The overall change therefore depends on which factor plays a larger role, and is ultimately application specific.

3.7 Discussion and Future Work

The experimental results have validated the approach we have proposed and implemented. Decoupling of execution between different parts of the control dataflow graph can yield significant benefits. With different parts of the graph each having their own controller, non-statically schedulable stalls get isolated and more opportunities for optimization can be exploited. Unfortunately, these benefits do not come for free. We observed significant area increase when generating our decoupled computational pipeline. How to strike the right balance between performance gain and the area cost by devising different ways to partition the CDFG is an interesting dimension for future exploration.

The algorithm we have implemented in this chapter is rather simplistic. It is also very aggressive in that all separable memory accesses are assigned to a new partition. Furthermore, as we perform topological sort on the nodes before partitioning them, many edges might be cut when a new subgraph is created, especially when the dependencies between instructions are relatively denser. Therefore, as the input kernels increase in size, our algorithm may result in pipelines with prohibitively high number of communication channels. The optimization we described in section 3.4 partially addresses this issue, but it does not reduce the total number of subgraphs. Similarly, it is possible to model the instruction partitioning step as a multiterminal cut problem [118], which is an NP-hard problem with a few known approximation algorithms, but the number of subgraphs can still get large, incurring high area overhead.

One possible direction for future exploration is to generalize our approach by incorporating, as a parameter, the set of nodes which must be decoupled from each other. It would then become possible to optimize for the right number of subgraphs, each centered around a group of memory access nodes or strongly connected components. It is of course a huge search space, so we might need to rely on randomized algorithms such as simulated annealing to search towards a good solution. A challenge there is to come up with a quick estimator of benefit from an instruction partitioning. Unlike the cost of communication channels introduced between separated instruction groups, which can be easily annotated into the edges of the original CDFG, the performance gain of decoupling instruction nodes is harder to model. Empirically, certain quantities can be precomputed or profiled to assist the estimation process. The initiation intervals associated with a strongly connected component can be used to estimate how fast an inner loop memory access node, if being placed in the same partition, can initiate new requests. Meanwhile, if more locality can be observed from the profiled memory access pattern of a particular load/store instruction, the performance hit it incurs on its partition is likely to be smaller, due to a lower cache miss rate. To the first order, decoupling nodes exhibiting strong locality into separate partitions has a small performance benefit while isolating nodes with completely random access patterns can bring about significant gain. In our motivating example, for instance, isolating the data fetch (assumed to target random addresses) is more meaningful than separating floating point multiply and the sequential write at the end of the inner loop. On the other hand, off-chip communication bandwidth also imposes an upper bound on how much concurrent data requests from the generated pipeline can be accommodated. After a certain point, having more partitions/subgraphs each firing off memory requests independently into a saturated channel would not increase the overall throughput of the computational pipeline. To integrate all these effects into a proper formulation, more in-depth investigation is required to determine how to best link these quantities with each other and associate appropriate coefficients.

Algorithm 2 Control Flow Simplification

```
1: procedure SimplifyCFG
2:     keepers ← {}
3:     keeperQueue ← BB_comm ∪ BB_comp
4:     while keeperQueue ≠ ∅ do
5:         curKeeper ← keeperQueue.pop()
6:         keepers ← keepers ∪ curKeeper
7:         allPredecessors ← getPredecessors(curKeeper)
8:         for all curPredecessor ∈ allPredecessors do
9:             ▷ look for divergent points from predecessors
10:            backwardDFS(curKeeper, curKeeper, curPredecessor, keeperQueue)
11:        end for
12:    end while
13: end procedure
14: procedure backwardDFS(curKeeper, prevStep, curPredecessor, keeperQueue)
15:    if curPredecessor ∉ BB_all then return
16:    end if
17:    if !curKeeper.postDominate(curPredecessor) then
18:        keeperQueue.push_back(curPredecessor)
19:        remapBranch(curPredecessor, prevStep, curKeeper)
20:        curKeeper ← curPredecessor
21:    end if
22:    allPredecessors ← getPredecessors(curPredecessor)
23:    for all nextPredecessor ∈ allPredecessors do
24:        backwardDFS(curKeeper, curPredecessor, nextPredecessor, keeperQueue)
25:    end for
26: end procedure
27: procedure remapBranch(predecessor, prevTarget, realTarget )
28:    branchInst = predecessor.getBranchInst()
29:    branchDstBBs = branchInst.getAllSuccessors()
30:    branchDstInd ← 0
31:    for all curDstBB ∈ branchDstBBs do
32:        if curDstBB = prevTarget then
33:            ▷ make the destination pointed to by branchDstInd realTarget
34:            predecessor.setSuccessor(branchDstInd, realTarget)
35:        end if
36:        branchDstInd ← branchDstInd + 1
37:    end for
38: end procedure
```

Chapter 4

Decoupled Computational Pipeline as a Process Network

When multiple processes communicate through finite communication channels, deadlocks may happen. The pipeline of subgraphs we have generated can be seen as a network of communicating processes, and thus may become deadlocked under certain circumstances. To facilitate the analysis of our pipeline, we relate it to a different model of computation, Kahn process network (KPN), which provides the theoretical context for looking at these issues. We develop a systematic approach to find solutions ensuring liveness and correctness of our pipeline.

The KPN is an inherently parallel model of computation where processes communicate with one another through unbounded FIFO channels. The processes cannot test for availability of data tokens in the channels without consuming them. If a channel is empty the process blocks as it reads from the channel. Writing to the FIFOs, however, is non-blocking. In addition, each FIFO can only be consumed by a single process, and be written to by a single process. The KPN model is deterministic in the sense that the scheduling of the process execution does not alter the final results. This gives great flexibility in implementing/controlling individual processes. By relating our pipeline to KPN, or more precisely, observing its deviation from a pure KPN, we can understand the freedom we have and the constraints we are subjected to in our synthesis flow, where the correctness and liveness of the final implementation is required. Note the required absence of deadlock does not apply to the eventual state of our pipeline, when the execution finishes. As mentioned in section 3.3, the execution of the pipeline completes when all subgraphs are either done running or waiting for inputs, which is a form of deadlock. This is expected and not the subject of our discussion in this chapter.

4.1 Memory and Fanout Process

A characteristic of KPN is that all data exchange are performed over the FIFOs and the notion of a shared global memory does not exist in the model. Previous works converting sequential programs to process networks [75][76] were able to transform the original memory operations to explicit data IOs at the boundary of the generated hardware as the targeted kernels are highly regular. In our case however, the synthesized pipelines may have irregular memory access patterns and are often used as accelerators along side a general purpose processor, the memory model needs to be preserved. It is therefore necessary to conceptually translate the behavior of processes accessing memory to that of inter-process communication.

In our system, the memory subsystem, including the associated interconnects, exposes multiple input and output ports to the other hardware modules. Just like other processes in the network, through these ports, it consumes and produces tokens. In the case of load operations, addresses generated by the other processes are read by the memory, and the corresponding data are returned. For store operations, on the other hand, the memory consumes an extra data token while returning an acknowledge token. This is certainly not sufficient to allow us to model it as a conventional process in the network. In particular, while a normal process blocks when reading from an empty channel, the absence of requests at one input port never prevents the memory from serving requests from other ports. In other words, it can choose which port to read depending on the availability of input tokens. In YAPI [119], a "select" function is added as an extension to the Kahn process network to support non-deterministic events. The behavior of the shared memory subsystem can be accommodated with this primitive as well.

With the introduction of "select", the semantics of the original KPN is violated. The memory subsystem can produce different output sequences depending on the relative timing of appearance of token at its different input ports. This implies the scheduling of other processes would have an effect on the final result of the execution, the system is no longer deterministic. However, as described in section 3.2, we partitioned the memory address space and added special memory-dependency edges between instructions. Consequently, the timing of memory requests coming from the normal processes are coordinated by the sending and receiving of special tokens among themselves. This added constraint effectively eliminated any input token sequences inconsistent with the behavior of the original sequential program, while presenting the deviant memory "process" a strict subset of the possible inputs. With the input streams across different ports adhere to the read-after-write, write-after-read and write-after-write requirements encoded in the memory access instructions of the original program, the system becomes deterministic again.

In the denotational semantics of KPN, each process is defined as a function mapping potentially infinite input streams to output streams, independent of the relative timing of tokens in the streams. After eliminating the possibility of the conventional processes generating request sequences which violates memory dependency constraints, for all practical purposes, the memory subsystem can be seen as exactly that. We have visualize the concept of this memory process in figure 4.1. For the *Select* operation in the process, at any point

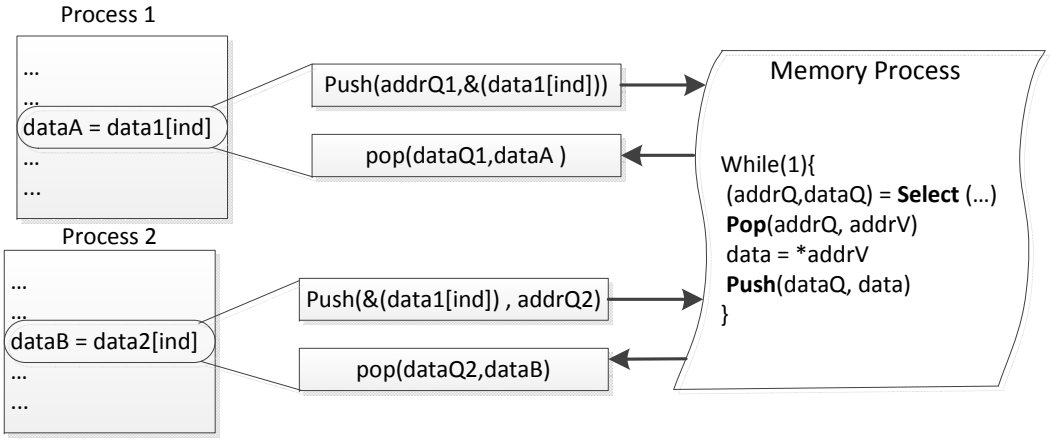


Figure 4.1: Modeling Memory Access as Inter-process Communication

in time, the selectable incoming requests can be served in any order without causing any difference in the final result.

Another requirement for KPN is the singularity of producers and consumers for each channel. However, in a typical program represented in single static assignment form, value defined by an instruction can be used by multiple consumer instructions. To make it compatible with the KPN model, explicit fanout processes are added. For every 1-to-n def-use relationship in the SSA, a fanout process with one input and n outputs is added. The process duplicates every input token n times and sends one to each of the consumer processes. As shown in figure 4.2, a fanout process, just like other processes created from partitioning CDFGs, consists of a sequence of instructions, some of which consume tokens while others produce and transmit tokens. With the addition of these processes all the channels in the network are one to one and we have a conceptualization of the entire computation described by the original program in terms of a KPN, with the addition of a more flexible memory "process".

4.2 Bounded Execution of the Process Network

In real implementations of a KPN, it is impossible to have unbounded communication FIFOs between processes. In [120], KPNs are categorized into strictly bounded, bounded and unbounded ones. A KPN is strictly bounded if and only if any execution of the network requires bounded space. It is bounded if and only if there are some execution requiring bounded space while it is unbounded if and only if any execution requires unbounded space. Methods to find bounds for communication channels in KPNs were discussed in [120][121]. In general, whether a KPN is bounded is undecidable [122], but for our process networks, which are generated from sequential programs expressed in single static assignment form, we

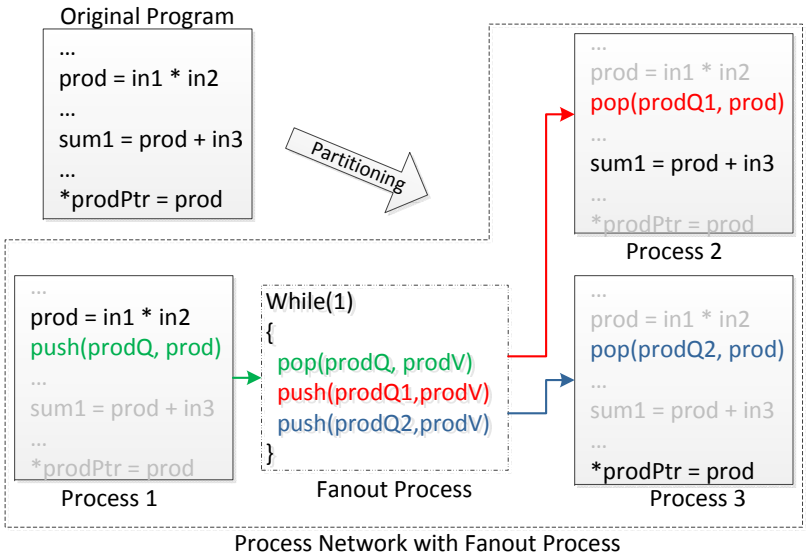


Figure 4.2: Addition of Fanout Process while Partitioning

can analyze the boundedness of the FIFO channels in various scenarios. Note here we try to deduce properties applicable for networks created by any arbitrary partitioning algorithms applied to the original CDFG. Thus they also apply to the pipeline generated using method outlined in chapter 3, which is just one specific instance of all possible KPNs producible from a sequential program.

More specifically, given a sequential program S in single static assignment form, we can partition the program into a process network N, consisted of a set of normal processes P, a set of fanout processes F, and a memory process M. Also, as mentioned in chapter 3, if an instruction is assigned to p and it is taking in tokens from instructions in other processes, a placeholder instruction (and its owner basic block) is instantiated local to p, whose sole purpose is to receive the actual token from the producer/fanout process. For M, it can be seen as executing a mirroring of each of the memory access instructions distributed among P, producing tokens for the corresponding p to consume.

Definition Let I be the set of instructions in the original S. For $i \in I$, each execution of i is defined as a instruction instance ${}_{(k)}i$, where $k \in \mathbb{Z}$. In $p \in P$, the set of instructions is I^p. Within I^p, each "push" and "pop" primitive added is associated with one i in S, i.e. the instruction generating the tokens to be transmitted. Every execution of these ${}_{push}i^p \in_{push} I^p$ or ${}_{pop}i^p \in_{pop} I^p$ is defined as ${}_{(k)push}i^p$ or ${}_{(k)pop}i^p$, corresponding to ${}_{(k)}i$. Similarly, for all other instructions in I^p, denoted as ${}_nI^p$, each ${}_ni^p$ is linked to a specific i and its instances are denoted as ${}_{(k)n}i^p$, also associated with ${}_{(k)}i$.

Note for each ${}_{(k)push}i^p$, there can be one or more ${}_{(k)pop}i^p$ in p' where $p' \in P \setminus p$. They are connected either directly or through $f_i \in F$.

Lemma 1 *Any instance of network N is bounded with FIFO space required for each channel = 1.*

To prove Lemma 1, we can constructively create an execution schedule H which does not overflow the FIFOs of size 1. This can be performed by following the original program order in S. Whenever a instruction instance i_k of i is executed in S, we find its associated set of instruction instances in all of P, distributed among one or more processes. For each element ${}_{(k)}j$ in this set:

- If ${}_{(k)}j \notin_{(k)push} I^p \cup_{(k)pop} I^p$, we can just schedule ${}_{(k)}j$.
- If ${}_{(k)}j \in_{(k)push} I^p$, i.e. this is a "producer" instruction, ${}_{(k)}j$ is scheduled and then the instructions in f_i are scheduled (assuming there is fanout), followed by all the other "consumer" instruction instances ${}_{(k)}j' \in_{(k)pop} I^{p'}, p' \in P \setminus p$.
- If ${}_{(k)}j$ accesses memory, then the instructions in M is immediately scheduled to serve the request before everything else.
- If ${}_{(k)}j \in_{(k)pop} i^p$, do not schedule it directly, wait for the producer instruction and f_i to be scheduled.

In essence, for every instruction executed in the original program, we schedule the process network's instructions derived from it in H. If these instructions are involved in communication, the source of the data tokens is scheduled first, immediately followed by the token sinks. It should be apparent that only one slot is needed in each communication channel for the process network to execute, since the produced tokens are promptly consumed and no backlog of tokens would occur in the FIFOs.

Definition Let the schedule of instructions in each $p \in P \cup F$ be $G(p)$, we say $G(p)$ is locally consistent with H if and only if ${}_{(x)}j' \prec_{(y)} j''$ in $H \Rightarrow {}_{(x)}j' \prec_{(y)} j''$ in $G(p)$, $\forall j', j'' \in I^p$, $x, y \in \mathbb{Z}$. Additionally, if we have a schedule H_s for a set of instruction instances which can belong to different processes, we say H_s is globally consistent with H if and only if ${}_{(x)}j' \prec_{(y)} j''$ in $H \Rightarrow {}_{(x)}j' \prec_{(y)} j''$ in H_s where $x, y \in \mathbb{Z}, j' \in I^{p1}, j'' \in I^{p2}, p1 \in P \cup F, p2 \in P \cup F$.

It should be apparent that if instances of a producer/consumer instruction pair around a FIFO are globally consistent to H, we do not need to have FIFOs with size greater one for our execution. Meanwhile, as H is derived from the original program order of the sequential program S, all locally consistent schedules $G(p)$ do not violate data dependencies imposed by S.

4.3 Artificial Deadlock in Process Network

In actual implementation of KPN where FIFOs have fixed sizes, blocking writes are introduced so that processes block when trying to write to a full FIFO. This induces artificial

deadlock as described in [121]. When an artificial deadlock occurs, a circular dependency is formed among processes, none of whom can make progress. At least one of the processes in the cycle is blocked on write – therefore the deadlock is "artificially" introduced by the size limit of FIFOs. Examples of this is shown in figure 4.3.

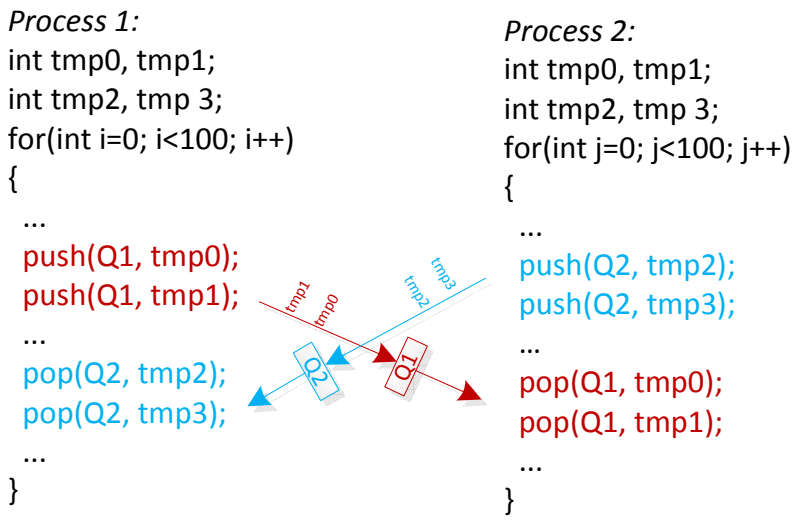


Figure 4.3: Limited FIFO Size Causes Artificial Deadlock

Being a network of sequential processes and memory connected together by bounded FIFOs, the computational pipeline synthesized may experience artificial deadlocks as well. The interaction between the sizing of the FIFOs and the occurrence of deadlocks needs to be analyzed to ensure the liveness of our pipeline.

Lemma 2 *Assuming all FIFOs are of size one and blocking write, as long as* $G(p), \forall p \in P \cup F$ *are locally consistent with* H*, artificial deadlock will not occur in* N*.*

Assume there is an artificial deadlock, we can go around the dependency cycle and examine the blocked processes $P_b \subseteq P \cup F$. For a process p_b to be blocked at instruction instance ${}_{(k)}j$, ${}_{(k)}j \in_{(k)push} I^p \cup_{(k)pop} I^p$ is either reading from an empty FIFO, or writing to a full FIFO. For the former case, the FIFO is empty because the producer of the token

$_{(k)}j' \in_{(k)push} I^{p'}$ cannot execute, which indicates an earlier instruction instance $_{(m)}l$ in $G(p')$ is blocked. For the later case, the FIFO is full because the consumer $_{(k-1)}j'' \in_{(k-1)pop} I^{p''}$ of the earlier token produced by $_{(k-1)push}j$ cannot execute, which also indicates an earlier instruction in $G(p'')$ is blocked. Apply this reasoning recursively around the circle of dependencies, we have a chain of instructions instances $_{(k)}j \succ_{(m)} l \succ ... \succ_{(k)} j$ or $_{(k)}j \succ_{(k-1)} j'' \succ ... \succ_{(k)} j$ in H, as each pair in the chain comes from a schedule locally or globally consistent with H. This chain is self-contradictory and therefore the scenario for artificial deadlock can never occur.

Note each memory operation is just one instruction in $G(p)$, the access is therefore modeled as an atomic action – the sending of the address is immediately followed by the receiving of response, both of which happen in the same process p. The memory process thus always produces response tokens some processes are waiting to consume, precluding the possibility of it being blocked on write. It therefore will not be part of the dependency cycle causing the deadlock. However, when we optimize the memory access mechanism by pipelining the requests, this may no longer be the case. We will elaborate the implication of this change later in section 4.4.

As each $G(p), p \in P \cup F$ is locally consistent with H, which is derived from the program order of the original S, we can conclude that if each of the generated subgraph is executed strictly according to the original program order, we will have a pipeline free from artificial deadlock. This will guarantee, for instance, when processors are used as the execution substrate, the network will produce the correct result as each process is executed according to program order. On the other hand, when HLS is used to create hardware accelerators from them, this guarantee may not hold anymore since aggressive parallelization and reordering of instructions would violate the consistency between $G(p)$ and S. We thus need to examine the effect of instruction reordering on the liveness of our pipeline.

4.4 Liveness in HLS-generated Computational Pipeline

Figure 4.4 shows a simple example of a otherwise deadlock free network becomes problematic when the instructions are statically reordered in individual processes by HLS.

Figure 4.4(a) shows how the example sequential program is decoupled into two independently running processes. Due to the size limit of the FIFOs, which is equal to 1 here, the execution of the producer instruction ("push") is dependent on the completion of the previous instance of the consumer instruction ("pop"). When both *Process 1* and *Process 2* are running according to the original program order, there is no cycle of dependency, and therefore no deadlocks during the execution of the network, as illustrated by 4.4(b).

However, in 4.4(c), a common HLS technique, loop pipelining, is applied to *Process 1*, where iterations of the loop are aggressively overlapped. Every cycle, a new iteration is started. Because of the long latency of load from *in[i]* and the multiplication, the sending of

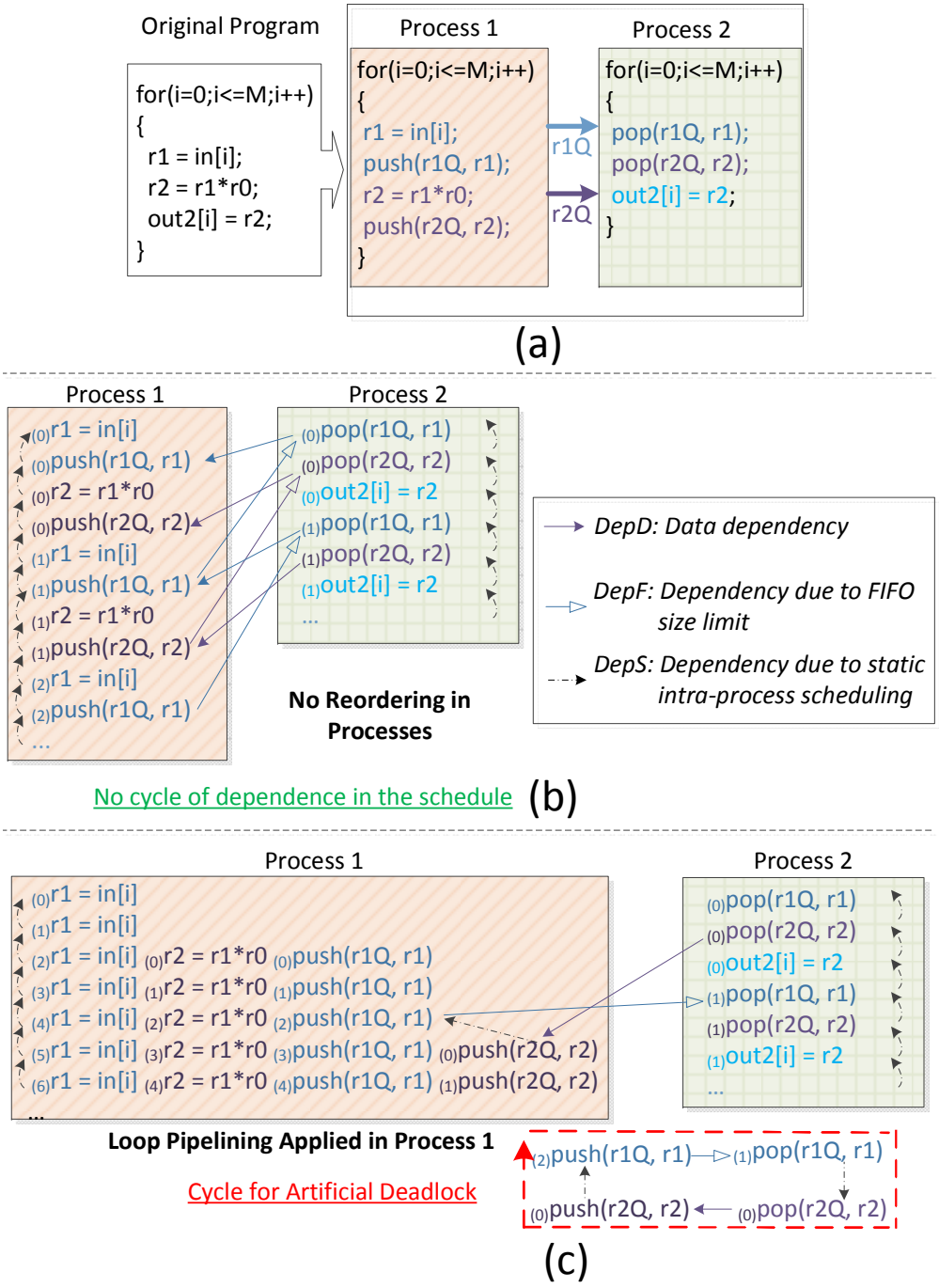


Figure 4.4: a) partitioning of the original program into two processes; b) each process executes according to the program order; c) instructions in process 1 are reordered statically

the first *r2* token, $_{(0)}$push(r2Q,r2), is scheduled after many instruction invocations for later iterations. It can not execute until after $_{(2)}$push(r1Q,r1), whose writing to the FIFO r1Q cannot happen unless the previous token has been consumed by $_{(1)}$pop(r1Q,r1). Because of the scheduling in process 2, this read from r1Q occurs after $_{(0)}$pop(r2Q,r2), which cannot complete unless $_{(0)}$push(r2Q,r2) can send the token. An artificial deadlock is therefore created.

In this particular example, if the size of r1Q is increased from 1 to 2, then the execution of $_{(2)}$push(r1Q,r1) would be dependent on $_{(0)}$pop(r1Q,r1) instead of $_{(1)}$pop(r1Q,r1). The channel can now buffer the tokens produced by $_{(0)}$push(r1Q,r1) and $_{(1)}$push(r1Q,r1) before $_{(0)}$pop(r1Q,r1) needs to be invoked to create empty slot. With this increase in buffer space, the cycle of dependence is broken as there will be no path of dependence from $_{(0)}$pop(r1Q,r1) to $_{(0)}$push(r1Q,r1) The deadlock no longer exists.

Indeed, this is the classic approach for resolving artificial deadlock [120]. When artificial deadlock is detected, the process which is write blocked is found, the full FIFO it tries to write to is given more capacity. However, in a network implemented in hardware, the FIFO size cannot be easily increased. To determine the needed buffer space in the channels *a priori*, simulations are sometimes used [123][124]. It relies on using representative dataset as input and for certain type of problems, simulation results may not provide any guarantee, especially if the control flow is runtime data dependent. In this section, we want to analyze the interaction between FIFO sizing and intra-process instruction scheduling, so we can better understand their joint effect on the liveness of the process network.

Artificial Deadlock Detection without Simulation

In our networks derived from sequential programs, given each process' schedule and each FIFO's size, it is possible to determine if a potential artificial deadlock can arise without running simulation. As demonstrated in figure 4.4, we can add *DepS* edges between instruction instances belonging to the same process, and *DepF* edges between producer/consumer instruction instances writing/reading from the same FIFO. Note for a FIFO of size s, *DepF* edges are added between $_{(k)push}j$ and $_{(k-s)pop}j$ to indicate that the producer instruction instance is disabled until the consumer instruction instance is invoked. The graph thus obtained can be checked for cycles, which indicate the occurrence of artificial deadlock.

The schedule for each process may contain potentially infinite instruction instances, resulting in a graph with infinite number of nodes, which makes it impossible to guarantee the absence of artificial deadlock. However, when we use HLS tool to map each process, a short, repeatable schedule is generated and we can represent the dependencies between instruction instances using a manageable sized graph. Figure 4.5 shows a more concise representation of the schedule we have in figure 4.4(c). Various dependency edges are associated with weights to represent the difference between the subscripts of the pair of instruction instances. For instance, as $_{(2)}$push(r1Q, r1) is scheduled before $_{(0)}$pushr2Q, r2 in process 1, the edge from pushr2Q, r2 to push(r1Q, r1) carries a weight of 2, meaning the occurrence of $_{(x)}$push(r1Q, r1) depends on the completion of $_{(x+2)}$push(r1Q, r1). Similarly, when the channel between

push(r1Q, r1) in process 1 and pop(r1Q, r1) in process 2 is of size one, ${}_{(x)}$push(r1Q, r1) depends on the completion of ${}_{x-1}$pop(r1Q, r1), which is represented by the -1 weight of the edge representing this dependency. The edges denoting data dependencies between processes are always of weight 0, as the end points are both associated to the same instruction instance. This graph is named *precedence graph* as it captures the necessary ordering of instruction instances, allowing for the analysis of deadlocks.

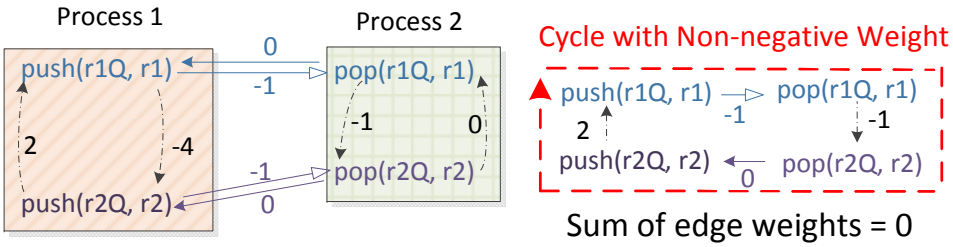


Figure 4.5: Deadlock Detection using Graph with Weighted Edge

To shrink the size of the precedence graph, we have taken advantage of the transitive property of dependencies to reduce the number of the *DepS* edges in the graph. Also, as only the scheduling of producer/consumer instructions matters for deadlock detection, we have trimmed the graph by omitting any *DepS* edges having end point $j \notin_{push} I^p \cup_{pop} I^p$. In figure 4.5, for instance, $out2[i]$ is not included and all its incoming edges are redirected to the nearest communication instructions.

Now, if a non-negatively weighted cycle is found within the precedence graph, we have an artificial deadlock. A cycle in the graph means we have an instruction instance ${}_xj$, whose occurrence depends on the completion of ${}_{x+w}j$, w being the weight of the cycle. With w being greater or equal to zero, ${}_xj$ cannot happen until itself or a later instance happens – we have a deadlock. The computation to find deadlocks may potentially be expensive. In the worst case, the number of simple cycles in the graph can be exponential in $|V|$, the number of nodes in the graph. Practically, however, the number of nodes involved are approximately the same as the number of channels needed, and the graph is rather sparse in the connectivity between these nodes. Methods such as [125] can be used for efficient enumeration of the cycles and the subsequent identification of the deadlocks.

Resolving the deadlock involves choosing one of the *DepF* edges in the cycle and add enough slots (making the weight more negative), such that the sum of the edge weights becomes negative. For one particular cycle, the *DepF* edge whose channel has the smallest width can be selected, so the cost of adding the extra slots is minimized. Of course, as multiple non-negatively weighted cycles can be present simultaneously, this heuristic does not necessarily produce the globally optimal solution, though it suffices for our use cases.

It is worth noting that the weights assigned to edges correspond to the largest differences in instance subscripts, i.e. the furthest an instruction is running ahead of its dependents.

In the presence of branches, it is possible that an instruction does not get executed every loop iteration. The subscripts assumed by the instruction instances do not take this into account. In figure 4.6 for instance, $_{(2)}$push(r2Q, r2) gets executed despite $_{(1)}$push(r2Q, r2) never appearing in the schedule of process 1. Thus under our scheme, the subscripts assigned to instruction instances are just the iteration numbers of the enclosing loop in the original program. Note in the literature about loop dependency analysis [116], the term iteration number is generally associated with analyzable loop index, with bounds and regular change steps. Here, however, we just use the term to represent the simple ordering of iterations, with or without actual loop indices.

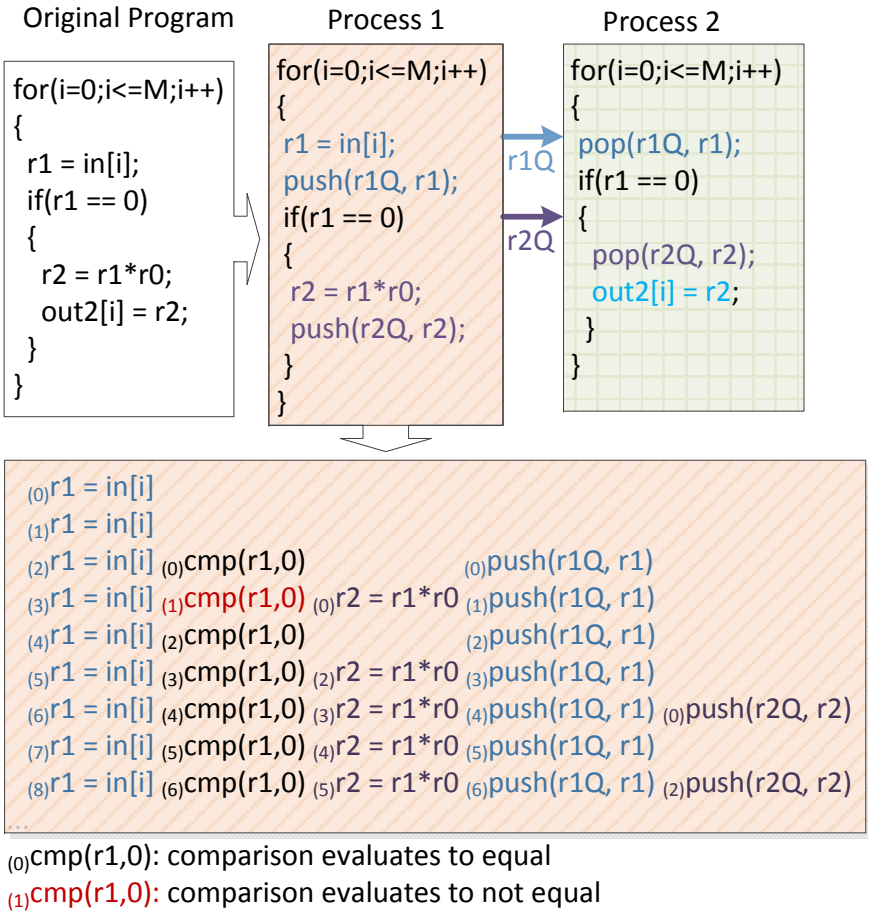


Figure 4.6: Execution Schedules and Subscript Assignment in the Presence of Branches

The method we have formulated so far can be easily applied for deadlock detection and resolution of a single level loop. To generalize this method for loop nests of multiple levels, we associate each instruction instance with a vector of iteration numbers, with the earlier elements corresponding to the outer loops. Subsequently, for the weight assigned to each of the edges, instead of a single number, a vector is used. The number of elements in the

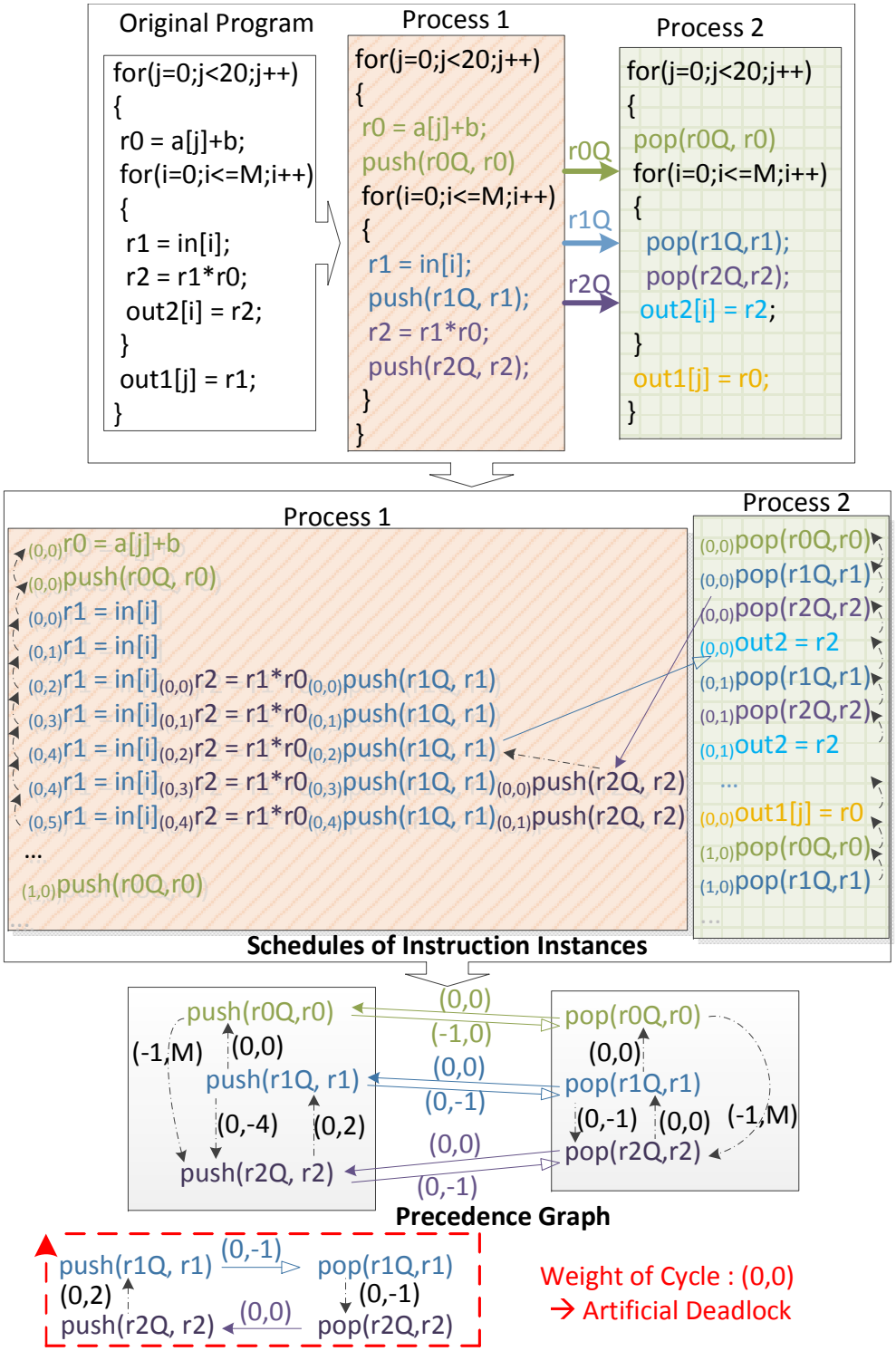


Figure 4.7: Multi-level Loop Nest Deadlock Detection

vector corresponds to the number of levels in the loop nest. An example of this is shown in figure 4.7, where a two level loop nest is decoupled into two processes and independently scheduled.

The vector weights assigned to the edges between instructions of the same level in the loop nest are really the same as in the single level loop example, except the padding of zeros. A more interesting case is the *DepS* edge from push(r0Q,r0) to push(r2Q,r2), which goes from an outer loop instruction to an inner loop instruction. The last components of the subscript vectors of push(r0Q,r0) instances are always 0 as the instruction does not execute in the innermost loop. The subscript vectors of push(r2Q,r2) instances, however, can take a value up to M for its right most element, as the inner most loop has an iteration count of M. Thus the weight $(-1, M)$ means that only after the Mth occurrence of push(r2Q,r2) in the previous iteration of the outer loop, another instance of push(r0Q,r0) can be scheduled within process 1.

In this enhanced precedence graph, every cycle in the graph now has a weight sum which is also a vector. A deadlock is manifested as a cycle with weight being the zero vector ($\vec{0}$) or a vector whose first non-zero element (leading element) is positive. Similar to the case when we have scalar weights, these weight vectors indicate an instruction instance depending on itself or a later instance in the execution trace, and thus a deadlock. The deadlocks can also be resolved by finding the *DepF* edges, whose change of capacity makes all the cycles' weights to have negative leading element.

Definition When an artificial deadlock occurs, a cycle of dependency between instruction instances is formed. Define this cycle as C_v. Each instruction instance in C_v corresponds to one instruction in the precedence graph. Define the cycle formed in the precedence graph by the corresponding instructions as C_V, and the set of instructions as V.

As mentioned before, non-communicating instructions/instruction instances can be omitted in the graph without affecting the deadlock analysis, thus $V \subseteq_{push} I^p \cup_{pop} I^p$. As the nodes in C_v are instantiations of nodes in C_V, the edges in C_v are also instantiations of edges in C_V. The weights of these edges are the differences between the subscript vectors of the end-points in C_v, which are always lexicographically smaller or equal to the weights carried by edges in C_V.

Lemma 3 *If there is an artificial deadlock in the network, there will be a cycle in the precedence graph whose sum of edges is either* $\vec{0}$ *or a vector with positive leading element.*

We can take any point in C_v, and go around the cycle C_v to get a sequence ${}_{(X_1)}V^1$, ${}_{(X_2)}V^2$ ... ${}_{(X_n)}V^n$ where $V^1 ... V^n \in V$, $V^1 = V^n$ and $X_1 = X_n = X_1 + (X_2 - X_1) + (X_3 - X_2) ... (X_n - X_{n-1})$. The sum of edges in $C_v = (X_2 - X_1) + (X_3 - X_2) ... (X_n - X_{n-1}) = \vec{0}$ as $X_1 = X_n$.

If no two elements in C_v map to the same element in V, then the sum of edges in C_v is equal to or smaller than the sum of edges in C_V, which would be zero or lexicographically greater – we have a cycle in the weighted graph whose sum of edges is $\vec{0}$ or a vector with positive leading element.

Otherwise, we can have C_v containing two instruction instances ${}_{Y_1}U, {}_{Y_2}U$ mapping to the same element $U \in V$, and instruction instances on the path between ${}_{Y_1}U, {}_{Y_2}U$ are all instances of unique instructions in $W_1 \subseteq V \setminus U$. Now we can construct a cycle C_{W_1} of instructions using W_1, and the weight of this cycle is the same or greater than the weight of the path between ${}_{Y1}U$ and ${}_{Y2}U$. This is essentially the case we have just discussed, where only single instances of the instructions are present in a cycle. We can decompose C_v further, and obtain more of these cycles, $C_{W_1}, C_{W_2} ... C_{W_n}$, the sum of their respective weights must be equal to or greater then the weight of C_v. Since weight of C_v is $\vec{0}$, these cycles can either all be of weight $\vec{0}$, or at least one of them will have positive leading element. This process is illustrated in figure 4.8 for a single level loop. In essence, if a cycle (of instruction instances) in a graph carries a weight of $\vec{0}$, then the weight of one of its component simple cycles must have a positive leading element/be $\vec{0}$. The weights of the simple cycles here provide lower bounds for the weights of cycles of instructions in the precedence graph.

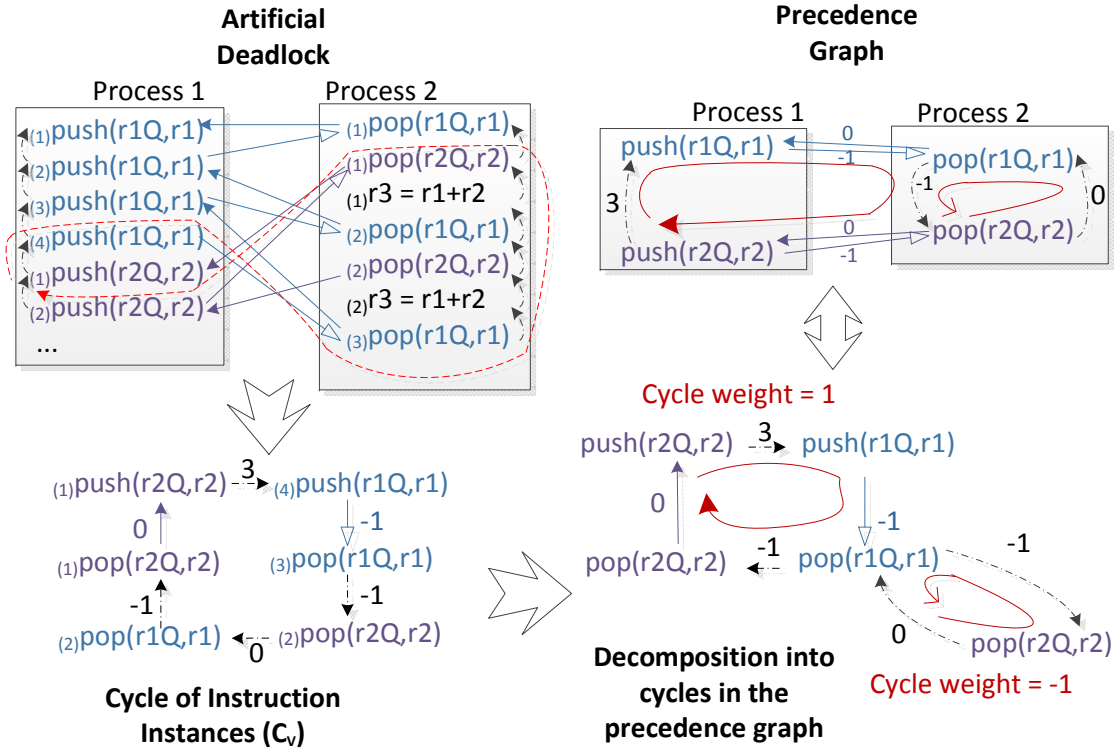


Figure 4.8: Decomposition of Cycle of Instruction Instances

As lemma 3 is proven to be true, the absence of cycles of weight $\vec{0}$ or with positive leading elements guarantees the the absence of artificial deadlock (the contrapositive of lemma 3).

Statically Unresolvable Artificial Deadlocks

Using the analysis framework we have formulated, we can foresee situations where artificial deadlocks may always occur, regardless of how much space is statically assigned to a FIFO channel. Figure 4.9 shows one example where no guarantee of deadlock freedom can be statically obtained. Here, the outer loop consumer instruction are rescheduled to before the start of the inner loop, while the corresponding producer instruction executes after the completion of the inner loop iterations. As the number of iterations of the inner loop is statically unknown, there will always be a *DepF* between a pair of the *push* and *pop* instruction instances (and therefore a cycle of dependence), regardless of how large the FIFO size is.

In the precedence graph, the number of iterations is represented symbolically and the *DepS* edge between the outer loop and inner loop producers carries a weight of $(0, i)$. Assuming FIFOs of size 1, a cycle of weight $(0, i-1)$ is found. Without knowing the bound of i, there is no way we can add enough space to the *DepF* edge between the producer and consumer of *data* such that the cycle weight has negative leading element.

The actual computation is shown in this example to demonstrate the validity of the reordering. There is no local data dependency violated, yet artificial deadlocks are created in the network. To prevent this type of unresolvable deadlocks, we can impose constraints on the reordering of instructions performed by the HLS tools. The starting point of instruction rescheduling is always the original program order, or more specifically the locally consistent $G(p)$. It was proven that if every process is run according to $G(p)$, there will be no artificial deadlock. Thus as long as we do not introduce symbolic vectors into the the weights of the cycles in the precedence graph, no unresolvable deadlock would be introduced.

A sufficient condition, which ensures no new symbolic vectors are introduced into the cycle weight, is to disallow outer loop instruction to be locally reordered with respect to inner loop instruction. According to our scheme, all the simple paths whose weights contain symbolic variables must be ones going from an outer loop instruction to inner loop ones. Meanwhile, simple paths between instructions of the same loop level, if not going through an inner loop instruction, are always free from unknown variables in their weights. Adhering to our proposed constraint essentially means we are only reordering instructions between which the simple paths are of statically known weights. Consequently, no new symbolic variable would be added to the cycle weight after the reordering. An example is shown in figure 4.10. Note this condition is not a necessary condition, in figure 4.9, if the sender of *outInd* is also scheduled to before *loop_inner*, we are not going to encounter the unresolvable deadlock. However, our proposed condition is easy to check against during HLS, and the liveness of the resulted hardware can readily be guaranteed.

Memory Access Representation

So far we have been modeling the memory operations as atomic actions. The process sending out memory requests waits for the responses before continuing. However, the sending of

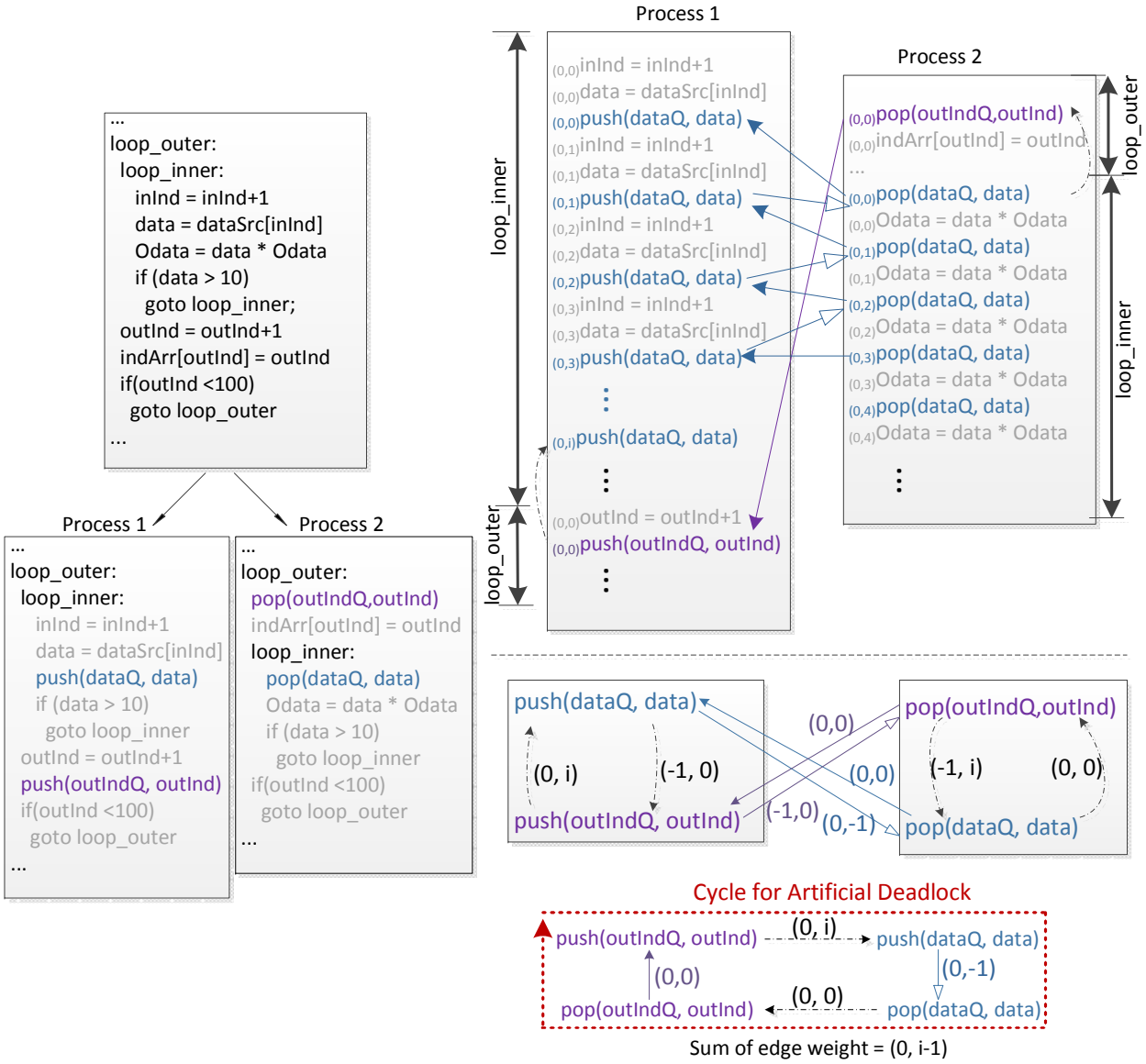


Figure 4.9: Rescheduling Creates Statically Unresolvable Deadlock

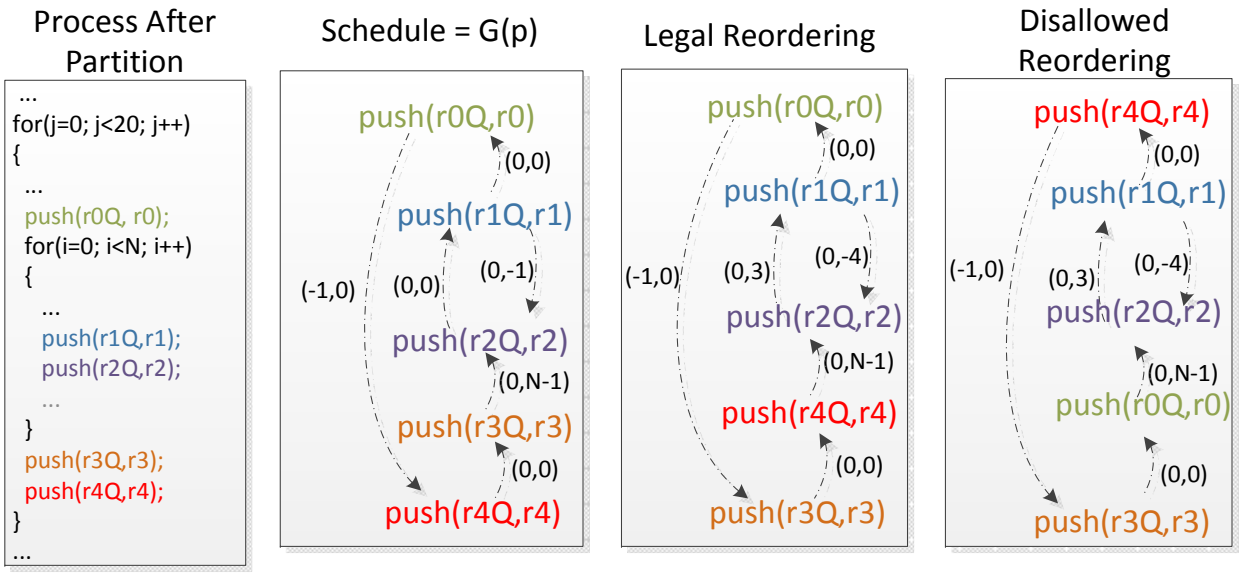


Figure 4.10: Safe Reordering of Instructions

requests and receiving of the responses can often be decoupled, as we have done in chapter 3. In the case when they are assigned to the same process, to accommodate the latency of data access, instructions can also be scheduled between them. Now when the responses are sent back from the memory, their designated consumers may not be ready to execute, and more buffering is needed. This behavior can be easily incorporated into our model.

As shown in figure 4.11, corresponding to each memory access, a *send_req/receive_resp* pair is created in the schedule. The reaction of the memory is modelled by a pop action responding to the incoming request, and a push action producing one response data token. As multiple *send_req*s are scheduled before *receive_resp*, i.e. due to aggressive loop pipelining, the memory subsystem must be able to buffer/serve multiple outstanding memory requests to prevent deadlock. In the example schedule, 4 *send_req*s are scheduled before the first *receive_resp*, while the memory subsystem (with the associated FIFOs) can only accommodate three, a deadlock is thus created. This is detected as the zero weight cycle in the precedence graph. Just like for normal processes, increasing the FIFO size resolves the issue. When the *send_req* and *receive_resp* are partitioned into two different processes, a similar graph can be created and the minimal amount of space for deadlock avoidance can be computed.

Another common optimization in memory access is the use of burst mode. As described in chapter 3, to concatenate memory requests across loop iterations into one burst access, the *send_req* operation is hoisted to one level out of the original loop. The *receive_resp*, meanwhile, remains in the initial loop level. Even though the memory "process" does not block on reads, if the response data it sends to other normal processes is not consumed promptly, it can blocked on writes. The head-of-line blocking can then create a dependency

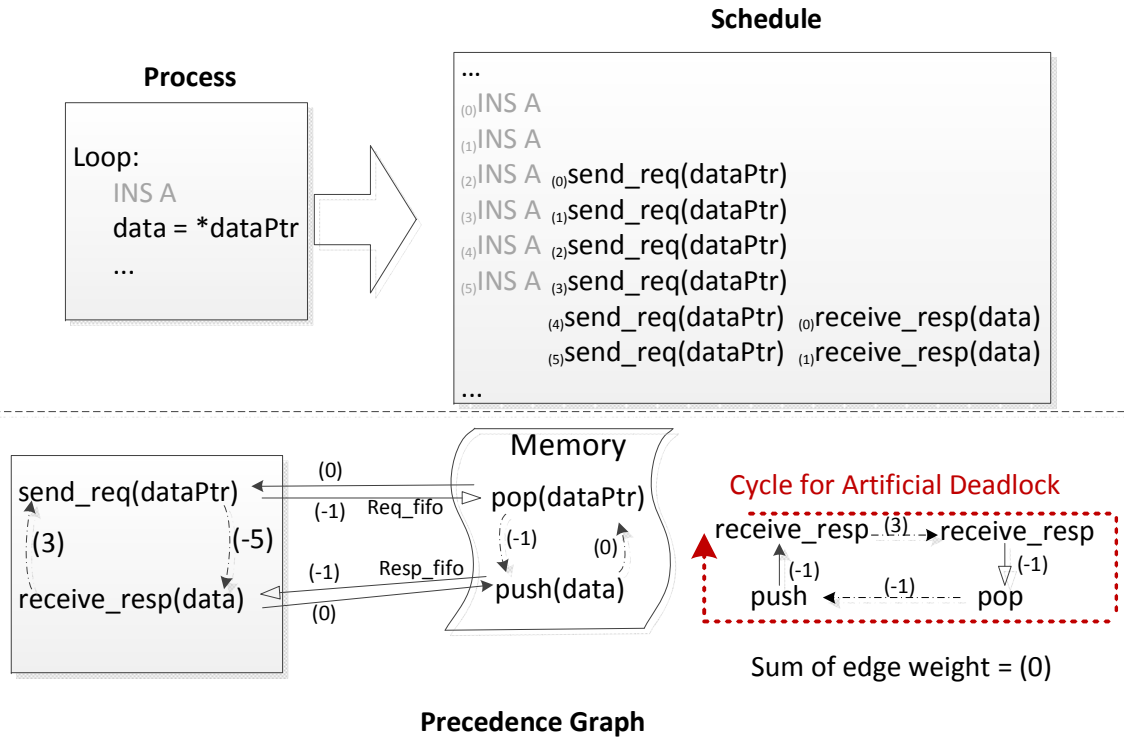


Figure 4.11: Deadlock Involving Memory Access

cycle, resulting in a deadlock. Our precedence graph can be used to compute the necessary buffer size to resolve this situation as well. Figure 4.12 shows how a precedence graph can be used to represent the transformed program. In (a), only a single process is involved, while (b) shows the precedence graph when the memory access is decoupled into two processes. In both cases, all the cycle weights have negative leading elements even when each FIFO is assumed to be of size one. No deadlocks would occur in this case. A more interesting scenario is when two memory accesses are converted to burst mode and the interaction between them causes deadlocks. This is illustrated in figure 4.13.

Besides the usual dependence edges, an extra *DepM* dependency edge is added between *push(data2)* and *push(data1)*. The symbolic variable B in the weight of this edge captures the head-of-line blocking due to large amount of response data, i.e. how many tokens have to be consumed/buffered before the response for the next request (*data*2) become accessible for the receiver. With this precedence graph, we can easily detect a potential deadlock in the implementation. To resolve this deadlock, the capacity of the FIFO between *push(data1)* and *receive_resp(data1)* should be increased to B, such that the weight of the deadlock-causing cycle becomes (0,-1). A similar buffering increase in the response channel for *data2* resolves the other remaining deadlock. In essence, the head-of-line burst mode responses need to be buffered entirely to ensure other subsequent *receive_resp*s are getting their data, and the whole datapath can then continue executing.

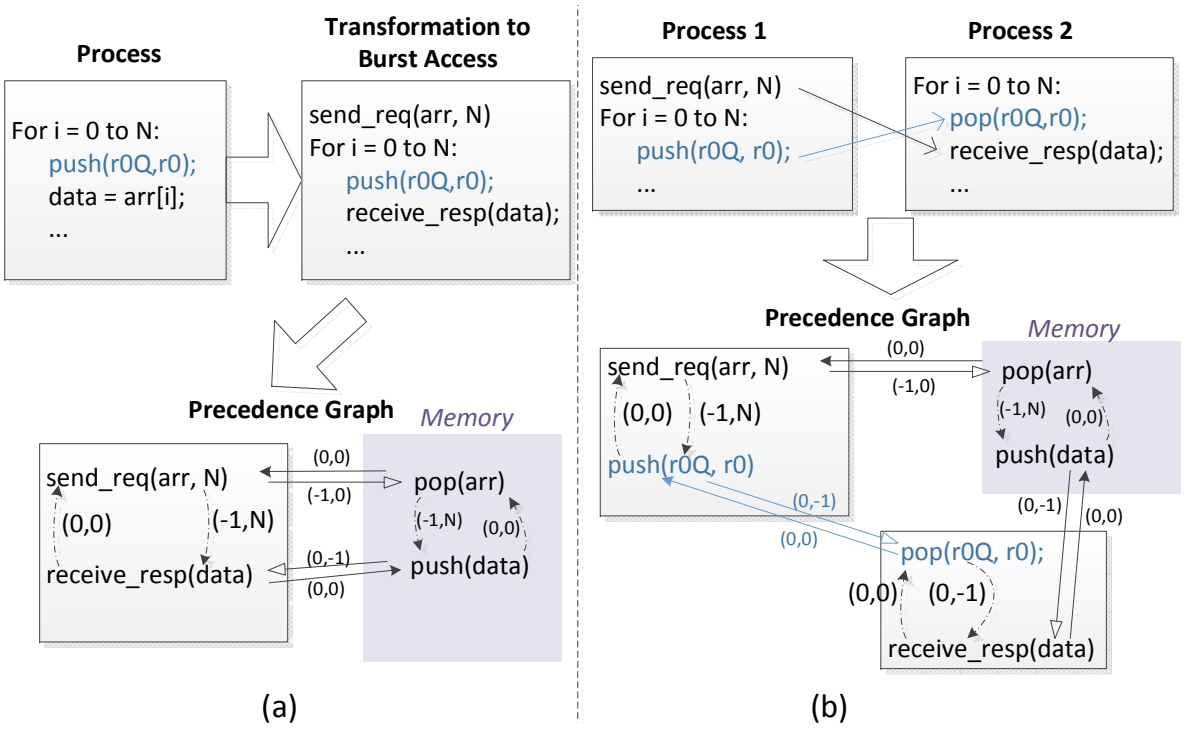


Figure 4.12: Example Precedence Graphs for Burst Mode Memory Access (a) send_req and receive_resp in a single process (b) send_req and receive_resp are decoupled into two processes

The actual value of B depends on a few factors. In chapter 3, we mentioned the burst access adapter which generates multiple burst mode requests according to the parameters of the *send_req* operation. For the process in figure 4.13, for example, two streams of requests S_1 and S_2, corresponding to *send_req(arr1,N)* and *send_req(arr2,N)* respectively would be created. Assume the arbiter for the memory subsystem implements a policy which can potentially take in and serve multiple (K) consecutive requests from one stream, we can compute B to be $BurstSize_{max} * K$. In the case of a round-robin schedule, $B = BurstSize_{max}$ and for our experiment platform which uses AXI bus protocal, this number is 256.

When multiple burst mode accesses are distributed across different decoupled processes, depending on the exact graph generated, the required buffer size can be a lot smaller and in certain cases, not at all affected by the size of the burst. This is shown in figure 4.14. By decoupling the memory accesses and duplicating the loop counters, we essentially have two separate data fetch engines running independently. As their schedules are completely separated, there are no cycles containing the *DepM* edge. Thus even with a buffer size of one in each channel, no deadlocks will occur.

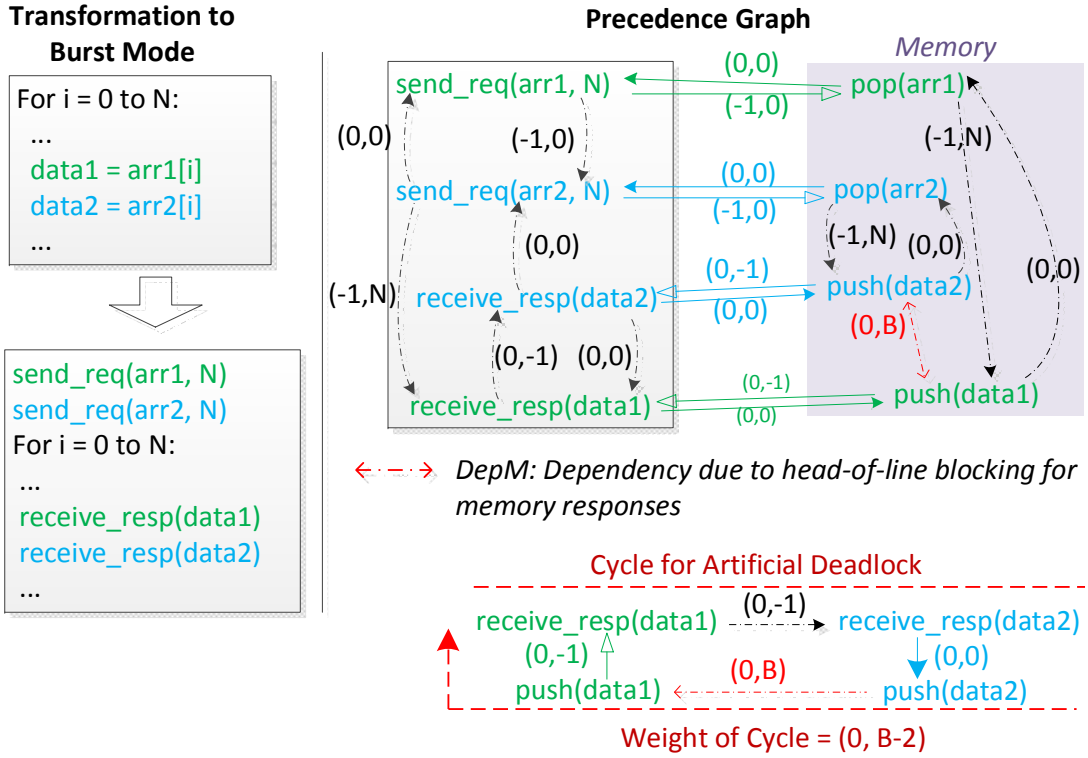


Figure 4.13: Deadlock Due to Interaction Between Two Burst Mode Memory Accesses

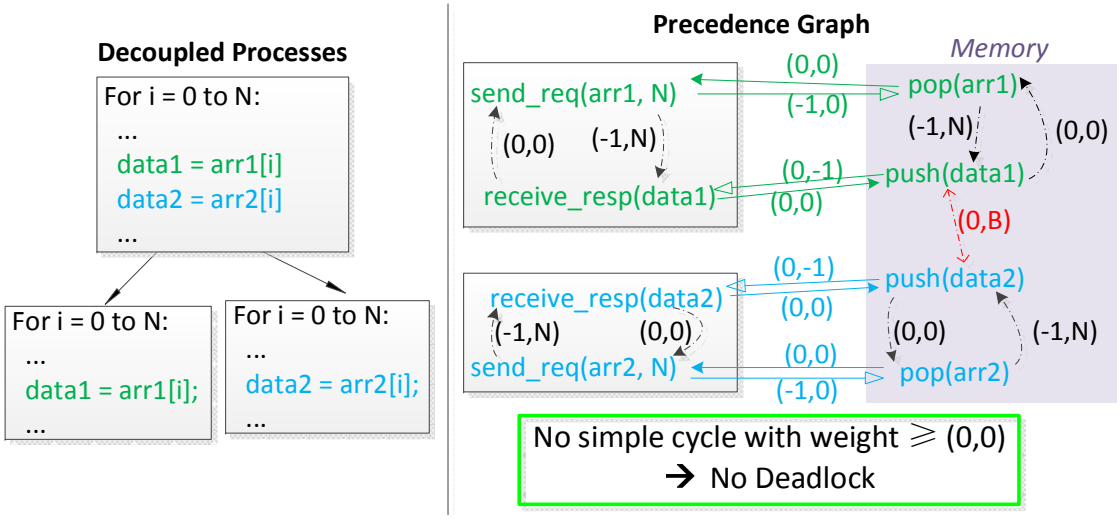


Figure 4.14: Burst Mode Memory Access by Decoupled Processes

4.5 Discussion and Future Work

In this chapter, we devised a proof for the absence of artificial deadlock in a pipeline of processes generated by our flow, assuming each of them is executed according to the original program order. A framework is then proposed to compute the minimum FIFO size needed to avoid deadlocks when the processes are synthesized into accelerators, which aggressively parallelize/reorder operations. This approach is proposed to reason about the properties of our accelerator pipeline, generated by splitting a single CDFG. However, it may also be applied to networks of processes created using other methods.

In our analysis, the construction of a precedence graph is based on 1) the scheduling of sender and receiver operators within each process, and 2) the size of communication channels between pairs of sender and receiver instructions in different processes. The local ordering of instructions can be easily extracted from any statically scheduled accelerators, but not all networks of processes satisfies the condition that each pair of sender and receiver has a dedicated communication channel. If each FIFO is read/written to by only one instruction locally, or more precisely, at a single time slot allocated in the local schedule, deadlock analysis can be performed statically. In commercial high level synthesis tools [29], for instance, computations can be manually structured into a series of functions which are annotated to allow the generation of a streaming pipeline. However, the problem of artificial deadlock is left to be addressed by the users as no systematic approach is provided by the vendor for sizing the FIFOs. If within every function, only one instruction accesses each FIFO interface, we can generate our precedence graph and perform deadlock analysis. Of course there are cases where this condition is not satisfied, when the user might need to rely on simulation or just devise special mechanisms to deal with potential deadlocks.

In many cases, the minimal FIFO sizes for deadlock avoidance may not be ideal for performance purposes. The decoupled computational pipeline works well when its member accelerators can execute out of sync with each other. Very small FIFO sizes potentially allow the lack of progress in one process to interfere with the other processes. One accelerator's slow rate of data consumption can exert back pressure which, if propagated quickly, can stall other parts of the pipeline – we are back to our original situation described in the beginning of chapter 3.

Of course given the decoupled pipeline architecture, we can explore the trade-off between area and performance in a much more flexible way. For instance, a unified schedule can choose to accommodate memory access latency by internally having more pipeline stages, which means adding register stages to a very wide datapath, incurring costly resource overhead. Our architecture, on the other hand, can dedicate buffer space to where decoupling is needed the most. To this end, our implementation can be tuned based on its run time behavior. Every FIFO can be instrumented with a histogram generator which tracks its occupancy. In the case of memory interface buffers, as the data count may not be directly available, we look at the percentage time when available data is not immediately pulled into the data path. These statistics can help us detect the point of inefficiency in the pipeline. More specifically, a FIFO channel fully occupied most of the time indicates the downstream/consumer acceler-

ator is not consuming tokens fast enough, where as a mostly empty FIFO reveals the relative inefficiency in the source of the data token. In some cases, certain stages in the pipeline can be made faster with more hardware invested for more aggressive parallelization or larger on-chip buffer, while in other scenarios, the information we gathered can help us reduce the amount of buffering in FIFO channels. A thorough exploration with this method is left to future work.

Chapter 5

Accelerator Generation and Integration Using Program Binaries

As usability remains to be one of the most significant obstacles for adoption of FPGA computing, in this chapter, we explore the possibility of a user transparent flow for mapping applications to systems with reconfigurable components. In contrast to previous chapters where source code in high level languages is used as input, here we try to only use the program binaries and their execution profiles as the design entries. The mechanism for integrating the accelerator back into the overall program execution is also discussed. From the users' perspective, a flow based on program binaries requires no source rewriting nor recompilation, the effort needed to take advantage of the reconfigurable platform is thus minimized.

With respect to the synthesis of the actual compute engine, we certainly take advantage of what we have developed in the previous chapters. The decoupled computational pipeline is again used as the architectural template to which the original software behavioral descriptions are mapped. The overall performance of the hardware implementation is thus a net result of exploiting pipeline, memory level and coarse grained parallelism in the application, as will be described in later parts of this chapter.

5.1 Profiling Program Execution with Binary Instrumentation

To perform various analysis on and eventually stitch the invocation of accelerators into the original program binary, we leverage the infrastructure developed in the field of binary instrumentation. In particular, Dyninst [126] provides great APIs for parsing, analyzing and modifying program binaries, both statically and dynamically.

To capture useful information during the execution of a program, Dyninst's dynamic instrumentation functionality is used. Its overall implementation is shown in figure 5.1. In Dyninst's terminology, the binary to be modified is called the "mutatee" while the program performing the instrumentation is the "mutator". Code that gets inserted into a program

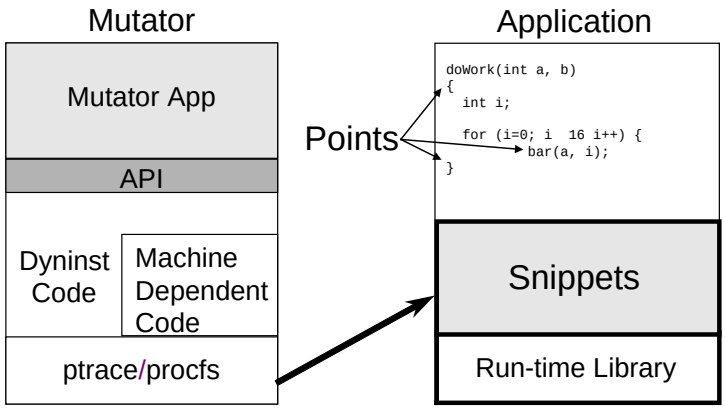


Figure 5.1: Dynamic instrumentation in Dyninst [127]

binary is organized into "snippets". The OS services used by debuggers (e.g., ptrace, /proc filesystem, etc.) are employed by the mutator to control process execution and to read and write the address space of the mutatee program. Dyninst also has a dynamic linked library which contains utility functions and two large arrays. With this library loaded into the mutatee application, the arrays can be used for dynamically allocating small regions of memory. The instrumentation variables and code are stored separately among the two arrays. The snippets are first translated into machine code in the memory of the mutator process and then copied into the array in the mutatee address space. The original code is then modified to branch to the newly generated code.

It's worth noting that the insertion and deletion of snippets can all be done dynamically during runtime. A mutator can attach to a running process to add information-gathering snippets, and after enough data is collected, remove all instrumentation and detach itself. The overhead of profiling is therefore only transient when applied to a long running program. Even though the performance may degrade to various extents depending on the actual snippets added, the application doesn't have to be interrupted.

Given Dyninst's capability, we can insert counters, track memory addresses referenced and log register accesses at points of interest in the program. It helps us identify the most frequently executed loop nests in the applications and allows us to capture all necessary information to perform the analysis and parallelization described in the later sections.

5.2 Characteristics of the Targeted Platform

The general approach of translating program binaries to accelerators can certainly be applied to various heterogeneous platforms. To justify some of the design decisions made in our flow, the assumed platform characteristics need to be outlined first. In section 2.3, a whole array of machines with reconfigurable components were examined and there is a wide

spectrum of configurations when it comes to how tightly integrated the reconfigurable fabric is to the processor. In this work, we focus on the part of the space where the general purpose processor and the reconfigurable component are loosely coupled. Instead of being a functional unit in the processor execution pipeline, the FPGA is used as a coprocessor for which communication and management is assumed to be expensive. Meanwhile, the capacity of the reconfigurable fabric can potentially be greater as it is not constrained in any way by the pipeline of the associated processor. Most of the off-the-self systems currently available or in development [128] fall into this category. As the costs of semiconductor manufacturing become prohibitive, special processors with modified pipeline accommodating reconfigurable functional units are less likely to be built and offered commercially, as compared to systems with conventional processor and FPGA integrated at either chip, package or board level.

Another important factor to consider is how sharable the memory is between the CPU and the FPGA. One possible configuration, as represented by the zynq SoC, has the CPU's address space shared with the programmable logic. The physical addresses used to access the memory are identical, whether it comes from the CPU or the FPGA. Of course in the presence of virtual memory, address translation needs to occur before the requests are forwarded to the memory subsystem. On the other hand, there are platforms where the FPGA has its own memory space which is explicitly populated with the working dataset before the activation of the accelerator [129]. It's worth noting that this difference in programming model is orthogonal to the actual physical configuration of the platform. For instance, CPU and FPGA located in two different sockets can have shared address space while a system based on a single chip SoC may associate the FPGA with a separate DRAM interface which are not directly accessible by CPU. Given the flexibility of FPGAs, there are certainly ways to create a layer of abstraction conforming to either one of these schemes, regardless of the original expected usage model. In this work, we assume a shared memory space between the CPU and the FPGA, which is one of the properties of the experiment platform we use, though a mechanism for explicit data movement applicable in non-shared memory system is also discussed in section 5.7.

5.3 Acceleratable Regions In Program Binaries

The trade-off between communication efficiency and capacity in the reconfigurable components of the systems dictates the granularity of the accelerators to be synthesized. Given the loose coupling between the programmable logic and the processor, while it is still possible to create accelerators for small windows of dynamic instruction stream, the cost of frequently controlling and communicating with these tiny hardware engines would nullify any performance gain achieved. It is ideal to have a relatively large chunk of computation handed off to the FPGA, which then works independently with little intervention from the CPU, before signalling the completion of the task. The natural targets for our flow are therefore loop nests in the program binaries. There may be cases where code segments are shared by multiple loops, making it harder to statically carve out the best code region for acceleration. For

these scenarios, we can leverage runtime profile of the program to find the prevalent loops and extract single-entry-multiple-exit loops which can undergo further optimizing transformations. Meanwhile, due to the presence of statically unresolvable control flow, e.g. indirect jump, not every part of the program can be analyzed. Several techniques were proposed to tackle this in dynamic binary translation [130], which we can potentially adopt. However, we do not aim to have complete coverage of the code as only the computation heavy loops are of interest to us. Practically speaking, the more regular loops, which are the ideal candidates for FPGA acceleration, are generally easy to detect and analyze.

Another characteristic of the FPGA platform is its low clock frequency (as compared to a typical CPU), thus to have significant speed up, there needs to be substantial amount of parallelism extracted, which requires large windows of instruction. In addition to instruction level parallelism within basic block or a single iteration of a loop, coarse grained parallelism also needs to be exploited. Blocks of loop iterations are to be executed in parallel, which implies very aggressive instruction reordering when the accelerators are created. Consequently, speculative execution, which some binary-based dynamic parallelization techniques were based on, may become rather expensive. As states generated by the speculatively performed operations need to be buffered, the amount of space required to accommodate the massively parallel execution engine can be large. The subsequent commit of these states may also induces long delays or requires complicated hardware mechanism. In particular, for any speculatively disambiguated memory accesses, the address streams need to be dynamically cross compared to ensure the reordering of loads and stores are indeed valid. Using past execution profile can boost the confidence with which the disambiguation is performed, but the probabilistic nature of the approach does not relieve us the need for costly dynamic checking mechanisms. To generate lean and fast accelerators, in this work, we try to extract a set of conditions for parallelization which does not vary with the amount of work in the loop. In other words, we want to find computation that can be done during run time with a fixed, statically known cost, yet still guarantees the validity of the instruction reordering we have performed for accelerator generation.

Dependencies in Loops

To understand what characteristics a loop nest should manifest for it to be a feasible target, it is useful to start from the theory of loop dependencies. As explained in section 3.2, statements cannot be parallelized or reordered when there are RAW, WAR or WAW dependencies between them. In the context of instructions in loop nests, [116] defined loop-carried and loop-independent dependencies between a pair of statements S_1 and S_2 (at least one of which is writing to memory):

- loop-carried dependency: S_1 accesses a memory location on one iteration of a loop, and S_2 accesses the same location on a subsequent iteration.
- loop-independent dependency: S_1 and S_2 access the same memory location in the same loop iteration, and there is a execution path from S_1 to S_2.

The concepts of *iteration number*, *iteration vector* and *iteration space* were also introduced. The iteration number is the index number for a particular loop iteration, while the iteration vector extends this concept to a multi-level loop nest. Each element in the vector corresponds to one level in the loop nest with the left most element representing the outermost loop index. The set of all possible iteration vectors then constitutes the iteration space. Figure 5.2 visualizes these concepts with a simple loop nest.

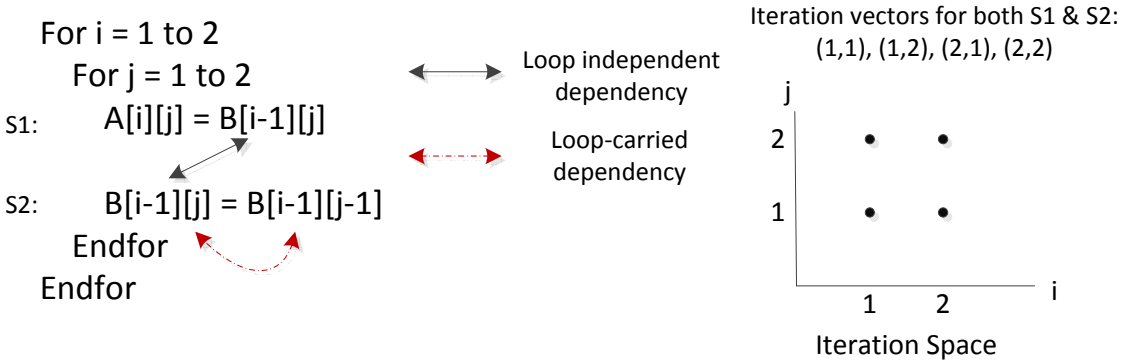


Figure 5.2: Dependencies in a Loop Nest

To exploit coarse grained parallelism in the loop nest, we need to ensure the absence of loop carried dependencies at a particular level in the loop nest. There are also simple transformations (e.g. loop interchange) which can move loop carried dependencies to other loop levels so only loop-independent dependencies are left. To enable the discovery of these opportunities, we need to find, in each statement's iteration space, if they intersect with other statements' accessed memory locations. In general, the memory addresses to be accessed can be arbitrary functions of loop indices and run time data, which makes it impossible to statically determine if dependencies exist. There are, however, a set of problems where the memory referenced can be analyzed during compile time as the addresses are affine functions of the loop index variables. For loop nests of this kind, dependency analysis is essentially finding integer solutions for the problem:

$$f(\vec{x}) = h(\vec{y}) \tag{5.1}$$

$$\begin{aligned} \text{where} \quad & f(\vec{x}) = a_0 + a_1 x_1 + ... + a_n x_n \\ & h(\vec{y}) = b_0 + b_1 y_1 + ... + b_n y_n \\ & \vec{x}_{lb} \leq \vec{x} \leq \vec{x}_{ub} \\ & \vec{y}_{lb} \leq \vec{y} \leq \vec{y}_{ub} \end{aligned}$$

Or equivalently this linear Diophantine equation:

$$a_1 x_1 - b_1 y_1 + ... + a_n x_n - b_n y_n = b_0 - a_0 \tag{5.2}$$

$$\begin{aligned} \text{where} \quad & x_{lb_k} \leq x_k \leq x_{ub_k} \\ & y_{lb_k} \leq y_k \leq y_{ub_k} \end{aligned}$$

Both function f and h takes a iteration vector from within the iteration space of the statement, point-wise bounded by $\vec{x}_{lb}/\vec{y}_{lb}$ and $\vec{x}_{ub}/\vec{y}_{ub}$, and map it to a memory location. When multi-dimensional arrays are used, variables can be separated such that we have multiple simultaneous equations which are simpler to solve. The loop carried dependency in figure 5.2, for instance, correspond to the following equations:

$$\begin{aligned} -1 + x_1 &= -1 + y_1 \\ x_2 &= y_2 - 1 \end{aligned}$$

$$\begin{aligned} \text{where} \quad & 1 \leq x_1 \leq 2 \\ & 1 \leq x_2 \leq 2 \\ & 1 \leq y_1 \leq 2 \\ & 1 \leq y_2 \leq 2 \\ & x_1, x_2, y_1, y_2 \in Z \end{aligned}$$

This problem is an integer linear program, one of the NP-complete problems. There are various techniques [131] [132], proposed over the years, to efficiently solve a relaxed version of this problem. In our binary-based flow, we also try to target these analyzable loop nests, some of which are especially amenable to FPGA acceleration. Identifying these regions from the program binaries, however, poses some challenges.

Challenges for Binary-based Analysis

The regular and analyzable memory access patterns expressed in high level languages become rather mangled when the program binaries are being examined. All memory accesses are pointer based with no high level information to indicate if the data structures they are targeting are disjoint. With dimensionality of the arrays eliminated, separate variables in original address calculation are now coupled to each other. In essence, we have to perform the dependency analysis on a huge linear array with all addresses being the result of some run time computation. Figure 5.3 illustrates how different a set of memory accesses manifest themselves in the actual high level source code v.s. program binaries.

In this example, even though the original array references are all affine functions of the loop index variables, in the program binaries, their calculation is much more complex. Dependency testing involving only a single loop index in the source code now have to deal with multiple induction variables. More importantly, for equation 5.2, the coefficients ($a_0...a_n$ and $b_0...b_n$) are statically known constants while in the binaries, everything is a runtime data item stored in either a register or a memory location. Naively substituting these symbolic variables into the Diophantine's equations yields a non-linear formulation which can not be solved by the common techniques in optimizing compilers. On the other hand, as long as

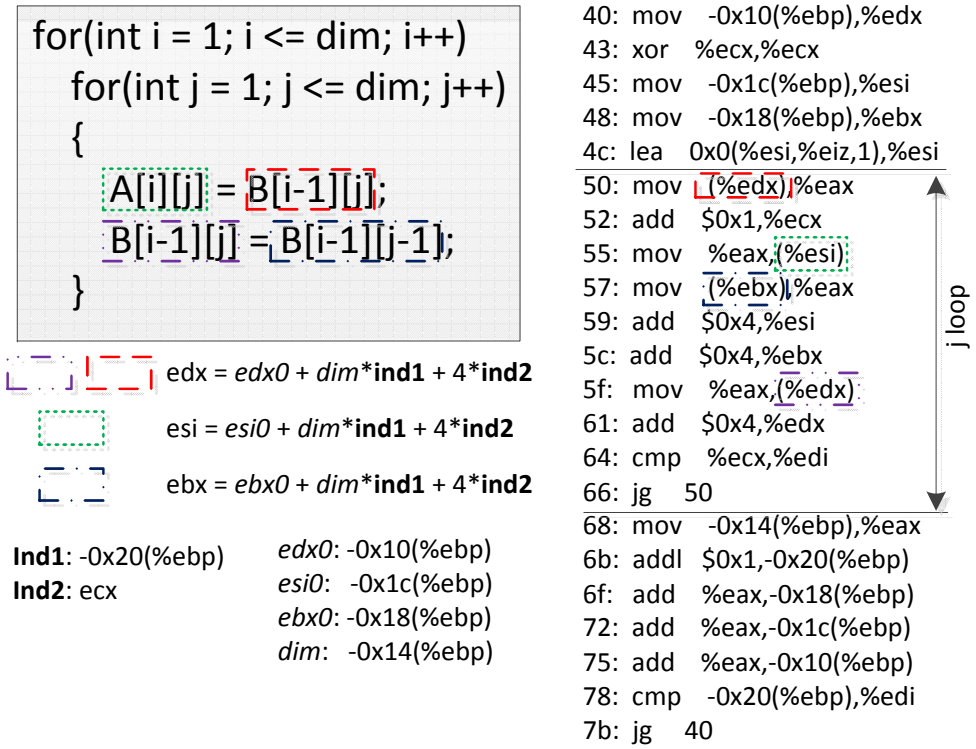


Figure 5.3: Memory Accesses in Program Binaries (x86)

these coefficients are unchanged during the execution of the loop nests, we can use past execution profile to extract these numbers, perform analysis and carry out parallelization. Meanwhile, as the assumed parallelism is based on past behaviors, we need to leverage various classic dependency testing techniques to formulate a set of checks to be performed during run time as well. The cost of these check, however, is fixed. The dependency analysis problem does not scale up with the number of iterations, but rather the number of levels in the loop nest, which is easily recognizable from the static binaries. We can therefore quickly verify our coarse grained parallelization before the invocation of the accelerator, avoiding speculative execution and the associated costs.

In our flow, dataflow analysis is always performed on loop nests to ensure there are no updates to these coefficients during the loops' execution, before more detailed dependency testing and parallelization are attempted. The potential for actual speed up of course, largely depends on the existence of memory level and coarse grained parallelism.

From Dependency Testing to Parallelization

To identify the opportunities in parallelizing a loop nest, the *dependency distance vectors* and *direction vectors* [133] are used.

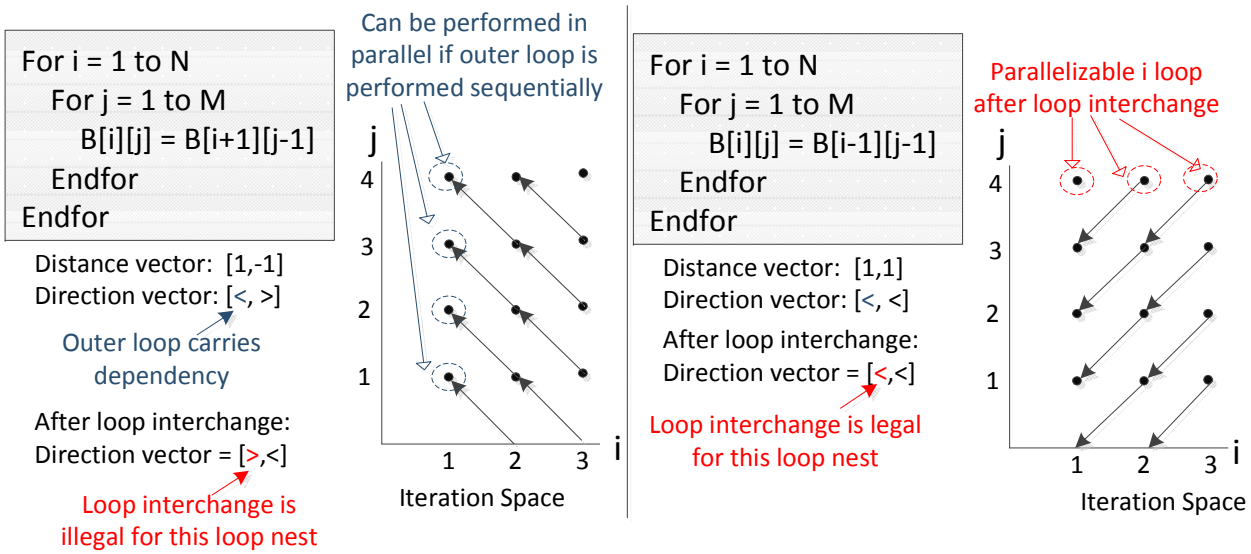


Figure 5.4: Parallelization with Direction Vectors

The *dependency distance vectors* can be computed:

$$\vec{d}(\vec{x},\vec{y}) = \vec{y} - \vec{x}$$
$$\text{where} \quad \vec{x} \text{ and } \vec{y} \text{ are solutions to equation 5.1}$$

These vectors give rise to the *directionvectors* $D(\vec{x},\vec{y})$:

$$\vec{D}(\vec{x},\vec{y})_k = \begin{cases} \text{“}<\text{”} \text{ if } \vec{d}(\vec{x},\vec{y})_k > 0 \\ \text{“}=\text{”} \text{ if } \vec{d}(\vec{x},\vec{y})_k = 0 \\ \text{“}>\text{”} \text{ if } \vec{d}(\vec{x},\vec{y})_k < 0 \end{cases}$$

The convention is to have the lexicographically earlier iteration vector to be $\vec{x}$ and with that the leftmost non-“=” element of a direction vector would always be “<”. The index for this leading “<” is also the level of the loop-carried dependency. Assuming a transformation T does not violate loop-independent dependencies, [116] proved that T is valid as long as it does not cause some of the direction vectors to have “>” as the leftmost non-“=” component. In addition, any iteration reordering at a level of the loop not carrying dependency is also valid. Using these proven theorems, we can decide if loop transformations and iteration parallelization are legal when we know all the direction vectors of a loop nest. Figure 5.4 illustrates a few scenarios where dependency direction vectors are used to determine validity of transformation/parallelization.

5.4 A Two Phased Approach for Accelerating Program Binaries Using FPGA

As the accelerator synthesis, which also includes the traditional FPGA CAD flow, can take hours to finish, it cannot be performed on the fly while the program to be accelerated is waiting. The preceding step, when loop nests are transformed/parallelized, is therefore also performed offline. In this work, Banerjee's test [131] is used to find the direction vectors used for the extraction of parallelism. Since the required coefficients for equation 5.2 are obtained from past execution profiles, the dependency testing and parallelization performed during this *offline phase* are a reflection of the programs' past behavior. When the accelerator is actually running, the input data would have changed. The *online phase* thus includes a mechanism to guarantee the semantics of the program is not violated by the reordering performed during the accelerator synthesis. As we have mentioned in section 5.3, a verifying function is invoked before the activation of the accelerator, ensuring the correctness of the overall execution.

This online phase test is also one of the reasons why we choose Banerjee's method for our dependency testing even though it is not the most accurate dependency testing method available. The Banerjee's inequality tests for existence of any real solutions for the Diophantine's equations. Since it only tackles the non-integer relaxation of the ILP formulation, the results obtained are conservative. It will never report lack of dependencies when one exists, but may report false dependencies, i.e. when all the real solutions are non-integral. More accurate tests like the Omega test [134] find integer solutions but are more costly. Since we are going to perform the test again using run time data when the accelerator is actually being activated, the faster, though more pessimistic, method is preferred.

When applying Banerjee's method, a particular direction vector $\vec{v}$ is subjected to test, and the result reveals if a pair of memory accesses are dependent in $\vec{v}$'s direction. To find all the direction vectors for dependencies, a hierarchy of tests may therefore be involved. For a pair of memory accesses in a two level loop nest, the possible tests are shown in figure 5.5, where "$*$" denotes a union of "$<$", "$=$" and "$>$". Negative test result from any node in the hierarchy eliminates the necessity to continue testing its subnodes. A subset of these dependency direction vectors are true for each loop nest and dictates what kind of transformations are valid for accelerator synthesis.

Offline Phase: Accelerator Synthesis

Dependency Analysis

To perform the offline dependency testing, we built an analysis pass within the LLVM framework. As LLVM takes C/C++ as input, a preprocessing step is performed to convert the binary of the selected loops to C functions. It is also in this step when we extract coefficients for the Diophantine equation and bounds for loop indices from past execution profiles. The numbers obtained are substituted as constants into the C functions, enabling the subsequent

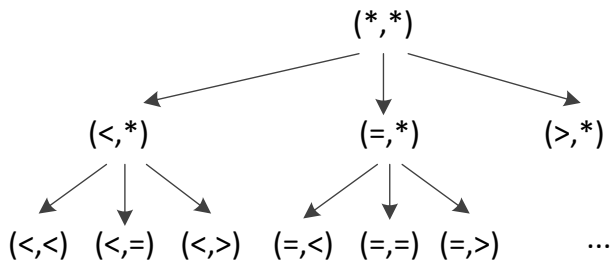


Figure 5.5: Hierarchy of Dependency Testing for A Two-Level Loop Nest

analysis. In the original binaries, these variables are often stored in the memory. Thus in addition to examining their past values, we also perform a check to ensure their memory locations do not alias with any other references performed by the loop nest, again using the addresses observed from the profile. This check can be easily formulated using Banerjee's method—we essentially have a set of affine functions which only contain constant terms. Alternatively, these locations can often be recognized as part of the call stack, and are normally completely disjoint from the "moving" part of the memory footprint in the loop. We can thus easily disambiguate them using a simple range test. Of course, when the accelerator is actually being invoked, these tests would need to be run again, as will be described in later in this section.

Figure 5.6 illustrates the steps involved in converting binary to a synthesizable C function. The original output from the disassembler is used to generate two different versions of C functions. The first, incorporating values collected from past execution profile, can be analyzed by Banerjee's method. The results are the dependency direction vectors between memory operations. In this particular case, the store and load operations are found to be non-aliasing. This information is then used to generate the final C function, where each memory access is offset from a different pointer. Whether we feed this version to conventional HLS or our pipeline generation flow, separate pointer arguments can potentially be mapped to multiple memory ports, through which data requests can be initiated independently.

Memory Level Parallelism

There are actually two levels of parallelization when it comes to scheduling memory accesses during HLS. As shown in figure 5.6, the absence of dependencies between memory operations allows for their association with different pointers. The scheduling engine in the HLS tool can then schedule them without being constrained by the original program order. However, multiple pointers may still be assigned to a single physical memory interface. The structural hazard would then constrain the action of the scheduler, as a single memory port may only accommodate one load and one store per cycle. To loosen this structural constraint, it is possible to create multiple memory ports, each associated with a subset of pointer arguments. More aggressive scheduling, from the datapath's perspective, can then be performed. In

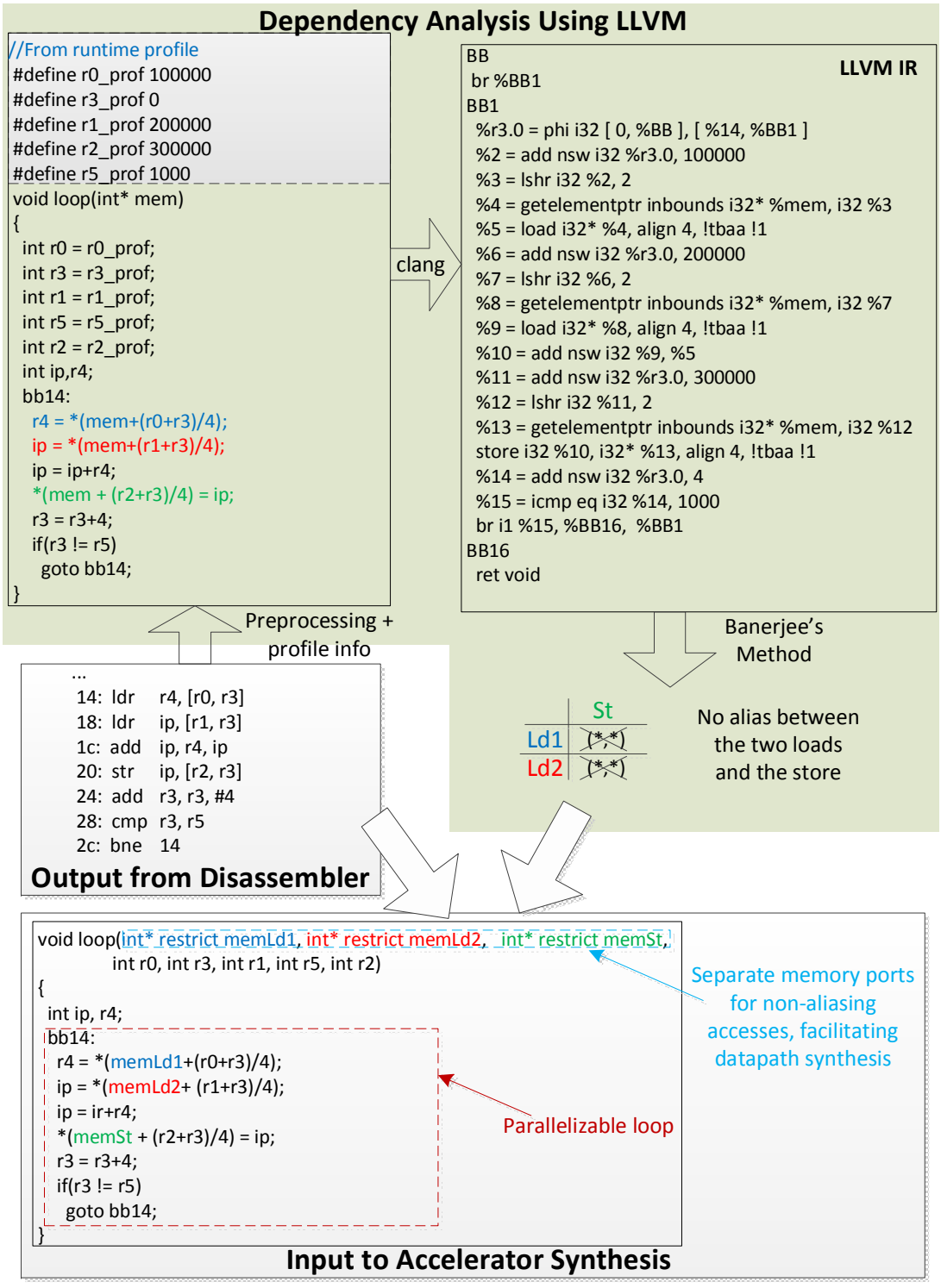


Figure 5.6: From Disassembled ARM Binary to Synthesizable C Code

addition, certain memory operation inside a loop can be converted to burst access when it does not have to share the port with others. This frequently boost the efficiency of the off-chip bandwidth usage. In the decoupled computational pipelines described in chapter 3, as each separable memory access is independently scheduled in a standalone module, which has its own memory port, both levels of memory parallelism are taken advantage of.

The precondition for exploiting memory level parallelism is the non-overlap of addresses accessed by memory operations. This is especially true if they are to be issued through multiple ports, as we assume the RTL generation backend, the interconnect and memory subsystem may all be aggressively reordering these requests—by inferring burst accesses or buffering. Therefore, for every pair of accesses whose ordering matters (i.e. excluding load-load pair), we check the direction vector (*,*,...), and associate each access to a separate pointer if the test result is negative. In section 3.2, we described how memory barriers are inserted to prevent overly conservative dependency annotation. Similarly, in our binary based flow, if memory dependencies are carried by outer loops, memory barriers can also be inserted to facilitate the partitioning of memory access interfaces. The direction vector to test here is (=,=,..,*,*), where "=" occupies all levels including and outside of the level of insertion. If negative results is produced, we can be certain that within the same iteration of the loop where the barrier is inserted, the pair of tested instructions do not reference the same addresses. Therefore any reordering of memory operations will be valid. An example of this is shown in figure 5.7.

Coarse Grained Parallelism

In addition to memory level parallelism, the loop shown in the example also has coarse grained parallelism between iterations of the inner loop. A typical high level synthesis flow can employ several common mechanisms to parallelize loops, as illustrated in figure 5.8. For the inner most loop, it is generally very cost effective to pipeline the loop, starting a new iteration before the previous one finishes. If the iterations are parallelizable, the initiation interval would be 1. Assuming there are M iterations, the total execution time of the loop would roughly be M cycles (not counting stalls introduced by cache misses). To further reduce this number, loop unrolling can be performed. It combines multiple iterations into one, effectively turning inter-iteration parallelism into fine grained parallelism. The iteration count is reduced by the unroll factor (U) while the II does not change. Consequently, the total execution time is reduced to roughly M/U cycles with a approximately U fold increase in area.

To improve the throughput of the accelerator even further, multiple independent datapaths can be instantiated, each performing a subset of the iterations. This technique can be used to parallelize outer loops as well. For loop with a large number of iterations, this technique provides a good way to trade off more on-chip resources for better performance, although it also incurs some extra synchronization overheads. Another advantage of having multiple datapaths is the independence between their controllers. In an earlier chapter (section 3.1), we illustrated the vulnerability of a single monolithic schedule to data access

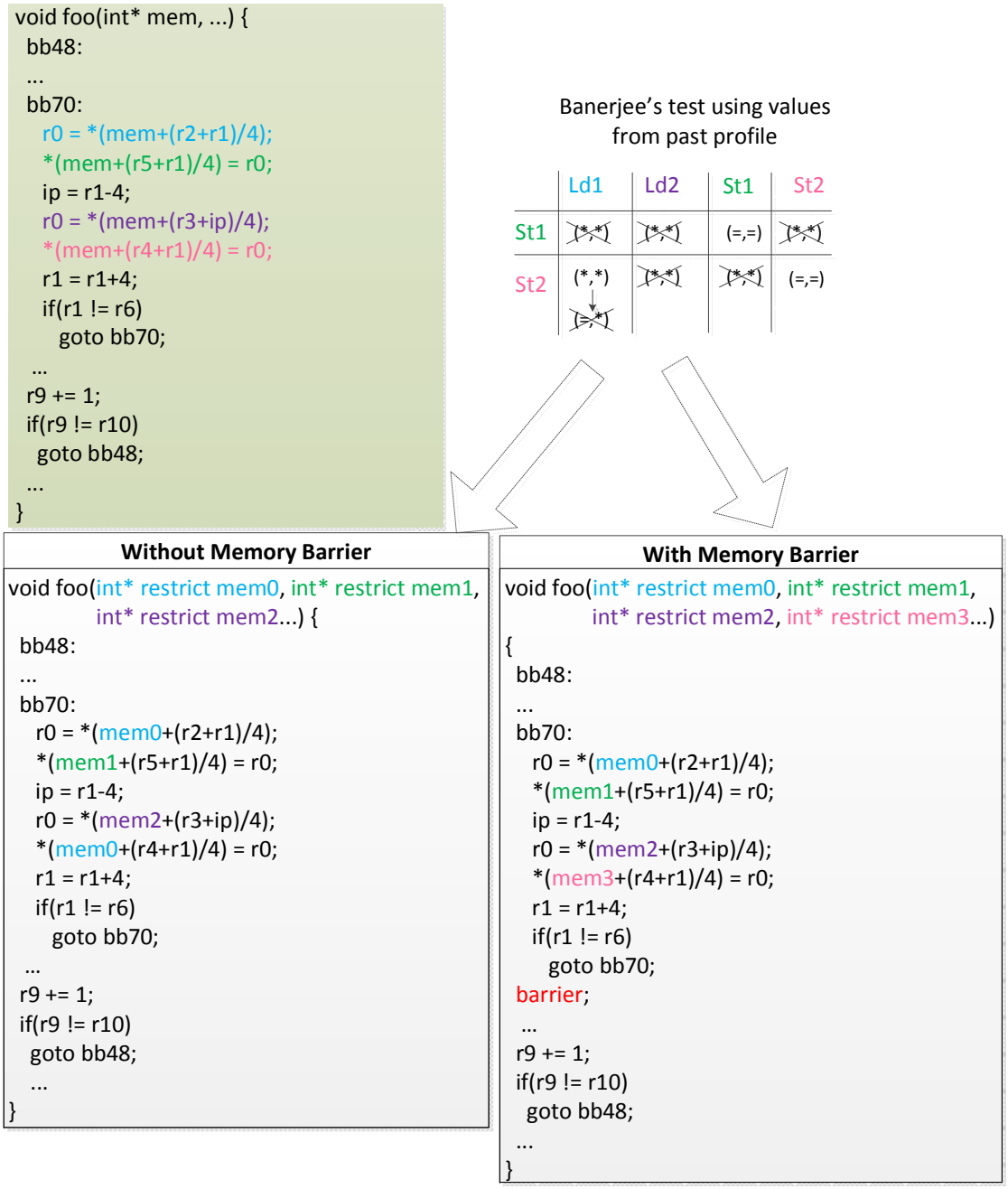


Figure 5.7: Partitioning of Memory Access Interface with Insertion of Memory Barriers

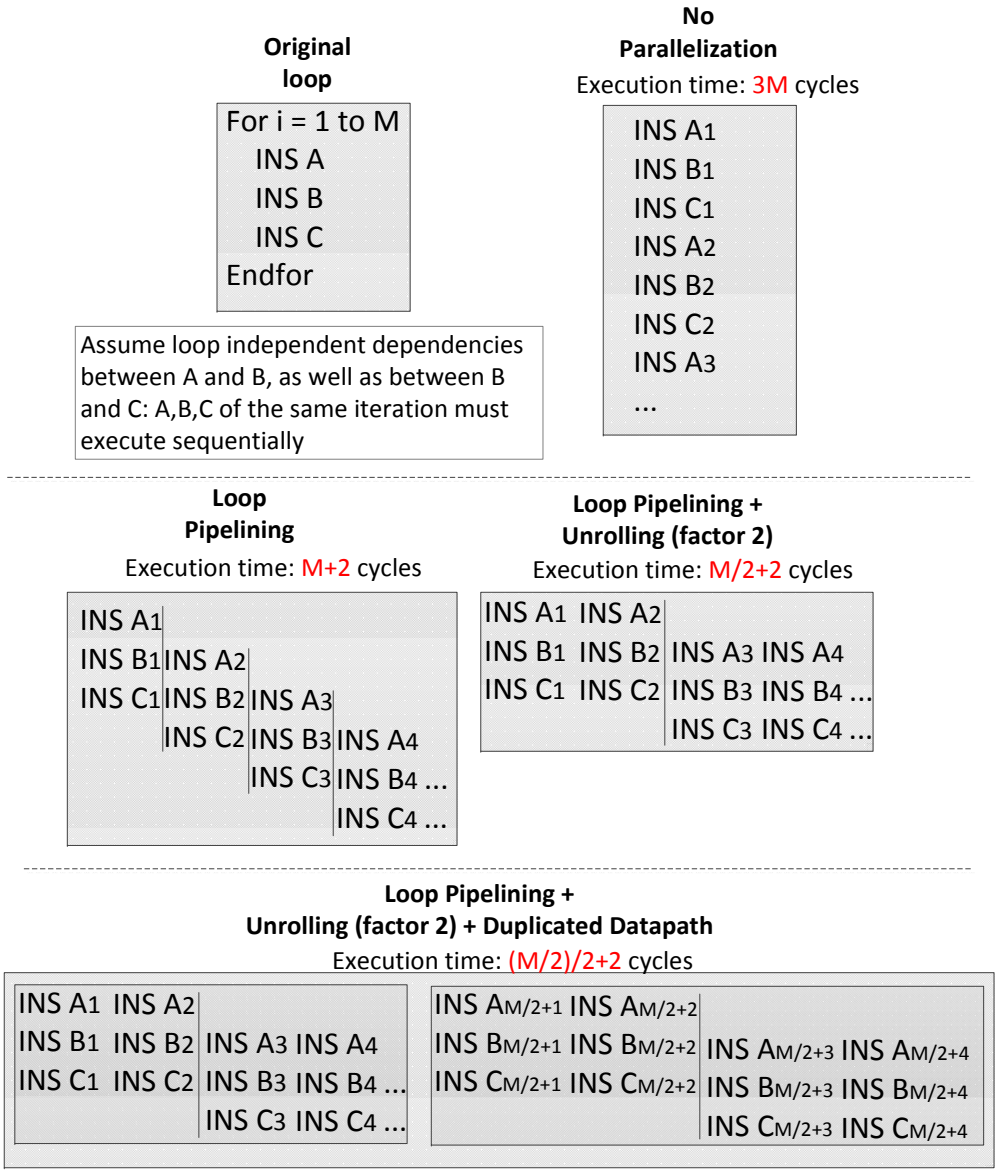


Figure 5.8: Parallelization in FPGA Accelerators

latencies and the resulted underutilization of the memory bandwidth. In the presence of coarse grained parallelism, it is much easier to fully exploit the data bandwidth provided by the platform, as each datapath independently generates requests, whether its peers are stalled or not. Certain platforms also provide multiple channels for off-chip memory access, which can be more easily utilized with duplicated datapaths.

We can draw similarities between these techniques and the compiler optimizations targeting architectural features of various parallel processors. Loop pipelining was a scheduling technique widely used for VLIW machines, where it is often called software pipelining. Loop unrolling, in the synthesis of FPGA accelerators, is similar to vectorization, as a wider processing engine is now used to process multiple iterations simultaneously. The replication of datapaths is essentially generating multithreaded implementations from a serial specification. Previous research in automatic parallelization can thus be leveraged for our flow. In this work, we are not trying to invent new techniques, but to enable past work to be applied to program binaries on FPGA platform. In fact, techniques like loop unrolling and loop pipelining are already built into commercial HLS tools, and can be easily applied using directives.

To generate multiple parallel and independent datapaths, one approach is to convert a single loop nest to multiple functions. Each function can be easily synthesized into an independent accelerator, but synchronization mechanisms might need to be explicitly inserted so that outer level loop and the container function can correctly manage accelerators running in parallel [135]. For our flow however, as we are using the decouple computational pipeline as the architecture template, a simple splitting of the iteration space can naturally lead to the generation of parallel processing pipelines, with proper synchronization achieved by the sending and receiving of tokens. Figure 5.9 illustrate how this process is achieved using a simple example. Even though we construct a control flow where the two parts of the iteration space resulted from the split are executed sequentially, due to the optimizations built into the pipeline generation flow, two parallel pipeline branches are created. Note the second half of the loop nest ($bb20\_dup$) is predicated by the same branch going into the first due to the aggressive predication described in section 3.2. Consequently, instead of being activated after the completion of loop $bb20$, it executes in parallel to it. Of course, if there are actual dependencies between these two halves of the loop, the pipeline branch running $bb20\_dup$ will be stalled waiting for the result from $bb20$. The overall execution would then be practically serialized. Also, the loop counters, which are relatively cheap, get duplicated to multiple stages in the computational pipeline, but only one generates the branch token to be consumed by subsequent pipeline stages. This, along with any other data tokens the successor basic blocks need to read, collectively synchronize the completion of the loop nest with all its dependent operations.

For conventional high level synthesis, it's often desirable to have thread-level auto-parallelization at the outer level of the loops. An architectural template is used in [135] to accommodate and explicitly coordinate multiple independent accelerators each with its own FSM controller. The synthesis of the accelerators and the instantiation of the top level management circuitry are two separate processes and to perform inner loop thread level

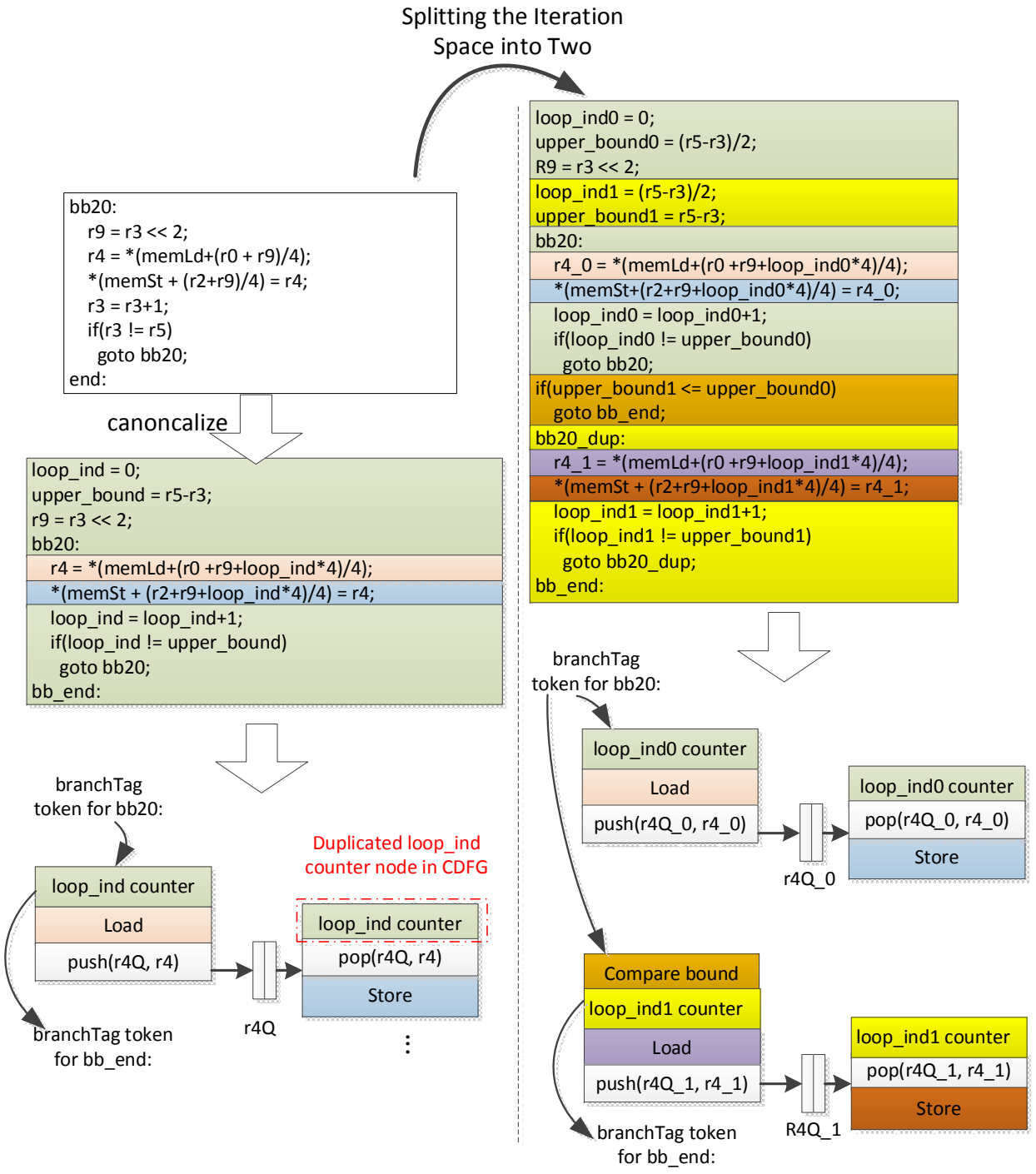


Figure 5.9: Thread-level Parallelization

parallelization would be rather intrusive. There are also cases where threads are manually programmed into the application using APIs such as OpenMP/Pthread. Since the users have explicitly specified the points and mechanisms of synchronization, the tool can leverage a prebuilt library of software/hardware to implement these primitives, managing multiple datapaths running simultaneously. In [136] for instance, each independent thread the user created is mapped to a separate datapath with its own controller, and a software wrapper is synthesized to manage it in the software domain at the top level. Meanwhile, OpenMP pragmas, when attached to the inner loops, direct the HLS tool to create parallel internal accelerators. The FSM managing the container thread would block until all the internal accelerators finish executing. Our flow, on the other hand, does not have any noticeable difference in the implementation of outer and inner loop parallel threads. As the pipeline generated is controlled in a highly distributed way, making independent inner loop (or outer loop) thread only requires a loop splitting which can be easily performed to the intermediate representation. A unified framework involving only source to source transformation can therefore be applied to all levels in the loop nests.

For the target applications of our flow, the memory footprint of the computation is assumed to be much larger than the capacity of on-chip RAM. With the fetching of data from off-chip storage being a major task of the generated accelerators, thread-level parallelization is often preferred to loop unrolling. Unrolled loops usually contain multiple memory accesses which were initially folded as one operation. In cases where the referenced addresses are contiguous, the original memory access can be converted to burst mode load or store, significantly improving the efficiency of memory bandwidth usage. Loop unrolling prevents this burst inference from occurring as the original single stream of addresses is now broken into two interleaved streams. On the other hand, in terms of the effort needed to perform loop unrolling v.s. generate multiple hardware threads, most existing HLS flows can easily perform the former when the user insert the appropriate directives, while thread creation and management, if being supported at all, may require code rewriting/refactoring. As mentioned in the earlier paragraph, the special mechanisms used to accommodate and synchronize multiple independent datapaths can also incur extra area overhead. Fortunately for our flow, the disadvantages in programming effort and resource usage for multithreaded hardware do not manifest themselves, as explained earlier. We can therefore focus on optimizing for the throughput of the final implementation.

So far the analysis and parallelization are all performed independent of the actual RTL generation. In essence we have devised a set of transformations applicable within the LLVM framework. A piece of disassembled binary is converted to C code, which is ready to be synthesized into the decoupled computational pipeline through the RTL generation backend. The last part of the *offline phase* involves pushing the RTL through traditional FPGA CAD flow. This step can take between tens of minutes to a few hours, depending on the size of the design and the utilization of the FPGA chip.

Online Phase: Parallelism Validity Check and Accelerator Execution

During the *offline phase*, when the direction vectors based off past execution profiles are assumed to be true, various transformations are performed. If any of these assumptions is wrong, the generated accelerator will produce wrong results and should not be run. The *online phase* therefore has to verify all the assumptions. More specifically, it needs to check for the presence of any direction vectors which would invalidate the instruction reordering the *offline phase* has done. There are a few main components involved in devising this online checking mechanism:

1. Constructing the list of vectors to test.
2. Generating C code to test each vector.
3. Compiling the vector testing C code into a software routine (SW_{check})
4. Running the SW_{check} created in the previous step

The first three preparatory steps are all done offline, along with the synthesis of the accelerators. The last part though, is invoked right before the execution of the accelerator. Its result determines if the original software binary or the generated hardware accelerator should perform the computation.

To enumerate the direction vectors to be checked, our flow examines all the transformations performed during the *offline phase*. More specifically, the memory level parallelism and thread level parallelism we have exploited are both going to be explicitly tested. Loop pipelining, on the other hand, is applied conservatively by Vivado HLS and in most cases, aggressive reordering only happens after separation of memory ports. Its validity check is therefore not invoked independently.

To ensure memory operations can be reordered or even issued through multiple ports, they need to be completely independent from each other as we have explained previously. The vectors tested during the *offline phase*, (*,*...) or (=...,*...) with barrier insertion, are again included for online testing. For thread-level parallelization, we have to ensure the level at which the iteration space is split does not carry dependency. That is, for any load-store or store-store pair, the direction vector cannot have leading " < " at this level—the vector (=...,<,*...) must be tested negative. Figure 5.10 illustrates these scenarios with a few simple examples.

The actual implementation of these tests was outlined in [116], which is transcribed into C functions in our flow. One optimization, which can potentially be applied in both *online* and $offline$ phases, is to replace Banerjee's test with a simple address range test. When two data structures occupied different ranges in the address space, it is computationally very cheap to just compare the upper and lower bounds of the accessed addresses. However, since Banerjee's test is relatively cheap to evaluate for large sized problems, as will be shown in section 5.6, this optimization may not be necessary.

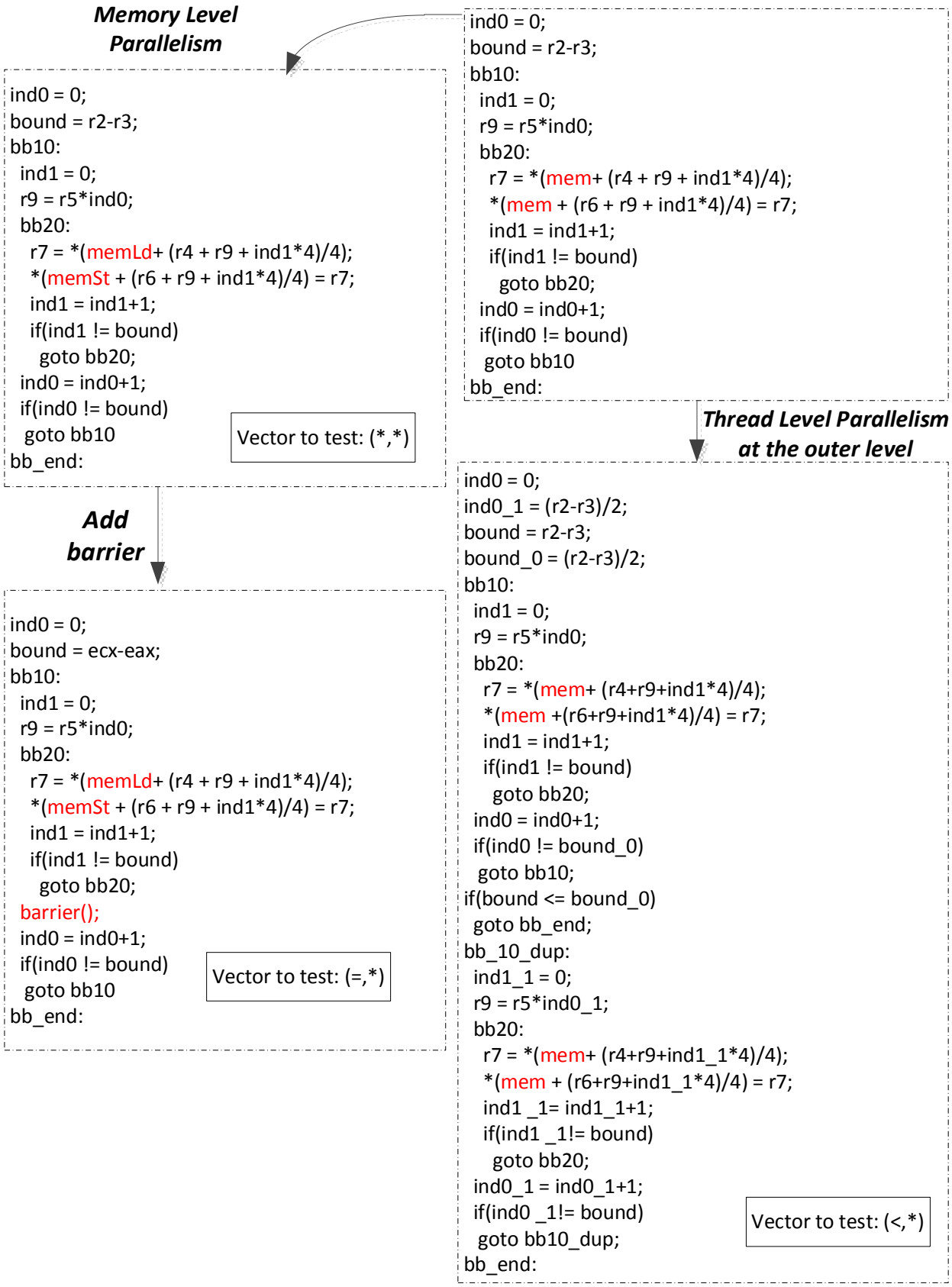


Figure 5.10: Vectors Used for Online Verification of Parallelism

In general, how significant a run time overhead the Banerjee's tests incur depends on how much computation is going on in the targeted loop nest. They certainly reduce the overall speed up attainable with the accelerator and for smaller loop nests, they may completely nullify the benefit of computation offloading. It is thus useful to estimate the performance of the hardware accelerator and the software binary when executing loop nests of different sizes. More specifically, to decide if the amount of computation justifies the invocation of the SW_{check} and the accelerator, we need to find the size of iteration space I such that

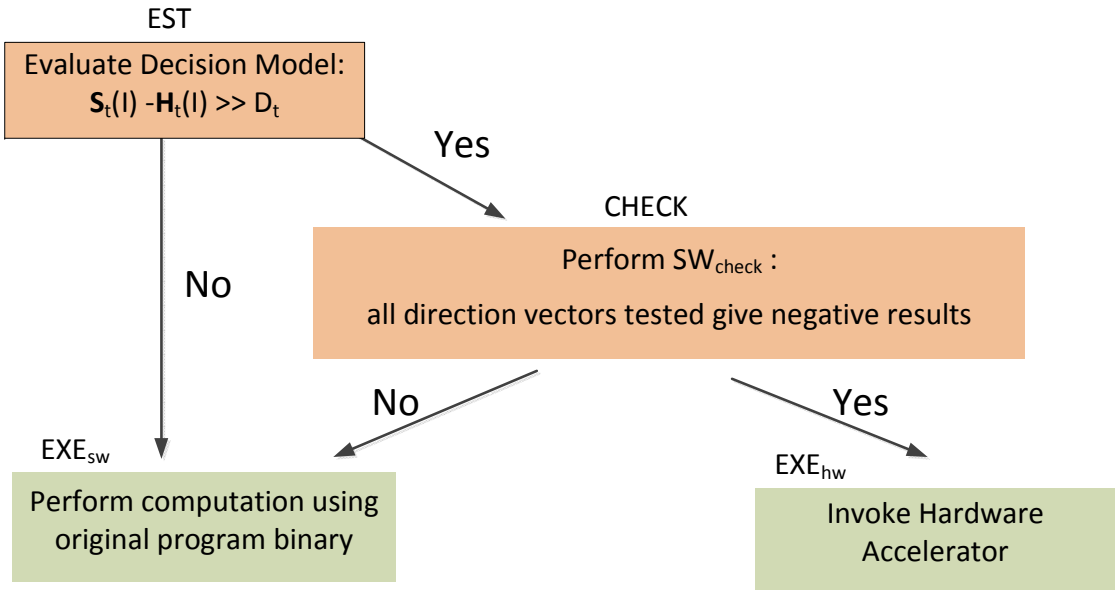
$$S_t(I) - H_t(I) >> D_t \tag{5.3}$$

$S_t(I)$ and $H_t(I)$ are functions predicting the execution time of the loop nest in software binary and hardware accelerator respectively, while D_t denotes the runtime overhead of performing parallelization validity check (SW_{check}). Note that D_t does not vary with the size of I, but the dimensionality of it, which is known during compile time. On the other hand, to accurately predict how the binaries or accelerators' performance vary with I can potentially involve complex mechanisms. Various processor architecture features, interaction with the memory subsystem, or the the paradigm the accelerator synthesis flow follows can all make a big difference for a particular workload. Simulation based approaches [137][138], which may be suitable for design time exploration, are certainly too costly for automatic run time decision making. One alternative is to measure the actual execution for the software implementation and hardware accelerators (with the online checks) over different sized input, and record the smallest I where computation offloading actually provides benefit. As larger iteration space further amortizes the performance degradation caused by the online checks, invocation of accelerators should lead to speed up as well. We can then reduce the decision model to a series of simple arithmetic and comparison operations involving the loop bounds and the recorded I, which can be evaluated during the *online* phase with negligible impact on the overall execution time.

To summarize, the main steps for the execution of the accelerator-augmented program binaries is illustrated in figure 5.11. The worst case scenario is when all the parallelism checks are performed and dependency vectors invalidating the parallelization are found. This should rarely occur, as previous studies found dependence behavior almost never changes with the data input [139][140].

5.5 Accelerator Integration with the Application Binary

An important part of integrating the accelerator with the application binary is the transfer of machine states between the FPGA and the CPU. As the memory are shared between them, updates to non-stack variables do not need to be explicitly handled. On the other hand, as the registers and stack variables are converted into local variables in the C code used for accelerator generation, their changes are not automatically visible to the processor



	Steps Taken	Performance Benefits
Best case	EST→CHECK→EXE_{hw}	$S_t(I)$-$H_t(I)$-D_t-E_t
Early abort	EST→EXE_{sw}	-E_t
Worst case	EST→CHECK→EXE_{sw}	-D_t-E_t

E_t: Time for evaluating the decision model

Figure 5.11: Main Steps in Running Accelerator-augmented Program Binaries

side. To resolve this issue, a special data structure is used to store the final values of these variables. A small segment of code, consisted of a series of memory writes, is also inserted to the end of the C function before it gets processed for accelerator synthesis. Consequently, when the generated accelerator pipeline finishes running, these up to date machine states are deposited, ready to be restored by the CPU.

On the processor side, to replace the loop nest in the original program binary with the generated decision model evaluation, validity check and accelerator invocation subroutine, Dyninst is again used. The abstract representation of the application binary is rather similar whether the program is running as a process or statically stored in the disk. Consequently, the main parts of the "mutator" can work both dynamically, with a already-running process getting patched to have improved speed, or statically, with application binary being rewritten to run faster next time it gets executed.

As mentioned earlier, instrumentation code in Dyninst is organized into "snippet". While it is possible to create a complete function responsible for all the previously described tasks by chaining together a large number of snippets, a more convenient method is to generate a

standalone function encapsulated in a shared library. Dyninst allows for addition of library dependencies to existing binaries, which can then use the newly created function through the function replacement mechanism. Dyninst can redirect all invocation of a function in the mutatee to a new implementation by inserting a non-return jump in the beginning of the old function body. Alternatively, it can rewrite the destination of specific call sites so the execution can be steered according to the context. Our flow currently only uses the former method for simplicity. However, the profile collection instrumentation can certainly be modified to build maps associating workload of loop nests with the invocation site of its container function. The selective function replacement can then be performed accordingly.

Other than the steps shown in figure 5.11, the new function is also responsible to extracting the run time constant for the accelerated loop nest. The register values are read out using inline assembly code, while the stack variables are dereferenced through offsetted stack pointer . The values corresponding to loop bounds and the coefficients are supplied as parameters for the SW_{check} routine. Then, the accelerator initialization and invocation are performed using these values as arguments. Finally, after the accelerator execution, the registers and stack variables updated by the original binary would contain stale values. The replacement function is also responsible for repopulating these registers/memory locations with new values extracted from the special data structure written to by the accelerator.

It should be noted that our flow is a best effort attempt to ensure the machine states in the CPU remain the same whether the original software binary or the FPGA accelerator gets activated. There are CPU specific mechanisms like precise exception for which our flow has no support for.

5.6 Experimental Evaluation

To validate the feasibility of our approach, binaries of a few compute intensive programs were pushed through our flow. We perform our experiments on the same hardware setup (Zynq-7000 XC7Z020 FPGA SoC on ZedBoard) as the one used in chapter 3. Using the Xilinx tool chain for RTL generation and FPGA mapping, we configure the timing constraints and the final clock frequency in the same way as the previous experiments. Vivado HLS is set to target 8ns clock period while the highest clock frequency achieved after place and route is used for performance benchmarking. The baseline of our performance comparison is the software binary running on the ARM core in Zynq. A decoupled computational pipeline is first generated, directly from the C function produced by our preprocessing step, and connected to the HP port before further design space exploration is attempted.

As the targeted regions of the application binaries are regular loop nests, our flow can be configured to take advantage of coarse grained parallelism. In our experiments, we gradually increase the amount of thread level parallelization by more aggressively splitting the iteration space, as described in section 5.4. Meanwhile, as the Zynq SoC contains multiple HP ports connecting the programmable logic to the memory, the amount of bandwidth available to the

hardware threads can be varied. Thus for every benchmark, we also increase the number of HP ports enabled to quantify the effects it has on the overall performance of the accelerators.

Another important component of executing accelerators generated from program binaries, as discussed in section 5.4, is the set of online checks which must be run before the accelerators are invoked. The size of the iteration space, or more specifically, the upper and lower bounds for each level of the loop nest, are first examined to estimate if the overhead of parallelization validity check would have out-weighted the benefit of running the FPGA accelerator. The actual Banerjee's test is then performed. The cost of completing these computation during run time is also evaluated for every benchmark in this section.

Note the purpose here is not to highlight, in absolute terms, how fast we can perform the computation. There are many techniques which can be applied, either manually or through compiler optimizations, to each individual benchmark. We just want to demonstrate how some of those can also be used, without source code modification and recompilation, to take advantage of the FPGA fabric to achieve good speed up.

Performance Results

Benchmark 1: GemsFDTD

The first application, GemsFDTD, is a general electromagnetic solver which solves the Maxwell equations in 3D in the time domain using the finite-difference time-domain (FDTD) method. Every timestep, an update of the E-field and H-field, stored in three dimensional arrays, are performed. Originally coded in Fortran 90, this benchmark would not have been directly synthesizable using the Xilinx tool chain without the binary based flow, even though the workload is particularly suitable for FPGA acceleration.

Two triple nested loops are used to compute the E-field from H-field/H-field from E-field. Their memory footprint is determined by the size of the grid used to represent the space. The data structures representing the fields in 3D scales up cubically with the number of units in each dimension. For our performance comparison, grids of multiple different sizes are used. Figure 5.12 shows the performance of the decoupled computational pipeline for different problem sizes. Coarse grained parallelism at the outermost loop is also exploited to split the iteration space into two halves. All numbers are normalized to the processor performance, reflecting improvement over the software binary.

The pipeline implemented in programmable logic, despite running at a fraction of the processor's clock frequency, is able to outperform the software binary. Its performance advantage is also more pronounced for larger sized grid as the constant cost of initializing the accelerator gets amortized over a greater amount of computation. For a space of 128x128x128 units, the computational pipeline achieves a 3x improvement over the processor, while only 1.5x speedup is observed when targeting an 8x8x8 space. In addition, when we take advantage of the coarse grained parallelism, the performance goes up even further. With the iteration space split two ways, the generated pipeline achieves 4.2x speedup. More aggressive

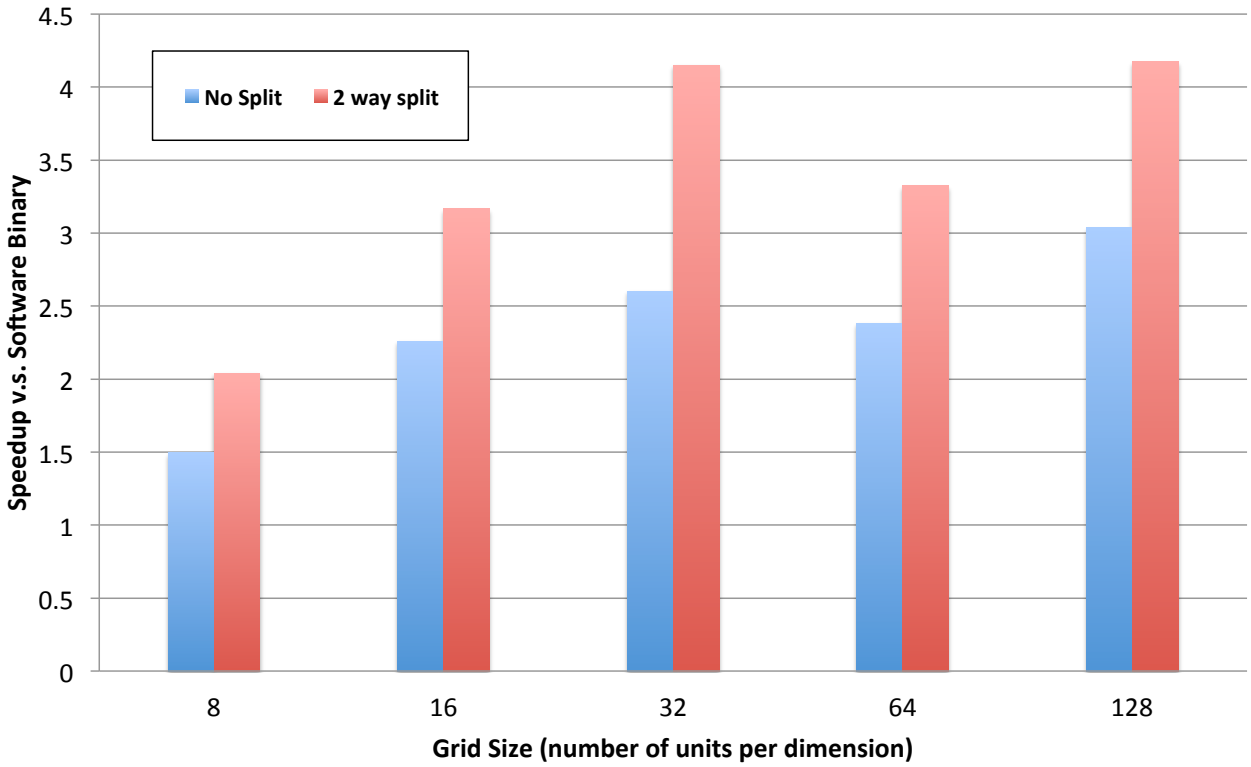


Figure 5.12: Performance Comparison of Decoupled Computational Pipeline and Software Binary for GemsFDTD

splitting of the loop nest results in resource utilization greater than 100% for our platform and is therefore not mapped on actual hardware.

The HP ports and associated interconnect IPs providing data access to the computational pipeline are often the limiting factor for the performance. Figure 5.13 shows how the speedup changes as we vary the number of ports to memory. Evidently, the compute capacity provided by the pipeline with a 2 way split of the iteration space can consume data faster than the rate sustained by a single HP port. As we enable more ports, across which the data access interfaces of the pipeline are evenly distributed, the performance increases further. For the largest problem size, with all HP ports to the programmable logic enabled, close to 9x speedup is achieved.

Shown in figure 5.14 is the overall execution time of the fastest and slowest accelerator configurations, including the overhead of all online checks. The execution time is normalized to that of the software binary and the lower the bar, the faster it is relative to the unaccelerated version. As expected, for small sized problems, the online checks can be expensive, relative to the execution time of both the software and the hardware accelerator. However, as the problem size gets larger and the loop nest itself takes up more time, the overhead

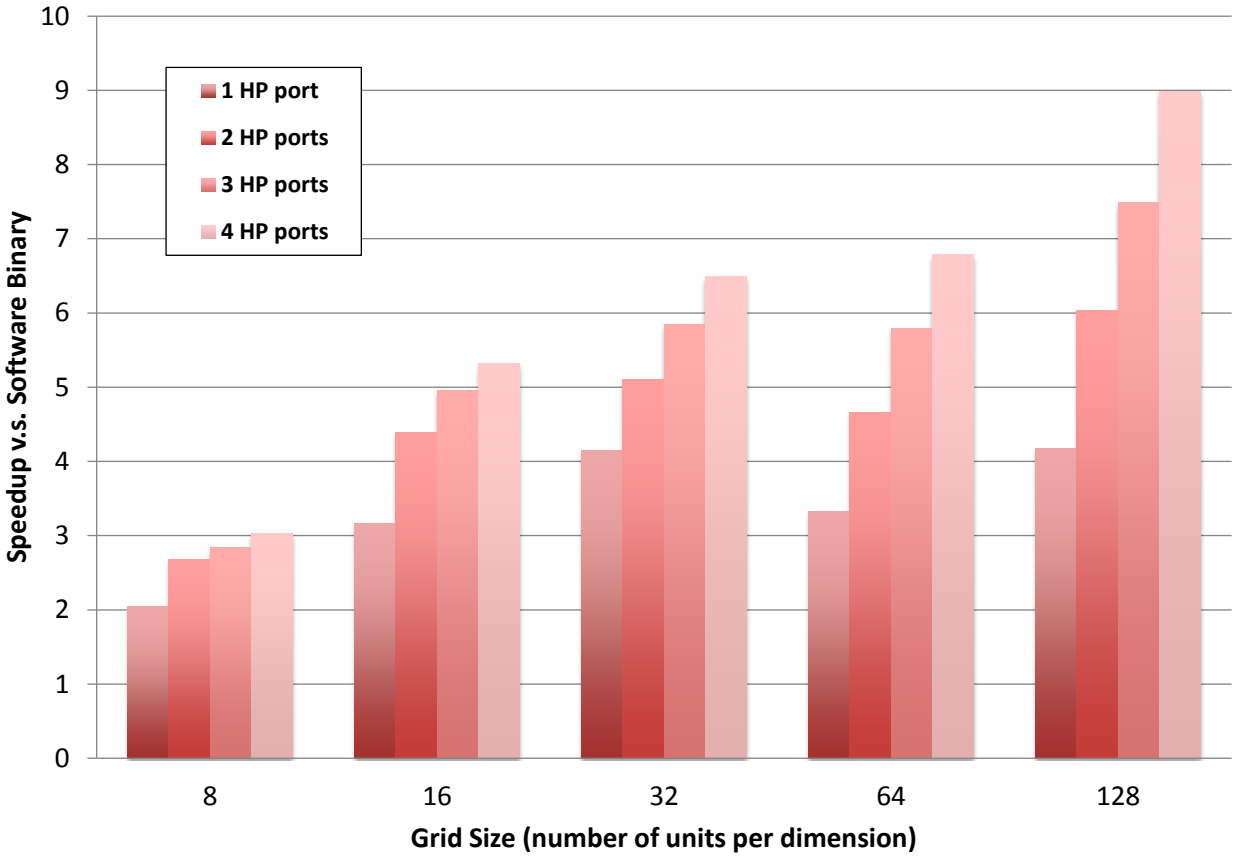


Figure 5.13: GemsFDTD Performance with 2-Way Split of the Iteration Space and Different Number of HP Ports

becomes negligible. For this benchmark in particular, even using the slowest accelerator configuration for the smallest grid we have tried, the performance advantage of the accelerators is big enough to outweigh the overhead introduced.

Benchmark 2: Matrix Multiplication

Matrix multiplication is an computation kernel which has been examined and optimized in many different contexts. In this experiment, we use a vanilla implementation to compare the FPGA accelerator against the processor for different problem sizes. Our data therefore covers various scenarios, from when all operand matrices can fit on the cache and the processor has cache hit most of the time, to the case where no data in a cache line is accessed more than once. Meanwhile, the accelerators being compared do not have any caches instantiated using the programmable logic. As we are only mirroring the memory access patterns of the processor instead of applying additional high level transformations such as blocking, the

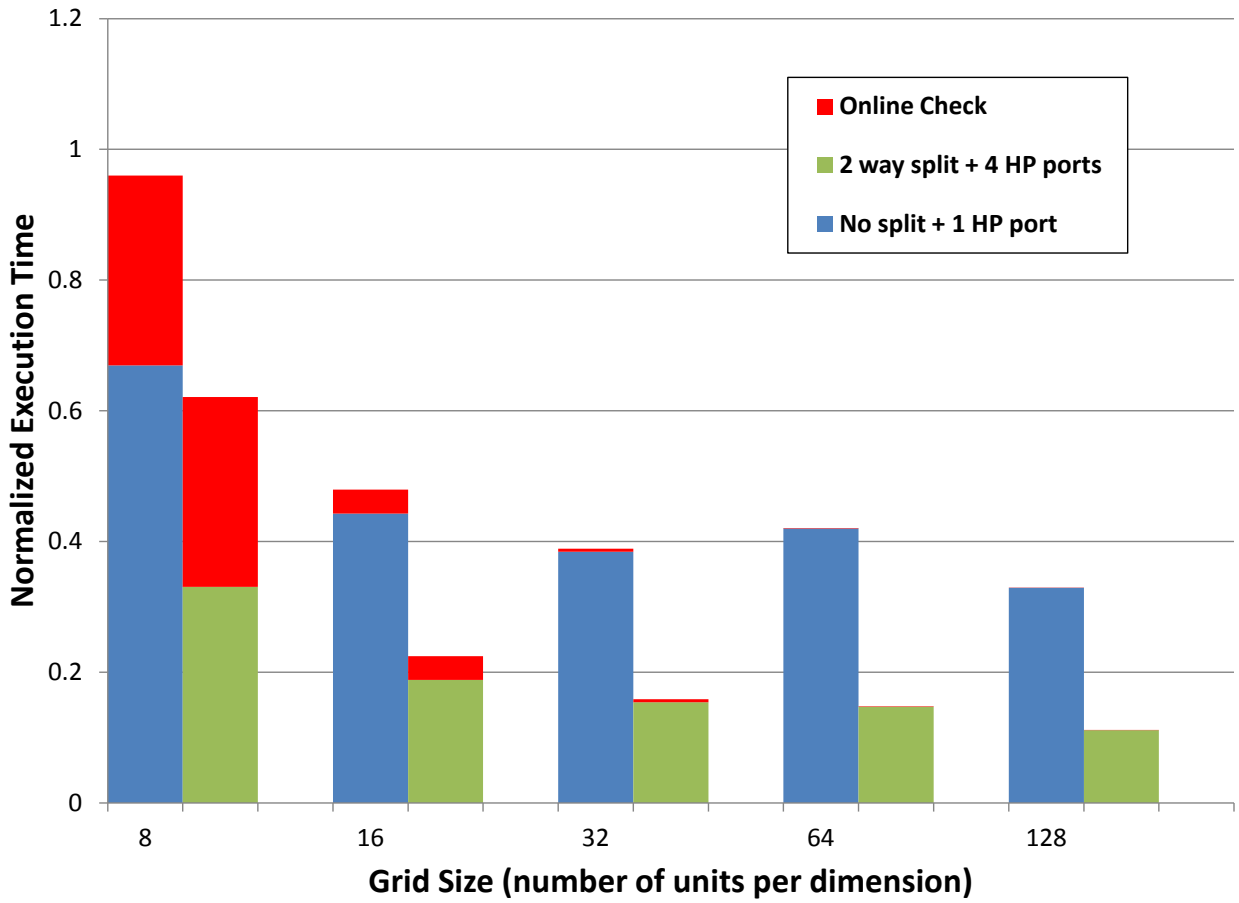


Figure 5.14: Overall Execution Time of Accelerators with Online Checks for GemsFDTD

instantiation of expensive caches on the FPGA provides little overall performance gain for large sized problems.

Shown in figure 5.15 is the performance as we vary the amount of iteration space splitting. As more coarse grained parallelism is exploited, the performance improves. In fact, without splitting the iteration space, the accelerator may run slower than the processor, especially when the problem size is small and fits in the CPU's cache completely. When the outer loop is split four ways however, the accelerator is always faster and for matrices of size 512 by 512, the speedup versus software reaches 2.7x. Note the overall performance does not scale linearly when more coarse grained parallelism is exploited. This is due to the decrease of the clock frequency as the resource utilization goes up, and the single HP port's incapability to keep up with the increase in memory requests. The second effect can be seen from figure 5.16, which shows how the performance of the fastest accelerator configuration (4 way split) varies as more HP ports are enabled. In particular, there is a significant increase in performance when the the number of HP ports changes from 1 to 2, though enabling more ports beyond

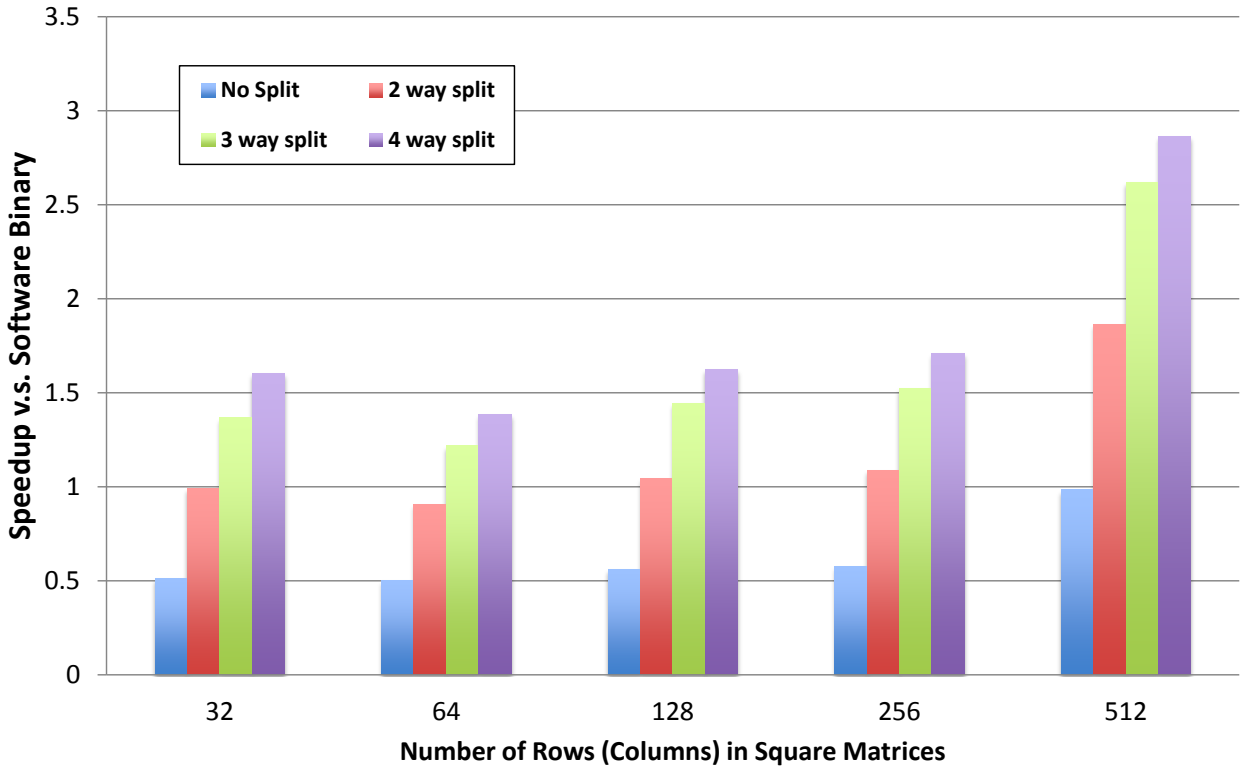


Figure 5.15: Performance Comparison of Decoupled Computational Pipeline and Software Binary for Matrix Multiplication

2 seems to produce less pronounced effects.

In figure 5.17, the effect of online checks is shown. The actual overhead is smaller compared to the previous benchmark as the number of memory access pairs going through the test is much smaller. The performance improvement provided by the two accelerator configurations listed is virtually unaffected as the actual computation takes up two orders of magnitude more time than the dependency tests.

Benchmark 3: Sobel Edge Detection

Another application we have benchmarked is the Sobel edge detection. An input image stored in an 2D array is convolved with small sized kernels, the result of which gets dumped into a separate 2D array. The image can be large and may not fit on chip while a cache instantiated with programmable logic can easily buffer the kernel array. The main loop nest extracted (convolution) in this benchmark is also a core computation pattern being used in digital signal processing, machine learning and many other fields.

As the memory accesses in this application exhibits very high spatial and temporal locality, caches are instantiated on the programmable logic. The iteration space is again split

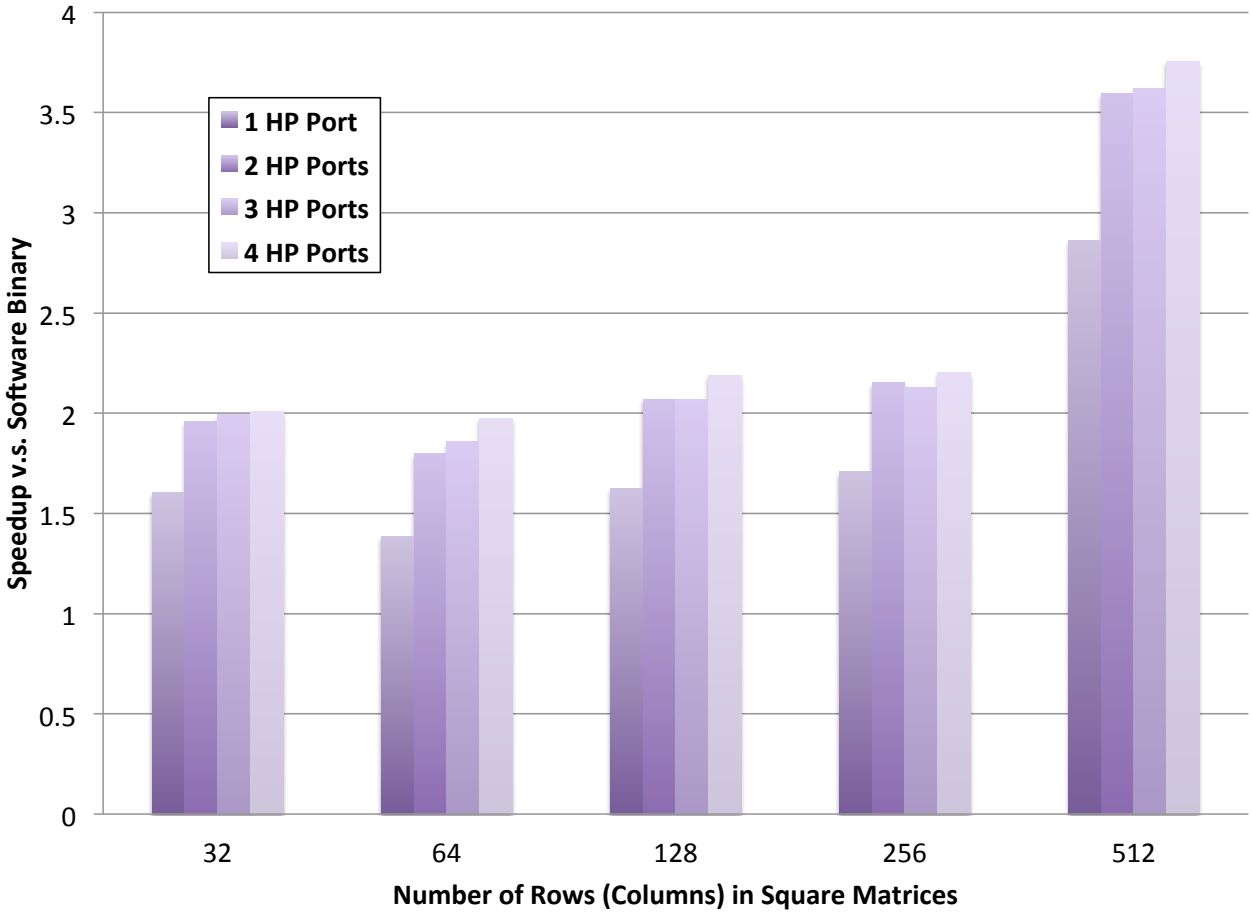


Figure 5.16: Matrix Multiplication Performance with 4-Way Split of the Iteration Space and Different Number of HP Ports

at the outermost loop level, chunks of the input image are thus convolved with the kernel in parallel. The performance improvement over the software implementation is shown in figure 5.18.

The speed improvement again does not scale up linearly when more aggressive thread level parallelization is employed. The achieved clock frequency drops from 111 MHz to 91 MHz when the iteration is split four ways, which seems to account for the plateauing of performance gain completely. Figure 5.19 confirms that by showing how enabling more ports into the memory has little effect on the performance. The caches absorbs most of the requests due to the locality of memory accesses. A single HP port is therefore sufficient for supplying data to the parallel computational pipeline.

Finally, the online checks' overhead for edge detection is shown in figure 5.20. Similar to previous benchmarks, the improvement provided the accelerator is not offset by the extra computation of performed before its invocation.

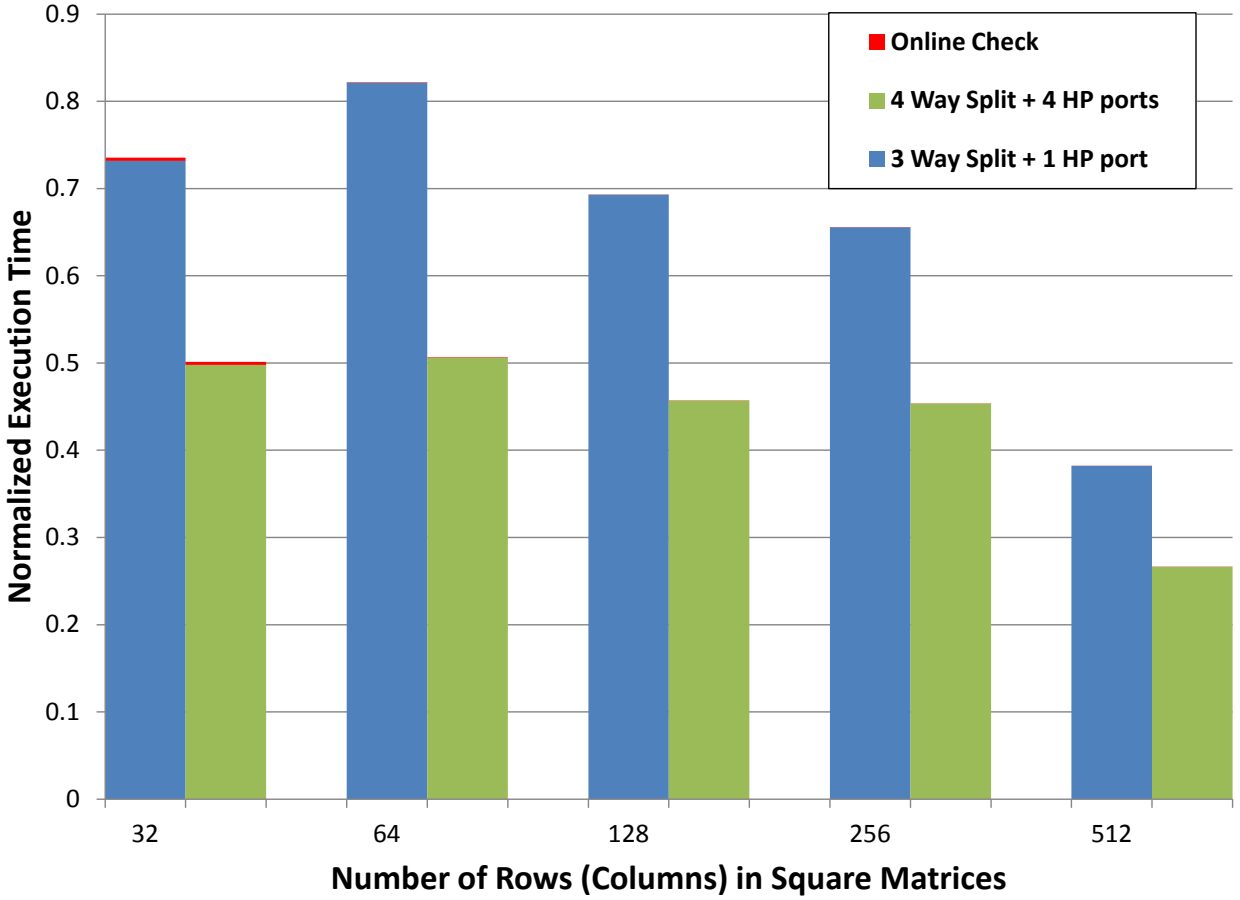


Figure 5.17: Overall Execution Time of Accelerators with Online Checks for Matrix Multiplication

Area Results

Table 5.1 lists how the resource usage varies with the amount of coarse grained parallelism utilized in each benchmarks. For all the benchmarks, the LUT and FF usage roughly scale linearly with the amount of coarse grained parallelism exploited in the implementation. DSP usage in matrix multiplication, on the other hand, increases super linearly as the iteration space gets divided. This is due to the fact that in the non-multithreaded implementation, the zeroes in the lower loop bounds make it possible to generate all memory addresses without using DSP blocks. Subsequent iteration space splitting creates lower loop bounds which are run time variables, leading to the instantiation of multipliers in the address calculation parts of the dataflow, demanding more DSP blocks. Meanwhile, GemsFDTD more than doubled its BRAM usage when two parallel threads are used. This non-linear increase did not come from the pipeline itself, but the AXI crossbars used to connect it with the HP port. As the

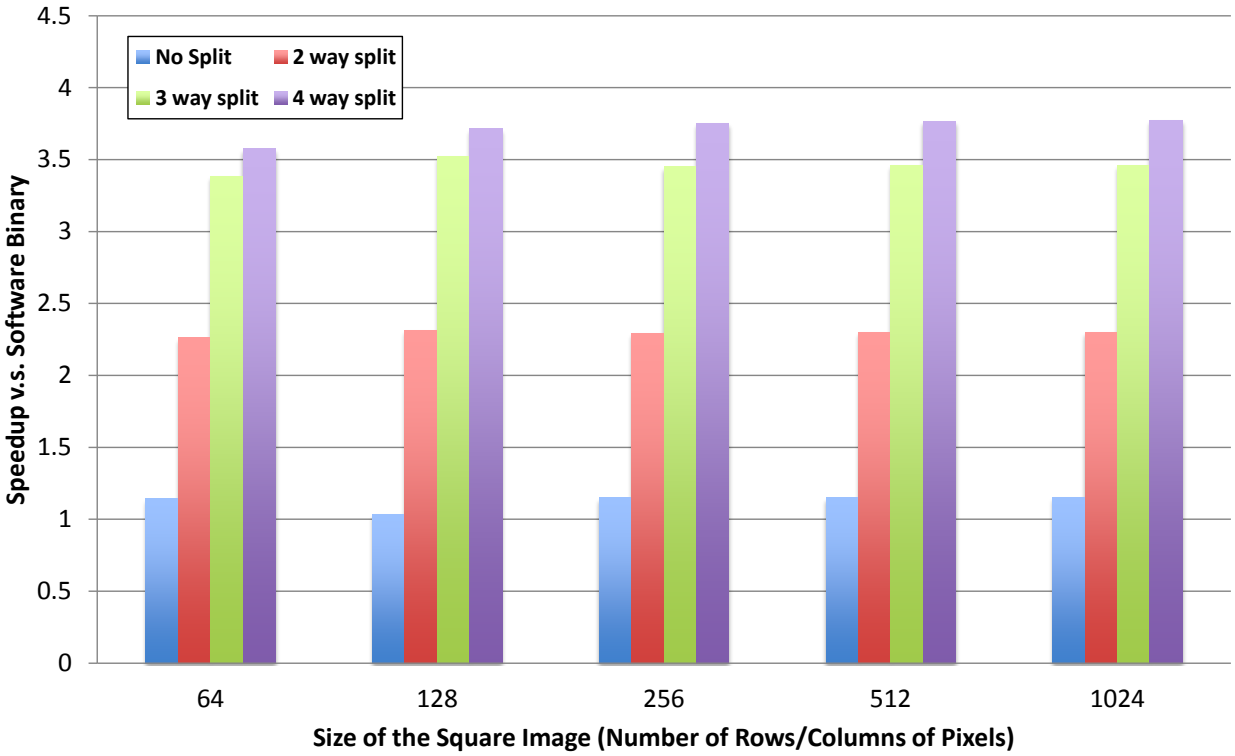


Figure 5.18: Performance Comparison of Decoupled Computational Pipeline and Software Binary for Edge Detection

number of memory ports exceeds 16 when two way split is performed, an additional AXI interconnect IP was instantiated, which uses a few extra BRAM blocks.

Overall, for each loop nest, which accelerator configuration to use depends on if the area used is justifiable in light of the other accelerators' potential to provide speedup. As we are only exploring the possibility of binary based accelerator generation by looking at a single loop nest at a time, the investigation of this trade-off is left to future work.

5.7 Discussion and Future Work

Application of Our Approach in Other Contexts

Our two phased approach leveraging both the run time profile and the static program can be used for parallelization in other contexts as well. There are certain functions whose memory access patterns are not necessarily analyzable using only the source code. An example of this is shown in figure 5.21. As the data structures are passed into the function using pointers, the constant terms (a, b and c) and the coefficient (dim) in the Diophantine equations are all

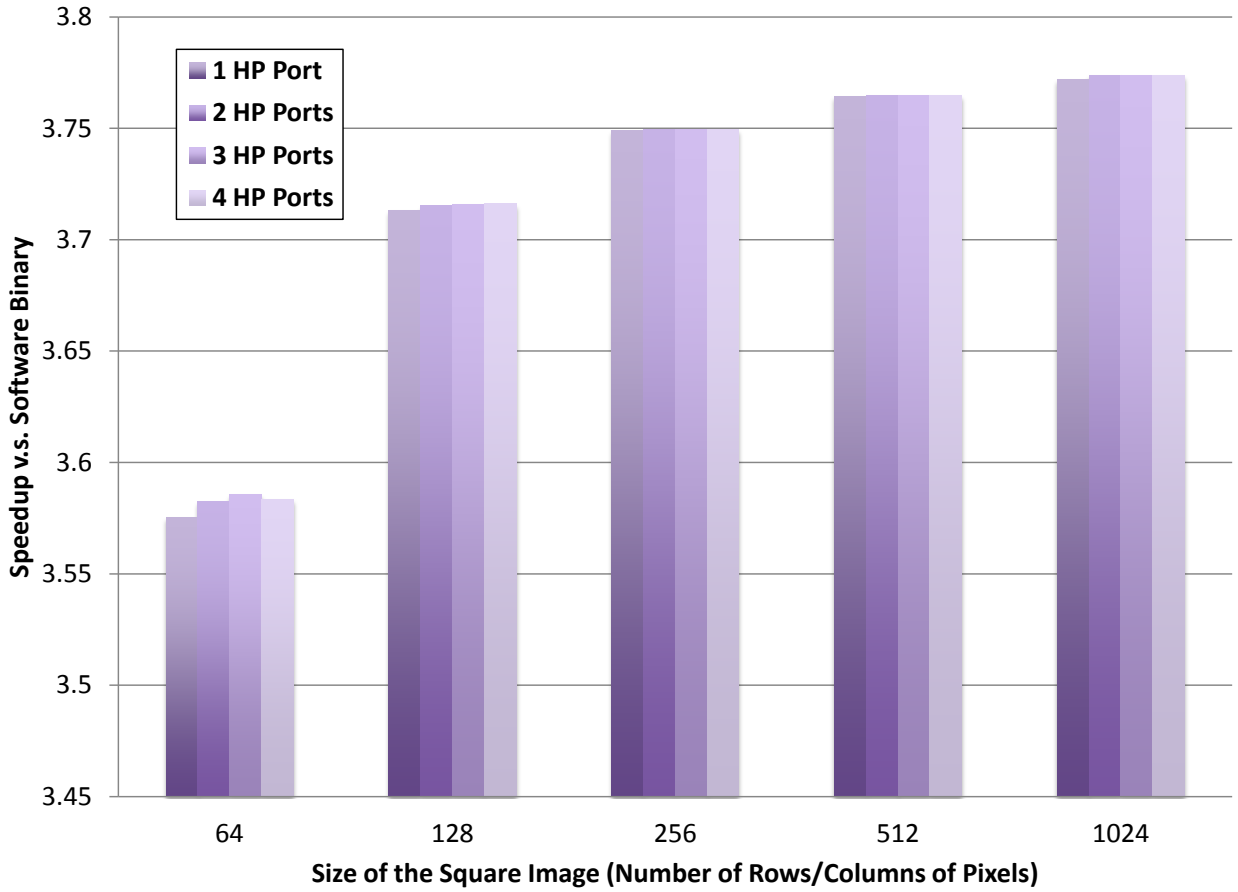


Figure 5.19: Edge Detection Performance with 4-Way Split of the Iteration Space and Different Number of HP Ports

Table 5.1: Resource Usage for Different Configurations of Decoupled Computational Pipeline

Benchmark	Iter. Space Split	LUT	FF	DSP	BRAM
GemsFDTD	No Split	19044	20209	48	20
	2 Way	38730	41749	96	45
Matrix Multiplication	No Split	9441	7945	5	6
	2 Way	18505	16772	14	12
	4 Way	36488	30254	32	24
Sobel Edge Detection	No Split	12184	13413	27	30
	2 Way	22879	25433	54	61
	4 Way	43500	49001	108	122

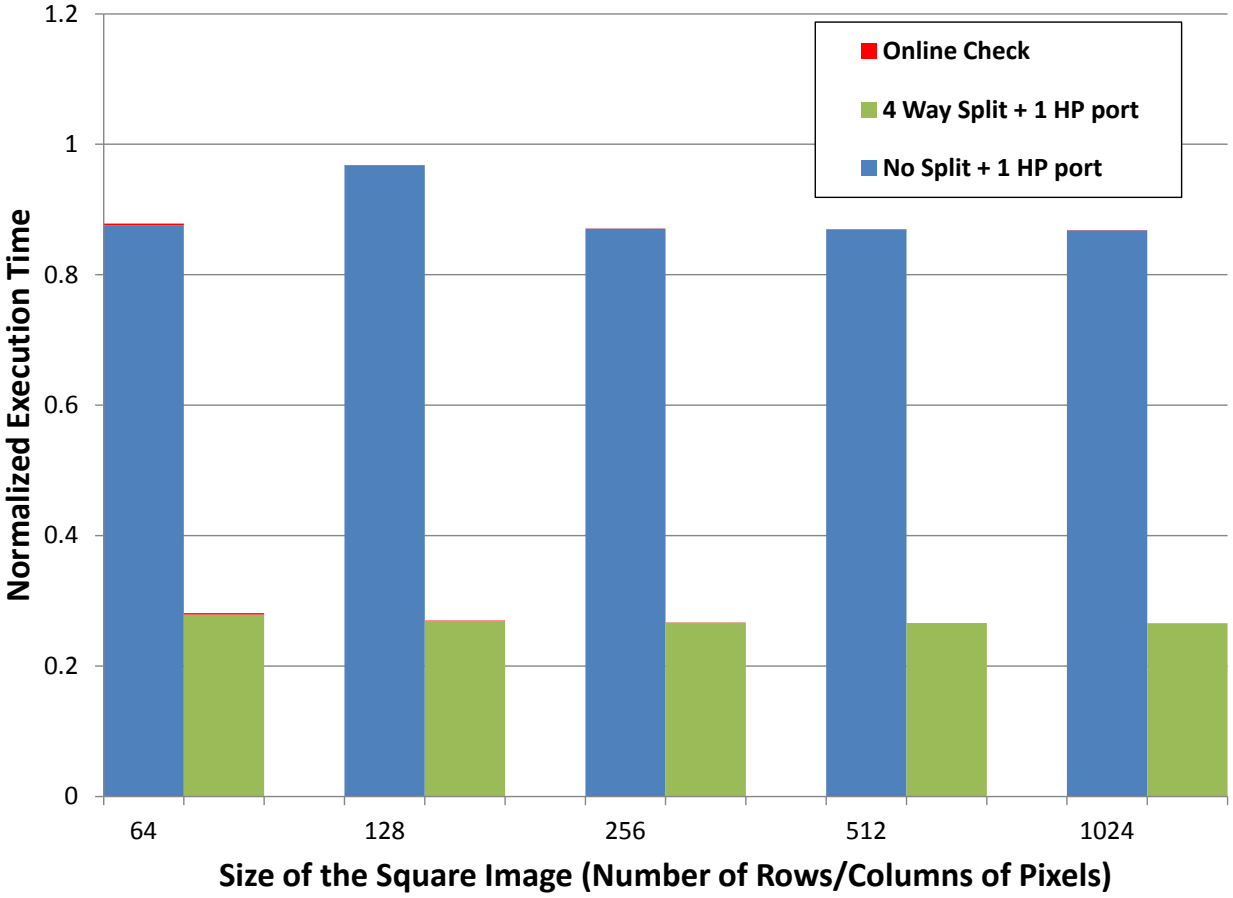


Figure 5.20: Overall Execution Time of Accelerators with Online Checks for Edge Detection

unknown. Banerjee's method, in this case, does not produce affirmative result. On the other hand, with the help of runtime profile, this function can also be analyzed. In this example, as negative results are produced when tested for the dependency vector (*,*), every level of the loop is parallelizable.

Other parallel compute platforms can also be the targets of our methodology. Depending on the speed of the compilation toolchain, the $offline$ phase may potentially be performed during run time. If this online compilation uses the current run time profile, we can forgo the additional online checks and essentially have just-in-time parallelization. In general the choice between having a cached parallel implementation with online checks and performing just-in-time parallelization largely hinges on the speed with which parallel implementation can be created, relative to the execution time of the original code. FPGA accelerators, with potentially hours of compilation time, falls on one end of this spectrum and the two phased approach ends up being the most appropriate one.

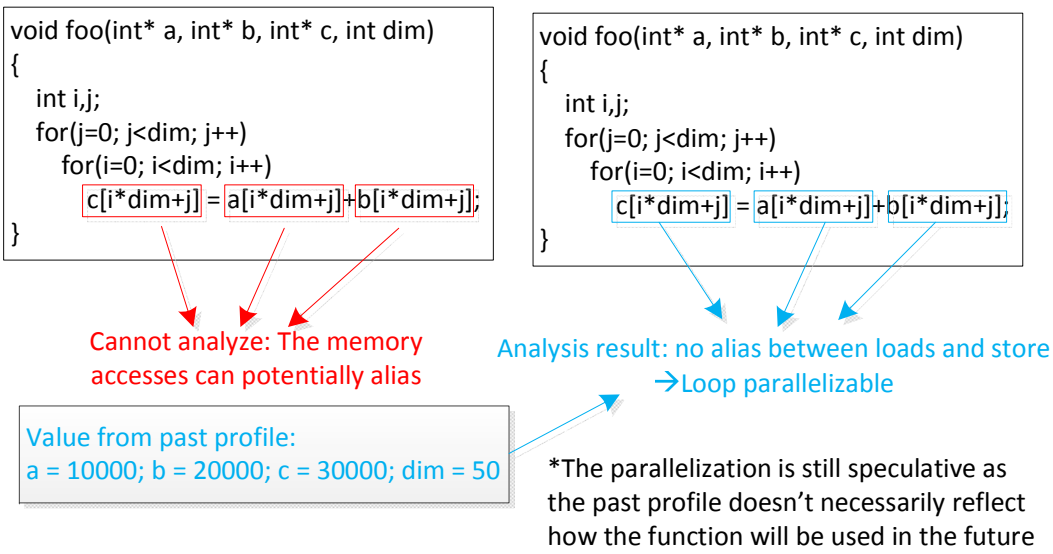


Figure 5.21: Analyze Loop in C with Value from Past Profile

Data Transfer for CPU-FPGA system with Separate Address Space

As the FPGA is used to implement accelerators for programs executed on CPU, both the source and final destination of its working datasets are in the address space of the processor. Our framework assumes the FPGA can directly access the CPU's memory, and leverages the decoupled computational pipeline synthesis flow to naturally extract sets of address generators who stream requests into the memory subsystem. On the other hand, in systems where the FPGA has its own address space, the program data would need to be explicitly transferred into and out of it before and after the accelerator execution. Many of the state-of-the-art platforms with this model are connected to the host through PCI-e connections, which is rather similar to discrete GPU platforms. As the GPGPUs were getting more widely adopted, the automatic management of their communication with the CPU hosts was also studied in multiple projects [141][142][143]. Most of these frameworks require compiler assistance or user guidance to facilitate the management of data transfers and is therefore not directly applicable to a flow targeting existing program binaries.

For the case where the offloaded computation is assumed to have analyzable memory access patterns, like in our flow, it is possible to calculate *a priori* the memory footprint for the accelerated loop nests. As the referenced addresses are always affine functions of the loop indices, for a given part of the iteration space, the boundaries of the accessed range of memory by an instruction can always be computed. This computation would become another part of the *online phase*, supplying parameters for the actual data movement mechanisms.

There is certainly a large space for exploration when it comes to performing the data

transfer, especially when the addresses referenced by the loop nest are non-contiguous. Many fine grained data transfers may be less efficient compared to a few coarser grained memcpy invocation, even though the later may incur wastage as some unused data is also moved. Co-optimizing the data transfer and the computation parallelization based on application specific memory access patterns requires significant effort and can be an interesting direction for future research.

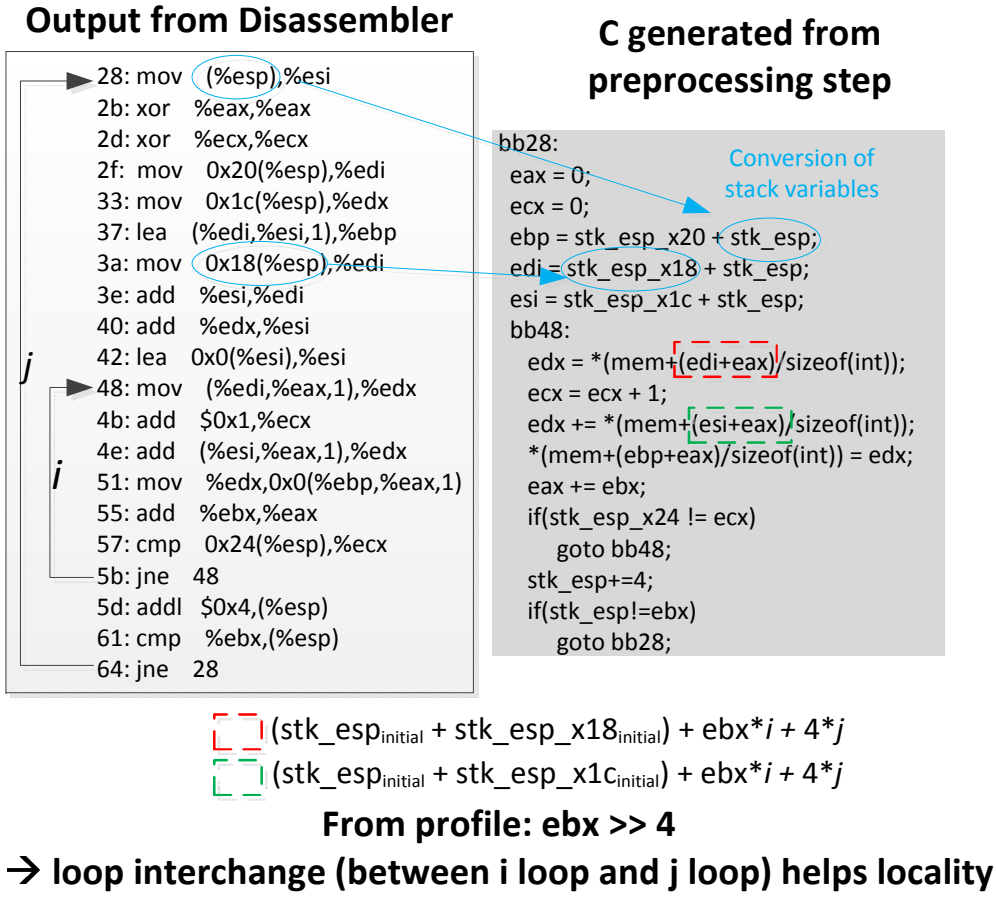


Figure 5.22: Loop Interchange Based on Coefficient Value from Past Execution Profile

Other Optimization for FPGA Accelerators

While our flow exploits parallelism in multiple different levels, there are many other potential optimizations incorporable into our framework. For instance, an important factor worth considering in generation of accelerators is the locality of data accesses. While multiple datapaths can easily saturate the off-chip memory bandwidth, there might be wastage if opportunities for data reuse are missed. Even in the absence of data reuse, requests for

contiguous segments or memory locations close to each other are more efficiently served by the caches and the memory subsystem. For programs which were written with locality in mind or compiled using toolchain capable of locality optimizations, the accelerators generated from their binaries would have already benefited from all the techniques applied. In the absence of those, our approach would still allow some helpful analysis and transformations to be performed during accelerator generation.

One example would be using loop interchange to increase locality. To generate the implementation with the smaller memory footprint (normalized by iteration count) at the inner loop level, we can leverage the run time values from the past profiles. As the targeted loops for our flow contain memory references whose addresses are affine expressions of the loop index variables, the coefficient for each of them decides how large of a stride each iteration takes, which conveniently approximate how "unlocal" the memory accesses are. Thus for a particular loop index variable, the smaller the coefficient, the deeper its corresponding loop level should go. This is illustrated in figure 5.22. Of course, as the interchange is performed based on past input data, for it to be valid, the resultant direction vectors can not have leading elements being " $>$ " when the accelerator is invoked with new data. To achieve this, the *online* phase would need to first check if the element being moved outward is " $>$ ". If it is, then we check if all the levels outside of it are " $=$ ". If the test result is affirmative, then the loop interchange has violated the original program order and the accelerator should not be invoked.

There are also other transformations which can be applied to extract additional parallelism or to achieve better data reuse. Loop fission and loop tiling, for instance, are techniques used in some optimizing compilers. Their incorporation is certainly feasible with our general approach and will add more dimensions to the design space.

Chapter 6

Conclusion

The main challenge in making FPGA a viable mainstream computing platform is to provide both good quality of results and ease of use to a large customer base. Conventional HLS serves as a great productivity tool for hardware engineers who have good understanding of the low level implementation details and the mapping process. It does not, unfortunately, provide a easy path for software programmers to quickly offload computation onto FPGA in an attempt to achieve better performance. Despite the relative ease of creating *a* hardware engine from high level languages, the advantage of programmable logic over general purpose processor can rarely be realized without extensive rewriting of the software code. In this thesis, we have constructed flows and performed experiments to validate a few possible approaches with which this gap between productivity and implementation quality can be narrowed.

The decoupled computational pipeline synthesis is an instance of a source to source transformation flow converting sequential programs to process networks. A partitioning algorithm was proposed to specifically cater to the characteristics of the FPGA platform. By converting a single threaded sequential program to a cascade of independently scheduled modules, the flow creates a pipeline through which data can stream. This paradigm is especially suitable for FPGA mapping and has proven to have significant performance advantage against CPU and direct HLS implementations. The complementary deadlock analysis scheme helps determine the required sizes of the FIFO channels to prevent artificial deadlocks. There are certain cases where the needed buffer space cannot be statically determined, which the analyzer can reveal as well. Process networks generated using other partitioning algorithms and mapped to platforms other than FPGAs may also benefit from the analysis framework we have proposed, as long as the usage of the channels follows certain rules. In essence, we have built the infrastructure with which a user can extract more performance from the FPGA without providing algorithm descriptions drastically different than typical processor-centric software kernels.

An even more aggressive attempt to reduce user involvement in generating accelerators was to use the program binaries as the starting point of high level synthesis. With the help of past execution profiles, memory level and coarse grained parallelism can all be exploited.

On the other hand, due to the probabilistic nature of this profile based approach, dynamic checking mechanisms are needed to ensure the parallelization performed during accelerator synthesis is actually valid. In general, the amount of computation involved in these run time checks scales up with the number of memory accesses performed by the executed binary. However, for loop nests with analyzable memory access patterns, the run time checks needed can be of constant cost. We devised a flow targeting these kernels, demonstrating the possibility of user-transparent accelerator generation and integration, into which past work in optimizing compilers can also be incorporated.

6.1 Future Explorations

For each part in this thesis, the potential for further investigation was also discussed. As the decoupled computational pipeline is used as the core template for the accelerator generation, different aspects of its generation can be further explored. For the algorithm of instruction partitioning, the current simplistic approach might have been overly aggressive in decoupling parts of the control data flow graph, resulting in large area overhead. As problem size gets even larger, our scheme would need to be modified to contain the cost of the generated pipeline. Of course, it is difficult to mathematically relate the final performance with how computation nodes are distributed, thus we did not formulate the partitioning process as a tractable optimization problem. Meanwhile, design space exploration requiring the actual mapping onto the FPGA platform would suffer from long compilation time. High level synthesis, while being relatively fast in providing schedules of compute operations, does not generate accurate estimate of final performance when unanalyzable interactions with the memory subsystem are involved. We believe a quick performance estimator incorporating data access overhead can be really helpful, especially if randomized heuristics are to be used for generating good solutions.

Another possible dimension for exploration is to vary the pipeline's flexibility by changing the depths of the FIFO channels. With the deadlock checking mechanism, we can protect against artificial deadlocks while making the trade-off between resource usage and rigidity of the pipeline's execution schedule. From the minimally required buffer sizes in the channels, we expect the initial increase in FIFO depths to improve the pipeline performance more significantly compare to when all the modules are sufficiently decoupled. Run time monitoring using simple occupancy counters may provide information about the usefulness of buffer spaces in individual channels, revealing opportunities for resource saving. Meanwhile, the same information can also be used to identify performance bottlenecks, guiding investment of hardware resources to specific modules, if more aggressive parallelization and better caching are implementable and have measurable benefits. It would be interesting, though challenging, to incorporate all these ideas and build an infrastructure performing automatic tuning of the pipelines as well as individually modules in them.

Finally for the pipelines generated from program binaries of regular computation kernels, there are other high level optimizations (e.g. loop interchange) which may boost performance

for some applications, assuming they are not already performed by the compiler producing the original binaries. The two phased approach proposed in this work provides a pathway for these existing techniques to be applied to the seemingly irregular memory access patterns generated from actual affine memory references, exposing the possibility of performing many optimizations in user transparent ways. Meanwhile, it is worth noting that this method is applicable to more than just the FPGA platforms. Given that modern SoCs already have multicore, SIMD and reconfigurable logic all integrated in the same chip, one possible future direction would be to apply our approach to all these different compute substrates. The comparison results might reveal new insight about the compatibility between components of a heterogeneous platform and various transformations. It may leads to a possible flow where program binaries can be decomposed and executed by different compute substrate collaboratively, in a user transparent manner. Of course, the current decoupled computational pipeline synthesis is designed to be rather FPGA specific. When different modules can be mapped to different compute substrate, the design space for the pipeline would become even larger, amplifying all the challenges we have mentioned previously.

6.2 Reflections and Closing Remarks

The methods proposed in this work try to make FPGA accessible to programmers or even end users who have little hardware knowledge. Thus the assumed starting points for accelerator generation are always single threaded software implementations written for CPU execution. There is no re-layout of data in the memory or rewriting of high level algorithms. Admittedly, the minimization of user effort we strive for in these flows may not be as important in more traditional FPGA applications, where the reconfigurable arrays are used as less risky, faster to market ASIC replacement. In those situations, system performance and resource utilization are supposed to be as optimized as possible. The design team, with more hardware expertise and engineering resources, can potentially create more efficient implementations by architecting the high level computation and data layout patterns with its specific FPGA as the target. HLS, in this context, allows for faster iteration and design space exploration of the core compute structure. Meanwhile, other components such as the on-chip interconnect, which are not suitable for HLS, can also be custom designed in HDL to complement the accelerators' operation. The speedups achievable can be higher than what have been realized in this study.

On the other hand, as reconfigurable computing platforms are getting more ubiquitous, the traditional way of performing FPGA design will not scale with the number of applications which can potentially benefit from hardware acceleration. A possible compromise is to create well optimized libraries targeting emerging applications. This seems to be the approach vendors have taken. For instance, the most recent development in deep learning has motivated both Intel and Xilinx to create FPGA IPs targeting developers in this space. From my own experience in this study, completely automated flows would always have limitations. Even though our approach has proven to be really helpful for certain types of computations,

we cannot claim it to be universally applicable. We do believe, however, the techniques introduced in this work can lower the barrier of using FPGAs and we are confident that that with future development of high level design methodologies, reconfigurable computing can benefit more users in more direct ways.

Bibliography

[1] Gordon E. Moore. "Readings in Computer Architecture". In: ed. by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: http://dl.acm.org/citation.cfm?id=333067.333074.

[2] Robert H Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *Solid-State Circuits, IEEE Journal of* 9.5 (1974), pp. 256–268.

[3] Andrew Danowitz et al. "CPU DB: Recording Microprocessor History". In: *Commun. ACM* 55.4 (Apr. 2012), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2133806.2133822. URL: http://doi.acm.org/10.1145/2133806.2133822.

[4] Robert Wilson et al. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*. Tech. rep. Stanford, CA, USA, 1994.

[5] Bill Blume et al. "Polaris: The Next Generation in Parallelizing Compilers". In: *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*. Springer-Verlag, Berlin/Heidelberg, 1994, pp. 10–1.

[6] H. Kasahara et al. "A multi-grain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor)". English. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Vol. 589. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 283–297. ISBN: 978-3-540-55422-6. DOI: 10.1007/BFb0038671. URL: http://dx.doi.org/10.1007/BFb0038671.

[7] CORPORATE Rice University. "High Performance Fortran Language Specification". In: *SIGPLAN Fortran Forum* 12.4 (Dec. 1993), pp. 1–86. ISSN: 1061-7264. DOI: 10.1145/174223.158909. URL: http://doi.acm.org/10.1145/174223.158909.

[8] Shih-Wei Liao et al. "SUIF Explorer: An Interactive and Interprocedural Parallelizer". In: *SIGPLAN Not.* 34.8 (May 1999), pp. 37–48. ISSN: 0362-1340. DOI: 10.1145/329366.301108. URL: http://doi.acm.org/10.1145/329366.301108.

[9] K. Kennedy, K.S. McKinley, and C.-W. Tseng. "Interactive parallel programming using the ParaScope Editor". In: *Parallel and Distributed Systems, IEEE Transactions on* 2.3 (1991), pp. 329–341. ISSN: 1045-9219. DOI: 10.1109/71.86108.

[10] I. Kuon and J. Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. ISSN: 0278-0070. DOI: 10.1109/TCAD.2006.884574.

[11] *Virtex-5 Family Overview*. Xilinx Inc. 2015.

[12] *Arria 10 Device Overview*. Altera Inc. 2016.

[13] Jeremy Fowers et al. "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '12. Monterey, California, USA: ACM, 2012, pp. 47–56. ISBN: 978-1-4503-1155-7. DOI: 10.1145/2145694.2145704. URL: http://doi.acm.org/10.1145/2145694.2145704.

[14] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. "High-throughput Bayesian Computing Machine with Reconfigurable Hardware". In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '10. Monterey, California, USA: ACM, 2010, pp. 73–82. ISBN: 978-1-60558-911-4. DOI: 10.1145/1723112.1723127. URL: http://doi.acm.org/10.1145/1723112.1723127.

[15] Xiang Tian and Khaled Benkrid. "High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU". In: *ACM Trans. Reconfigurable Technol. Syst.* 3.4 (Nov. 2010), 26:1–26:22. ISSN: 1936-7406. DOI: 10.1145/1862648.1862656. URL: http://doi.acm.org/10.1145/1862648.1862656.

[16] *FPGA Fundamentals*. National Instruments, 2012.

[17] *510T FPGA Accelerator Cards*. Nallatech. 2015.

[18] *Maxeler Technologies*. 2015.

[19] Melissa C Smith, Jeffery S Vetter, and Sadaf R Alam. "Scientific computing beyond CPUs: FPGA implementations of common scientific kernels". In:

[20] O. Lindtjorn et al. "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications". In: *Micro, IEEE* 31.2 (2011), pp. 41–49. ISSN: 0272-1732. DOI: 10.1109/MM.2011.17.

[21] J.W. Lockwood et al. "A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)". In: *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*. 2012, pp. 9–16. DOI: 10.1109/HOTI.2012.15.

[22] *Zynq-7000 All Programmable SoC Overview*. Xilinx Inc. 2013.

[23] *Cyclone V Hard Processor System Technical Reference Manual*. Altera Inc. 2014.

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

[25] P. Coussy et al. "An Introduction to High-Level Synthesis". In: *Design Test of Computers, IEEE* 26.4 (2009), pp. 8–17. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.69.

[26] J. Cong et al. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 30.4 (2011), pp. 473–491. ISSN: 0278-0070. DOI: 10.1109/TCAD.2011.2110592.

[27] *Catapult C Synthesis Users and Reference Manual*. Mentor Graphics. 2010.

[28] *Nios II C2H Compiler User Guide*. Altera Inc. 2009.

[29] *Vivado Design Suite High-level Synthesis*. Xilinx Inc. 2012.

[30] Andrew Canis et al. "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems". In: *ACM Trans. Embed. Comput. Syst.* 13.2 (Sept. 2013), 24:1–24:27. ISSN: 1539-9087. DOI: 10.1145/2514740. URL: http://doi.acm.org/10.1145/2514740.

[31] J. Villarreal et al. "Designing Modular Hardware Accelerators in C with ROCCC 2.0". In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. 2010, pp. 127–134. DOI: 10.1109/FCCM.2010.28.

[32] Jongsok Choi et al. "Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. 2012, pp. 17–24. DOI: 10.1109/FCCM.2012.13.

[33] Shaoyi Cheng et al. "Exploiting Memory-Level Parallelism in Reconfigurable Accelerators". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. 2012, pp. 157–160. DOI: 10.1109/FCCM.2012.35.

[34] Mario Roberto Barbacci. "Automated Exploration of the Design Space for Register-transfer (Rt) Systems." AAI7414648. PhD thesis. Pittsburgh, PA, USA, 1973.

[35] A. Parker et al. "The CMU Design Automation System: An Example of Automated Data Path Design". In: *Proceedings of the 16th Design Automation Conference*. DAC '79. San Diego, CA, USA: IEEE Press, 1979, pp. 73–80. URL: http://dl.acm.org/citation.cfm?id=800292.811694.

[36] P. G. Paulin and J. P. Knight. "Force-directed Scheduling in Automatic Data Path Synthesis". In: *Proceedings of the 24th ACM/IEEE Design Automation Conference*. DAC '87. Miami Beach, Florida, USA: ACM, 1987, pp. 195–202. ISBN: 0-8186-0781-5. DOI: 10.1145/37888.37918. URL: http://doi.acm.org/10.1145/37888.37918.

[37] Nohbyung Park and Alice Parker. "Sehwa: A Program for Synthesis of Pipelines". In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. DAC '86. Las Vegas, Nevada, USA: IEEE Press, 1986, pp. 454–460. ISBN: 0-8186-0702-5. URL: http://dl.acm.org/citation.cfm?id=318013.318086.

[38] Raul Camposano and Wayne H. Wolf, eds. *High-Level VLSI Synthesis*. Norwell, MA, USA: Kluwer Academic Publishers, 1991. ISBN: 0792391594.

[39] P.G. Paulin and J.P. Knight. "Scheduling and Binding Algorithms for High-Level Synthesis". In: *Design Automation, 1989. 26th Conference on*. 1989, pp. 1–6. DOI: 10.1109/DAC.1989.203360.

[40] S. Devadas and A.R. Newton. "Algorithms for hardware allocation in data path synthesis". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 8.7 (1989), pp. 768–781. ISSN: 0278-0070. DOI: 10.1109/43.31534.

[41] Chia-Jeng Tseng and D.P. Siewiorek. "Automated Synthesis of Data Paths in Digital Systems". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 5.3 (1986), pp. 379–395. ISSN: 0278-0070. DOI: 10.1109/TCAD.1986.1270207.

[42] Cheng-Tsung Hwang, J.-H. Lee, and Yu-Chin Hsu. "A formal approach to the scheduling problem in high level synthesis". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 10.4 (1991), pp. 464–475. ISSN: 0278-0070. DOI: 10.1109/43.75629.

[43] W. Grass, M. Mutz, and W.-D. Tiedemann. "High level synthesis based on formal methods". In: *EUROMICRO 94. System Architecture and Integration. Proceedings of the 20th EUROMICRO Conference.* 1994, pp. 83–91. DOI: 10.1109/EURMIC.1994.390403.

[44] C Karfa et al. "A Formal Verification Method of Scheduling in High-level Synthesis". In: *Proceedings of the 7th International Symposium on Quality Electronic Design*. ISQED '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 71–78. ISBN: 0-7695-2523-7. DOI: 10.1109/ISQED.2006.10. URL: http://dx.doi.org/10.1109/ISQED.2006.10.

[45] D. Genin et al. "DSP specification using the Silage language". In: *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on*. 1990, 1056–1060 vol.2. DOI: 10.1109/ICASSP.1990.116097.

[46] L. Claesen et al. "Automatic synthesis of signal processing benchmark using the CATHEDRAL silicon compilers". In: *Custom Integrated Circuits Conference, 1988., Proceedings of the IEEE 1988*. 1988, pp. 14.7/1–14.7/4. DOI: 10.1109/CICC.1988.20869.

[47] Chi-Min Chu et al. "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications". In: *IN PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN*. 1989, pp. 432–435.

[48] David W. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-569252-0.

[49] *Mentor Graphics Monet User Manual*. Mentor Graphics Inc. 1999.

[50] *C-to-Silicon Compiler High-Level Synthesis*. Cadence Inc., 2008.

[51] *Cynthesizer User Guide*. Forte Design System. 2003.

[52] V. Kathail et al. "PICO: automatically designing custom computers". In: *Computer* 35.9 (2002), pp. 39–47. ISSN: 0018-9162. DOI: 10.1109/MC.2002.1033026.

[53] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *Design Test of Computers, IEEE* 26.4 (2009), pp. 18–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.83.

[54] Greg Stitt, Frank Vahid, and Walid Najjar. "A Code Refinement Methodology for Performance-improved Synthesis from C". In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '06. San Jose, California: ACM, 2006, pp. 716–723. ISBN: 1-59593-389-1. DOI: 10.1145/1233501.1233649. URL: http://doi.acm.org/10.1145/1233501.1233649.

[55] Yun Liang et al. "High-level Synthesis: Productivity, Performance, and Software Constraints". In: *JECE* 2012 (Jan. 2012), 1:1–1:1. ISSN: 2090-0147. DOI: 10.1155/2012/649057. URL: http://dx.doi.org/10.1155/2012/649057.

[56] G. Inggs et al. "Is high level synthesis ready for business? A computational finance case study". In: *Field-Programmable Technology (FPT), 2014 International Conference on*. 2014, pp. 12–19. DOI: 10.1109/FPT.2014.7082747.

[57] David W. Wall. "Limits of Instruction-level Parallelism". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 176–188. ISBN: 0-89791-380-9. DOI: 10.1145/106972.106991. URL: http://doi.acm.org/10.1145/106972.106991.

[58] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 318–328. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54022. URL: http://doi.acm.org/10.1145/53990.54022.

[59] B. Ramakrishna Rau. "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: ACM, 1994, pp. 63–74. ISBN: 0-89791-707-3. DOI: 10.1145/192724.192731. URL: http://doi.acm.org/10.1145/192724.192731.

[60] P. Tirumalai, M. Lee, and M. Schlansker. "Parallelization of Loops with Exits on Pipelined Architectures". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 200–212. ISBN: 0-89791-412-0. URL: http://dl.acm.org/citation.cfm?id=110382.110438.

[61] Vicki H. Allan et al. "Software Pipelining". In: *ACM Comput. Surv.* 27.3 (Sept. 1995), pp. 367–432. ISSN: 0360-0300. DOI: 10.1145/212094.212131. URL: http://doi.acm.org/10.1145/212094.212131.

[62] Kemal Ebcioğlu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps". In: *Proceedings of the 20th Annual Workshop on Microprogramming*. MICRO 20. Colorado Springs, Colorado, USA: ACM, 1987, pp. 69–79. ISBN: 0-89791-250-0. DOI: 10.1145/255305.255317. URL: http://doi.acm.org/10.1145/255305.255317.

[63] Corinne Ancourt and François Irigoin. "Scanning Polyhedra with DO Loops". In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '91. Williamsburg, Virginia, USA: ACM, 1991, pp. 39–50. ISBN: 0-89791-390-6. DOI: 10.1145/109625.109631. URL: http://doi.acm.org/10.1145/109625.109631.

[64] Cdric Bastoul et al. "Putting Polyhedral Loop Transformations to Work". English. In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Vol. 2958. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 209–225. ISBN: 978-3-540-21199-0. DOI: 10.1007/978-3-540-24644-2_14. URL: http://dx.doi.org/10.1007/978-3-540-24644-2_14.

[65] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think". English. In: *Compiler Construction*. Ed. by Rajiv Gupta. Vol. 6011. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 283–303. ISBN: 978-3-642-11969-9. DOI: 10.1007/978-3-642-11970-5_16. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_16.

[66] Sylvain Girbal et al. "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies". In: *International Journal of Parallel Programming* 34.3 (2006), pp. 261–317.

[67] William Pugh. "Uniform techniques for loop optimization". In: *Proceedings of the 5th international conference on Supercomputing*. ACM. 1991, pp. 341–352.

[68] Michael E Wolf and Monica S Lam. "A loop transformation theory and an algorithm to maximize parallelism". In: *Parallel and Distributed Systems, IEEE Transactions on* 2.4 (1991), pp. 452–471.

[69] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-performance Computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: http://doi.acm.org/10.1145/197405.197406.

[70] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. "Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 86–100. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125886. URL: http://doi.acm.org/10.1145/125826.125886.

[71] Paul Feautrier. "Semantical analysis and mathematical programming; application to parallelization and vectorization". In: *Workshop on Parallel and Distributed Algorithms, Bonas*. Citeseer. 1988, pp. 309–320.

[72] Wei Zuo et al. "Improving Polyhedral Code Generation for High-level Synthesis". In: *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '13. Montreal, Quebec, Canada: IEEE Press, 2013, 15:1–15:10. ISBN: 978-1-4799-1417-3. URL: http://dl.acm.org/citation.cfm?id=2555692.2555707.

[73] Louis-Noel Pouchet et al. "Polyhedral-based Data Reuse Optimization for Configurable Computing". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: ACM, 2013, pp. 29–38. ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435273. URL: http://doi.acm.org/10.1145/2435264.2435273.

[74] Q. Liu et al. "Combining Data Reuse With Data-Level Parallelization for FPGA-Targeted Hardware Compilation: A Geometric Programming Framework". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.3 (2009), pp. 305–315. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2013541.

[75] Tim Harriss et al. "Compilation From Matlab to Process Networks Realized in FPGA." In: *Design Autom. for Emb. Sys.* 7.4 (2002), pp. 385–403.

[76] Sven van Haastregt and Bart Kienhuis. "Automated Synthesis of Streaming C Applications to Process Networks in Hardware". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 890–893. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.org/citation.cfm?id=1874620.1874837.

[77] F. Mayer-Lindenberg. "High-Level FPGA Programming through Mapping Process Networks to FPGA Resources". In: *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*. 2009, pp. 302–307. DOI: 10.1109/ReConFig.2009.73.

[78] A. Papakonstantinou et al. "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs". In: *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*. 2009, pp. 35–42. DOI: 10.1109/SASP.2009.5226333.

[79] I. Lebedev et al. "MARC: A Many-Core Approach to Reconfigurable Computing". In: *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*. 2010, pp. 7–12. DOI: 10.1109/ReConFig.2010.49.

[80] *Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS*. Xilinx Inc. 2013.

[81] Eric S. Chung, James C. Hoe, and Ken Mai. "CoRAM: An In-fabric Memory Architecture for FPGA-based Computing". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 97–106. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950435. URL: http://doi.acm.org/10.1145/1950413.1950435.

[82] Gabriel Weisz and James C. Hoe. "C-to-CoRAM: Compiling Perfect Loop Nests to the Portable CoRAM Abstraction". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: ACM, 2013, pp. 221–230. ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435302. URL: http://doi.acm.org/10.1145/2435264.2435302.

[83] Jorge E. Carrillo and Paul Chow. "The Effect of Reconfigurable Units in Superscalar Processors". In: *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. FPGA '01. Monterey, California, USA: ACM, 2001, pp. 141–150. ISBN: 1-58113-341-3. DOI: 10.1145/360276.360328. URL: http://doi.acm.org/10.1145/360276.360328.

[84] S. Hauck et al. "The Chimaera reconfigurable functional unit". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.2 (2004), pp. 206–217. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2003.821545.

[85] A. Lodi et al. "A VLIW processor with reconfigurable instruction set for embedded applications". In: *IEEE Journal of Solid-State Circuits* 38.11 (2003), pp. 1876–1886. ISSN: 0018-9200. DOI: 10.1109/JSSC.2003.818292.

[86] R. Razdan and M. D. Smith. "A high-performance microarchitecture with hardware-programmable functional units". In: *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*. 1994, pp. 172–180. DOI: 10.1109/MICRO.1994.717456.

[87] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. "A C Compiler for a Processor with a Reconfigurable Functional Unit". In: *Field-Programmable Gate Arrays, International ACM Symposium on* 0 (2000), pp. 95–100. DOI: http://doi.ieeecomputersociety.org/10.1109/FPGA.2000.2.

[88] J. R. Hauser and J. Wawrzynek. "Garp: a MIPS processor with a reconfigurable coprocessor". In: *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. 1997, pp. 12–21. DOI: 10.1109/FPGA.1997.624600.

[89] Ming-Hau Lee et al. "Design and Implementation of the MorphoSys Reconfigurable ComputingProcessor". In: *J. VLSI Signal Process. Syst.* 24.2-3 (Mar. 2000), pp. 147–164. ISSN: 0922-5773. DOI: 10.1023/A:1008189221436. URL: http://dx.doi.org/10.1023/A:1008189221436.

[90] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs*. Xilinx Inc., 2004.

[91] *Alpha Data ADM-PCIE-8K5*. Alpha Data Parallel Systems, 2015.

[92] *TR5-F40W User Manual*. Terasic Inc., 2015.

[93] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. "The Garp architecture and C compiler". In: *Computer* 33.4 (2000), pp. 62–69. ISSN: 0018-9162. DOI: 10.1109/2.839323.

[94] Deshanand P. Singh, Tomasz S. Czajkowski, and Andrew Ling. "Harnessing the Power of FPGAs Using Altera's OpenCL Compiler". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: ACM, 2013, pp. 5–6. ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435268. URL: http://doi.acm.org/10.1145/2435264.2435268.

[95] Jiunn-Yeu Chen et al. "A static binary translator for efficient migration of ARM-based applications". In: *Workshop on Optimizations for DSP and Embedded Systems*. Citeseer. 2008.

[96] Bor-Yeh Shen et al. "LLBT: an LLVM-based static binary translator". In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2012, pp. 51–60.

[97] Kemal Ebcioğlu and Erik R. Altman. "DAISY: Dynamic Compilation for 100Compatibility". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ISCA '97. Denver, Colorado, USA: ACM, 1997, pp. 26–37. ISBN: 0-89791-901-7. DOI: 10.1145/264107.264126. URL: http://doi.acm.org/10.1145/264107.264126.

[98] James C. Dehnert et al. "The Transmeta Code Morphing&Trade; Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '03. San Francisco, California, USA: IEEE Computer Society, 2003, pp. 15–24. ISBN: 0-7695-1913-X. URL: http://dl.acm.org/citation.cfm?id=776261.776263.

[99] Leonid Baraz et al. "IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems". In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 191–. ISBN: 0-7695-2043-X. URL: http://dl.acm.org/citation.cfm?id=956417.956550.

[100] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *SIGPLAN Not.* 35.5 (May 2000), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/358438.349303. URL: http://doi.acm.org/10.1145/358438.349303.

[101] Dean Deaver, Rick Gorton, and Norm Rubin. "Wiggins/Redstone: An on-line program specializer". In: *Proceedings of the IEEE Hot Chips XI Conference*. 1999.

[102] Ali-Reza Adl-Tabatabai et al. "Fast, effective code generation in a just-in-time Java compiler". In: *ACM SIGPlAN Notices*. Vol. 33. 5. ACM. 1998, pp. 280–290.

[103] Luca Gherardi, Davide Brugali, and Daniele Comotti. "A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark". In: *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. SIMPAR'12. Tsukuba, Japan: Springer-Verlag, 2012, pp. 161–172. ISBN: 978-3-642-34326-1. DOI: 10.1007/978-3-642-34327-8_17. URL: http://dx.doi.org/10.1007/978-3-642-34327-8_17.

[104] Jing Yang et al. "Feasibility of dynamic binary parallelization". In: *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*. 2011.

[105] Efe Yardımcı and Michael Franz. "Dynamic parallelization and vectorization of binary executables on hierarchical platforms". In: *Journal of Instruction-Level Parallelism* 10 (2008), pp. 1–24.

[106] N. Hallou et al. "Dynamic re-vectorization of binary code". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. 2015, pp. 228–237. DOI: 10.1109/SAMOS.2015.7363680.

[107] Keith Cooper, Anshuman Dasgupta, and Ken Kennedy. "Vizer: A system to vectorize intel x86 binaries". In: *Intl. Symp. on Computer Architecture, Santa Fe, NM*. Vol. 64. 2002.

[108] Aparna Kotha et al. "Automatic Parallelization in a Binary Rewriter". In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 547–557. ISBN: 978-0-7695-4299-7. DOI: 10.1109/MICRO.2010.27. URL: http://dx.doi.org/10.1109/MICRO.2010.27.

[109] Guilherme Ottoni et al. "Automatic Thread Extraction with Decoupled Software Pipelining". In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 38. Barcelona, Spain: IEEE Computer Society, 2005, pp. 105–118. ISBN: 0-7695-2440-0. DOI: 10.1109/MICRO.2005.13. URL: http://dx.doi.org/10.1109/MICRO.2005.13.

[110] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565.

[111] Andrew Putnam et al. "Performance and Power of Cache-based Reconfigurable Computing". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 395–405. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555804. URL: http://doi.acm.org/10.1145/1555754.1555804.

[112] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0-7695-2102-9. URL: http://portal.acm.org/citation.cfm?id=977395.977673&coll=GUIDE&dl=GUIDE&CFID=48424181&CFTOKEN=16724426.

[113] John Wawrzynek. "EECS150-Digital Design: Lecture 3Field Programmable Gate Arrays (FPGAs)". In: *Jan* 26 (2010), pp. 1–10.

[114] ILOG, Inc. *ILOG CPLEX: High-performance software for mathematical programming and optimization*. See http://www.ilog.com/products/cplex/. 2006.

[115] David A. Patterson. "Latency Lags Bandwith". In: *Commun. ACM* 47.10 (Oct. 2004), pp. 71–75. ISSN: 0001-0782. DOI: 10.1145/1022594.1022596. URL: http://doi.acm.org/10.1145/1022594.1022596.

[116] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[117] Zhi Guo, Walid Najjar, and Betul Buyukkurt. "Efficient Hardware Code Generation for FPGAs". In: *ACM Trans. Archit. Code Optim.* 5.1 (May 2008), 6:1–6:26. ISSN: 1544-3566. DOI: 10.1145/1369396.1369402. URL: http://doi.acm.org/10.1145/1369396.1369402.

[118] E. Dahlhaus et al. "The Complexity of Multiterminal Cuts". In: *SIAM Journal on Computing* 23.4 (1994), pp. 864–894. DOI: 10.1137/S0097539792225297. eprint: http://dx.doi.org/10.1137/S0097539792225297. URL: http://dx.doi.org/10.1137/S0097539792225297.

[119] E. A. de Kock et al. "YAPI: Application Modeling for Signal Processing Systems". In: *Proceedings of the 37th Annual Design Automation Conference*. DAC '00. Los Angeles, California, USA: ACM, 2000, pp. 402–405. ISBN: 1-58113-187-9. DOI: 10.1145/337292.337511. URL: http://doi.acm.org/10.1145/337292.337511.

[120] Thomas M. Parks. "Bounded Scheduling of Process Networks". PhD thesis. EECS Department, University of California, Berkeley, 1995. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html.

[121] Marc Geilen and Twan Basten. "Requirements on the Execution of Kahn Process Networks". In: *Proceedings of the 12th European Conference on Programming*. ESOP'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 319–334. ISBN: 3-540-00886-1. URL: http://dl.acm.org/citation.cfm?id=1765712.1765736.

[122] Joseph Tobin Buck. "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model". AAI9431898. PhD thesis. 1993.

[123] Joseph T Buck et al. "Ptolemy: A framework for simulating and prototyping heterogeneous systems". In: (1994).

[124] T. Basten and J. Hoogerbrugge. "Efficient execution of process networks". In: *Proc. of Communicating Process Architectures 2001*. Bristol, UK, 2001, pp. 1–14.

[125] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph". In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84. DOI: 10.1137/0204007. eprint: http://dx.doi.org/10.1137/0204007. URL: http://dx.doi.org/10.1137/0204007.

[126] *Dyninst Programmer's Guide*. Computer Science Department, University of Wisconsin-Madison & University of Maryland. 2015.

[127] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (Nov. 2000), pp. 317–329. ISSN: 1094-3420. DOI: 10.1177/109434200001400404. URL: http://dx.doi.org/10.1177/109434200001400404.

[128] *Intel starts baking speedy FPGAs into chips*. IDG News Service, 2016. URL: http://www.pcworld.com/article/3055526/intel-starts-baking-speedy-fpgas-into-chips.html.

[129] *SDAccel Development Environment User Guide*. Xilinx. 2015.

[130] Jason D. Hiser et al. "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 61–73. ISBN: 0-7695-2764-7. DOI: 10.1109/CGO.2007.10. URL: http://dx.doi.org/10.1109/CGO.2007.10.

[131] U. Banerjee. "Data dependence in ordinary programs". MA thesis. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Nov. 1976.

[132] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988. ISBN: 0898382890.

[133] Michael Joseph Wolfe. "Techniques for improving the inherent parallelism in programs". MAS. PhD thesis. 1978. URL: http://opac.inria.fr/record=b1000180.

[134] William Pugh. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 4–13. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125848. URL: http://doi.acm.org/10.1145/125826.125848.

[135] David Sheffield. "Three Fingered Jack: Productively Addressing Platform Diversity". PhD thesis. EECS Department, University of California, Berkeley, 2013. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-185.html.

[136] Jongsok Choi, S. Brown, and J. Anderson. "From software threads to parallel hardware in high-level synthesis for FPGAs". In: *Field-Programmable Technology (FPT), 2013 International Conference on*. 2013, pp. 270–277. DOI: 10.1109/FPT.2013.6718365.

[137] M. Oyamada et al. "Software Performance Estimation in MPSoC Design". In: *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. ASP-DAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 38–43. ISBN: 1-4244-0629-3. DOI: 10.1109/ASPDAC.2007.357789. URL: http://dx.doi.org/10.1109/ASPDAC.2007.357789.

[138] Mitesh R. Meswani et al. "Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators". In: *Int. J. High Perform. Comput. Appl.* 27.2 (May 2013), pp. 89–108. ISSN: 1094-3420. DOI: 10.1177/1094342012468180. URL: http://dx.doi.org/10.1177/1094342012468180.

[139] Georgios Tournavitis et al. "Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 177–187. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542496. URL: http://doi.acm.org/10.1145/1542476.1542496.

[140] Arnamoy Bhattacharyya. "Do Inputs Matter?: Using Data-dependence Profiling to Evaluate Thread Level Speculation in BG/Q". In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. PACT '13. Edinburgh, Scotland, UK: IEEE Press, 2013, pp. 401–402. ISBN: 978-1-4799-1021-2. URL: http://dl.acm.org/citation.cfm?id=2523721.2523775.

[141] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. "Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 33–42. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370824. URL: http://doi.acm.org/10.1145/2370816.2370824.

[142] Thomas B. Jablin et al. "Automatic CPU-GPU Communication Management and Optimization". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 142–151. ISSN: 0362-1340. DOI: 10.1145/1993316.1993516. URL: http://doi.acm.org/10.1145/1993316.1993516.

[143] Thomas B. Jablin et al. "Dynamically Managed Data for CPU-GPU Architectures". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: ACM, 2012, pp. 165–174. ISBN: 978-1-4503-1206-6. DOI: 10.1145/2259016.2259038. URL: http://doi.acm.org/10.1145/2259016.2259038.