Exploiting Memory-level Parallelism in Reconfigurable Accelerators

Shaoyi Cheng

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-40
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-40.html

May 1, 2013

Acknowledgement

I would like to thank my graduate advisor, Prof. John Wawrzynek for his support and guidance. Thanks to Prof. Krste Asanovic for his time and feedback.

Special thanks goes to Haojun Liu and Simon Scott who were vital in completing the implementation of the design described in the report. Thanks to Prof. Mingjie Lin for his great insight and help in the process of completing this project. Many thanks to my other colleagues, Greg Gibeling, Alex Krasnov and James Martin, for interesting discussion and feedback on this project.

Exploiting Memory-level Parallelism in Reconfigurable Accelerators

by Shaoyi Cheng

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek

Research Advisor

(Date)

* * * * * * *

Professor Krste Asanovic

Second Reader

(Date)

Exploiting Memory-level Parallelism
in Reconfigurable Accelerators

by

Shaoyi Cheng

Abstract

Exploiting Memory-level Parallelism
in Reconfigurable Accelerators

by

Shaoyi Cheng

Master of Science in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor John Wawrzynek, Advisor

As memory accesses increasingly limit the overall performance of reconfigurable accelerators, it is important for high-level synthesis (HLS) flows to adopt a systematic way to discover and exploit memory-level parallelism. This work develops 1) a framework where parallelism between memory accesses can be revealed from runtime profiles of applications and provided to a high level synthesis flow, and 2) a novel multiaccelerator/multi-cache architecture to support parallel memory accesses, taking advantage of the high aggregated memory bandwidth found in modern FPGA devices. Our experimental results have shown that for 10 accelerators generated from 9 benchmark applications, circuits using our proposed memory structure achieve on average 51% improved performance over accelerators using a traditional memory interface. We believe that our study represents a solid advance towards achieving memory-parallel embedded computing on hybrid CPU+FPGA platforms.

Professor John Wawrzynek
Project Advisor

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my graduate advisor, Prof. John Wawrzynek for his support and guidance. Thanks to Prof. Krste Asanovic for his time and feedback.

Special thanks goes to Haojun Liu and Simon Scott who were vital in completing the implementation of the design described in the report. Thanks to Prof. Mingjie Lin for his great insight and help in the process of completing this project. Many thanks to my other colleagues, Greg Gibeling, Alex Krasnov and James Martin, for interesting discussion and feedback on this project.

Chapter 1

Introduction

Reconfigurable devices such as FPGAs contain computing elements of extremely flexible granularities, ranging from elementary logic gates to complete arithmetic-logic units such as DSP blocks. This characteristic of reconfigurable devices gives them huge potential in utilizing application-specific parallelism — computation can be spatially mapped to the device, enabling much higher operation throughput than processor-centric platforms.

Unfortunately, to fully realize the FPGA's performance and efficiency potential, cumbersome HDL programming and laborious manual optimizations are often required. Specifically, programming FPGAs demands skills and techniques well outside the application-oriented expertise of many developers, forcing them to step beyond their traditional programming abstractions and embrace hardware design concepts, such as clock management, state machines, pipelining, and device-specific memory management. The emergence of high level synthesis flows can potentially alleviate this difficulty and make the performance benefits of FPGA computing much more attainable. In particular, the CPU+FPGA hybrid platform, when used in conjunction with the HLS tools, allows the developers to readily offload the most compute-intensive portions of the applications to hardware, while using the CPU for control and management purposes. As a result, the system designers can better leverage programmable logic to optimize and differentiate their solutions.

In this context, the methodology the HLS tools use for hardware generation would have a huge impact on final system performance. Traditionally, HLS tools have focused on the extraction of instruction level parallelism and loop level parallelism while little effort has been invested in the optimization of the associated memory structures. Consequently, the datapath optimization is often restricted by the serialization of memory accesses, imposed by the conventional memory model. Meanwhile, a single monolithic memory usage model often fails to take advantage of the large number of block RAMs and the high memory bandwidth on the FPGA devices. In order to better realize the potential of the FPGA platforms and push the performance of the generated accelerators even further, the memory model used in the CPU+accelerator solutions should be reexamined.

1.1 Project Overview

The objective of this project is to devise a systematic approach to discover the parallelism between memory accesses, and then exploit this parallelism for performance enhancement in reconfigurable accelerators. More specifically, our work includes:

- the infrastructure used to capture and examine the runtime traces of applications, from which the candidates for acceleration are identified and independence between memory accesses is established.
- a multi-cache architecture that accomodates parallel accesses to different part of the address space, with automatic mechanisms to ensure the coherence of the system.
- a tool for generating HDL representation of accelerators for kernel discovered in the benchmark applications.
- the validation of our approach by comparing the performance of the accelerators with a single memory interface versus those with the new multi-cache interface.

1.2 Related Work

It is widely understood that mapping software applications to hardware can greatly improve the overall system performance and energy consumption. The research community has tried to facilitate this process by creating high-level synthesis (HLS) flows, where dataflow graphs can be systematically translated to hardware structures [1], [2]. Recently, in CPU+FPGA systems, HLS has attracted significant interest on both academic and commercial fronts [3], [4]. Many of these tools use profiling information to discover kernels, whose source code is then transformed to FPGA circuits. The *Warp processor* [5], on the other hand, performs translation from the binary running on the processor, directly utilizing the dynamic profile of the applications. Our work fits into the overall system architecture of these works, where the CPU performs control tasks while the computation is offloaded to accelerators. However, instead of focusing on converting software to hardware, we investigated the effect of memory-level parallelism in accelerator performance and how the benefit can be obtained. A previous project did attempt to address memory access parallelism for reconfigurable accelerators [6]. Their approach was to have the user specify independent memory accesses, such that neither complex alias analysis nor hardware support for cache coherence is needed. But ultimately this approach restricts the range of suitable applications to ones that have easily determined and static memory access patterns—such as those found in scientific computing. We take a much more general approach and rely solely on runtime profiling to determine the memory access behavior and therefore potentially address a wider range of applications.

Chapter 2

Motivations

To better understand how our solution fits into its context, it is necessary to first examine how the accelerators and the memory subsystem interact on reconfigurable platforms. At the same time, by observing the behavior of application code, we can identify the potential benefit of our approach.

2.1 Accelerators and Memory on FPGAs

Pushing for broader acceptance of FPGA-based SoC designs, the FPGA vendors have provided system design tools [7], [8] which make possible the integration of application-specific accelerators with their microprocessor cores. As both the accelerators and the processor need to have access to common storage, they often share the memory interface as well. The system shown in Figure 2.1 represents the architecture paradigm used in many previous works [3]–[5].

The cache interface in this system can satisfy one memory access every cycle, be it from the accelerator or the processor. Designed to complement the execution of program on the processor, this structure does not work well with accelerators and can potentially limit the performance achievable. As accelerators parallelize compute operations aggressively, they need to be fed with data at a much faster rate, which may exceed the capability of the shown interface. At the same time, a single cache interface fails to take advantage of the fact that multiple accesses can be independently fired off to the memory subsystem if they target different parts of the address space. This is a fact which the HLS tools can make good use of to generate high-performance accelerators. However, as we deviate from the conventional model of a centralized monolithic memory, extra effort is required to establish the independence between accesses. This may require the users to partition the memory in the source code, as in the case of [6], [9], or the application of complicated alias analysis, which is conservative and may not reveal all opportunities for access parallelization.

To address these challenges, this work has proposed a multi-cache architecture, and a complementary profile-based memory analysis flow to discover access parallelization. Our tool flow would take advantage of the new system template, which puts the underutilized memory bandwidth on FPGA devices to good use.

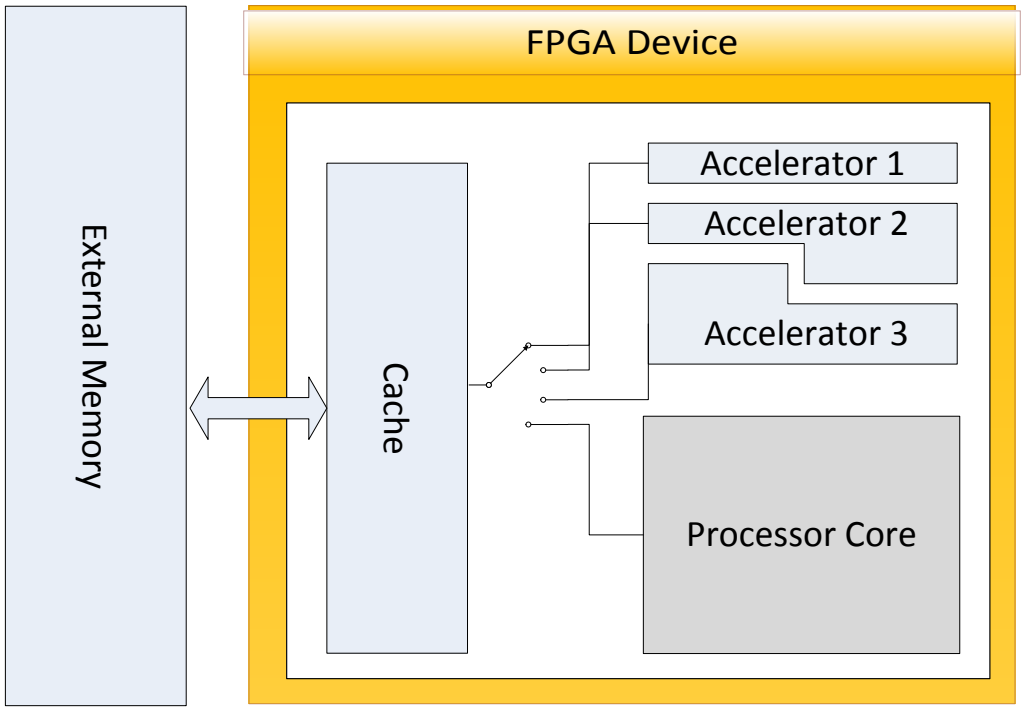

Figure 2.1. CPU+Accelerators System with Shared Memory Interface

2.2 Motivating Example

Before delving into details of the method for finding parallelizable memory accesses and our new architecture, we first demonstrate their potential benefit using a program segment extracted from jpeg encoder running on an x86 machine. Figure 2.2 shows a loop that is executed thousands of times when the program runs. It is an ideal candidate for hardware acceleration—a so-called computation kernel. In the code segment, out of the nine memory accessing instructions, six target constant addresses on the stack (constant offset from the base pointer ebp). They can be optimized away and turned into wires/registers during high-level synthesis. The remaining three references at 31b7, 31bf, and 31db would be kept as actual memory references when the hardware is generated. If we assume a single memory interface is used and maintains the ordering of memory accesses in the original program, the store at 31db for the current iteration has to happen before the load at 31b7 for the next iteration. When the HLS tool tries to pipeline the loop iterations, this constraint would not allow the minimal initiation interval to drop below three. On the other hand, if we can establish that the memory locations accessed by 31b7 and 31bf are disjoint from those by 31db, the load at 31b7 of the next iteration can be started before the store at 31db for the previous iteration completes. The minimal initiation interval for loop iterations would no longer be constrained by the RAW dependencies between the store and load instructions, and can potentially be reduced to one. Of course, along with showing the independence between the memory locations accessed by each of the three instructions, the physical implementation would also need to satisfy three memory request simultaneously in order to achieve the rate of one iteration completion per cycle. The effect of this memory level parallelism is shown in figure 2.3.

```
31b7:   mov     (%esi),%eax          ; mem load
31b9:   mov     -0x140(%ebp),%ecx
31bf:   movswl  (%ecx,%eax,2),%eax   ; mem load
31c3:   mov     %eax,%edx
31c5:   sar     $0x1f,%edx
31c8:   xor     %edx,%eax
31ca:   sub     %edx,%eax
31cc:   movzbl  -0x144(%ebp),%ecx
31d3:   sar     %cl,%eax
31d5:   mov     -0x164(%ebp),%edx
31db:   mov     %eax,(%edx)          ; mem store
31dd:   cmp     $0x1,%eax
31e0:   jne     0x31e8
31e2:   mov     %ebx,-0x154(%ebp)
31e8:   add     $0x1,%ebx
313b:   add     $0x4,%esi
31ee:   addl    $0x4,-0x164(%ebp)
31f5:   cmp     %ebx,-0x148(%ebp)
31fb:   jge     0x31b7
```

Figure 2.2. Binary Segment from JPEG Encoder.

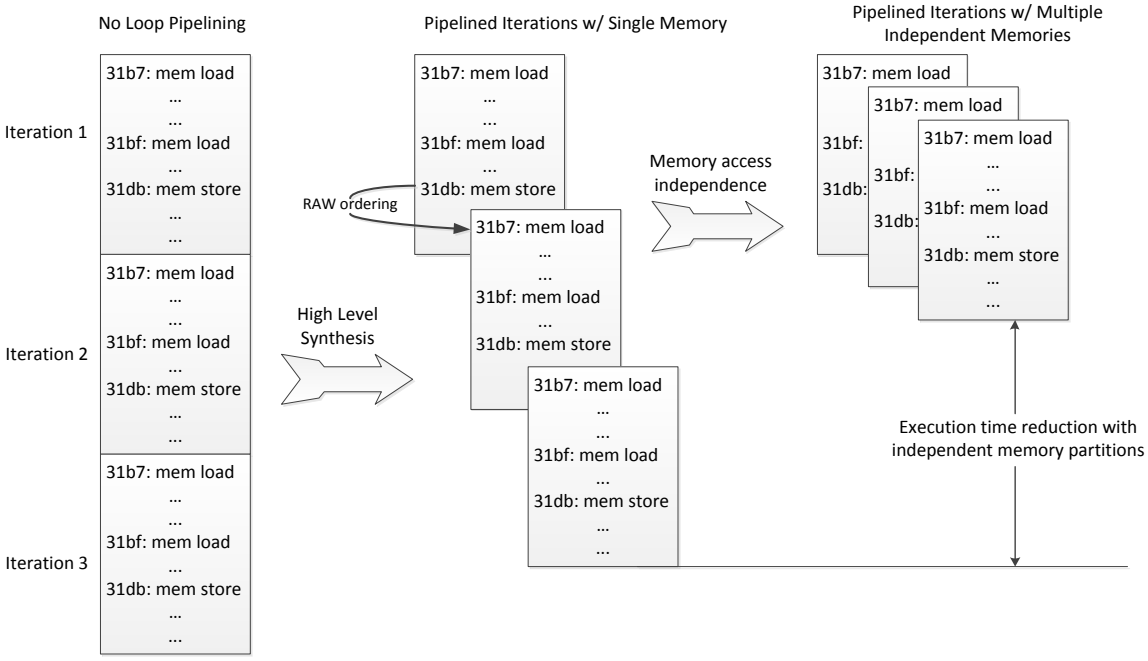

Figure 2.3. Performance Implication from Independence between Memory Accesses

This runtime reduction shown is of course application-dependent. The nature of the dataflow in the accelerated kernel, the memory access pattern and the actual physical implementation of the memory subsystem would all affect the final performance of the system. We will evaluate our approach with a set of applications in chapter 7, after describing our tool flow and architecture.

Chapter 3

Analysis Framework

The analysis framework in our project performs two important tasks prior to generation of any hardware. It first selects candidates for acceleration and then within each accelerated region, it finds independence between memory accesses.

3.1 Profiling infrastructure and Code Selection for Acceleration

As mentioned in chapter 1, our analysis framework is entirely profile-based. We leveraged the QEMU emulator infrastructure [10] to capture the stream of instructions executed at runtime, along with the memory addresses accessed by them. Using different inputs, multiple instances of the same application are executed. The profiles of all the runs are recorded and considered for our study.

To identify which sequences of basic blocks are being repeatedly executed, we devised a mechanisms similar to [11]. The most frequently executed loops are then selected for acceleration, approximating an optimal partitioning in terms of the ratio of performance benefit over hardware resource consumption. The main focus of our work though, is to identify parallelism in memory references within each of these accelerated loops.

3.2 Generation of Memory Access Partitions

As the instructions captured are accompanied by their referenced memory locations, we can create partitions of memory instructions according to their accessed memory addresses. The purpose of this process is to generate groups of memory accesses which can be performed in parallel or out-of-order with each other. To satisfy this requirement, any two of the generated partitions cannot access the same addresses, or they have to be performing only read operations, making an ordering between them unnecessary. Within each partition, however, the RAW, WAR and WAW ordering must be preserved. This information is used during the subsequent hardware synthesis to reschedule

memory accesses to achieve the best performance. In the final implementation, each partition would be physically associated with a customized cache, described in chapter 4.

The actual process of partitioning memory accesses involves cross-checking the addresses accessed by each instruction. If two instructions do not access common elements in the memory space, they can be separated into two partitions. As illustrated in figure 3.1, each memory accessing instruction is initially placed in its own partition. During the program execution, if two instructions access the same memory location, their partitions are merged. The comparison of accessed memory addresses are performed within an observation window, the sizing of which will be discussed in 3.2.2. As more and more instructions are clustered, we would eventually converge to a stable partitioning for all the instructions under consideration.

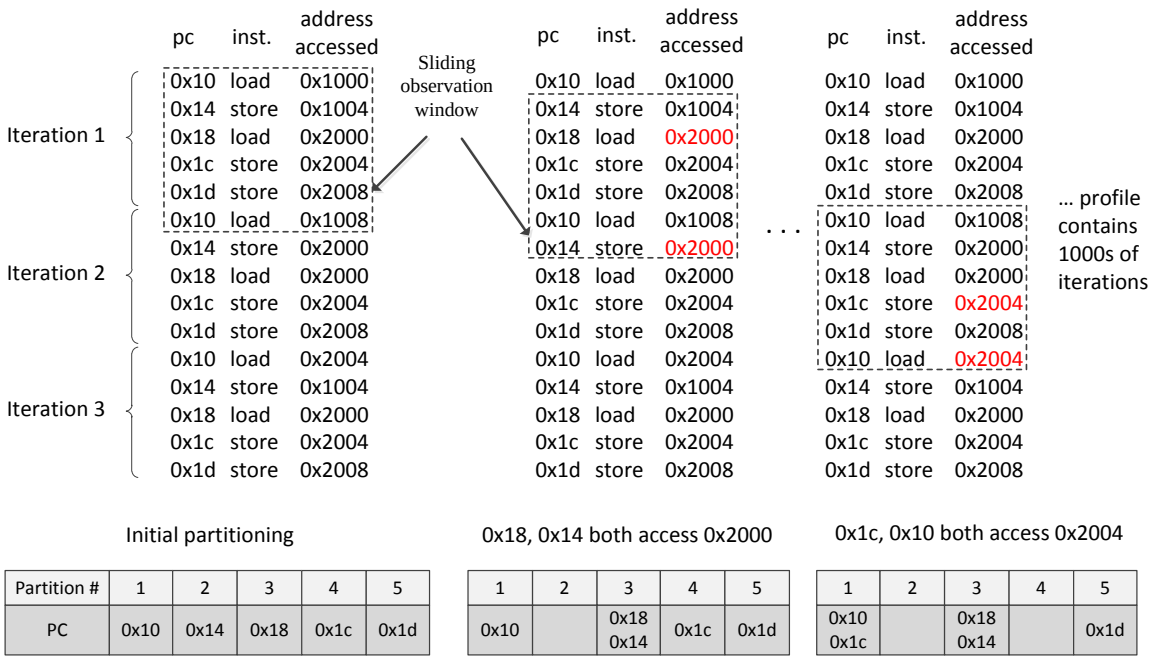

Figure 3.1. The Partitioning Process

3.2.1 Spatial Granularity for Address Comparison

One important parameter for the paritioning process is the spatial granularity with which we check for address clashes, called the comparison frame. It can be as large as the entire address space, or as small as the minimal addressable unit, in our case, a byte. Too large a comparison frame would result in lots of observed clashes and too many memory instructions would be clustered together, removing opportunities for parallelization. In the extreme case, where we use the entire address space as one comparison frame, there would only be one partition left at the end, making the entire process meaningless. Looking at the other end of the spectrum, we can imagine having two instructions writing to different bytes of the same word. They can potentially be placed into two partitions and be subsequently free to be reordered/parallelized. However, considering the

associated physical implementation, partitioning at the byte level would result in extremely small cache lines, wasted bandwidth and extra communication cost. Maximal parallelism is produced if the comparison frame is made as small as the behavior of the individual instructions allows, but it might not translate to best-performing accelerator implementation. Thus to get a good trade-off, we begin by checking address clashes at the word level, and in a later stage, tune the partition sizes, as will be described in 5.1.

3.2.2 Temporal Granularity for Address Comparison

The other factor we have to consider for the partitioning process is the size of the temporal observation window. Accesses by two instructions to the same data, if separated by this window, would not require them to be placed in the same partition. The largest window possible is the entire execution of the program, or in our case, the entire profile we have recorded. However, it would result in a partitioning that is overly conservative. Again, we would have the over-clustering issue as when too large a comparison frame is used. On the other hand, too small a window would produce a partitioning that can possibly result in incorrect execution. This is because the partitioning passed to the high level synthesis tool is assumed to be valid even under rescheduling of instructions. If the observation window is smaller than the potential relative movement of memory instructions, the inter-partition independence (albeit a statistical one) established here would not be compatible with the reordering during HLS. In the most extreme case, if the observation window is only one instruction, every instruction would have its own partition. Using this partitioning, the HLS tool would reorder the memory instructions as if their referenced addresses never overlap, which apparently would leads to WAW, RAW or WAR violations. This scenario is illustrated in figure 3.2.

The objective here is to make the window as small as possible yet ensuring the resulted partitioning does not cause reordering that violates the original program behavior. Since the main optimization techniques in our high level synthesis tool are loop pipelining and intra-iteration instruction parallelization, it is possible to compute a worst-case relative movement of instructions (M_{wc}). As long as our observation window is larger than M_{wc}, the HLS tool is free to use the partitioning as the basis for reordering memory operations.

M_{wc} for Inner Loops: Assuming the most aggressive loop pipelining can be achieved during accelerator generation, we would initiate one iteration per cycle. If we also observe the longest latency of one loop iteration to be L, the maximal number of iterations in flight simultaneously, N, would be $L/1$. Coupled with the most aggressive intra-iteration instruction reordering, the worst case relative instruction movement would be approximately $N * I$, where I is the number of instructions per iteration.

M_{wc} for Outer Loops: For outer loops, pipelining of iterations is not applied during hardware synthesis, so the window just needs to be larger than the number of instructions within one iteration, not counting the inner loops.

With the M_{wc} computed for all inner and outer loops within the region for acceleration, the largest one would be used as the size of our observation window. Currently, at this stage, we do not perform hardware synthesis to determine the exact minimal initiation interval and the exact maximal relative intra-iteration movement of instructions. In fact, these two numbers depends on the result of our partitioning and indirectly on the size of our observation window. Therefore,

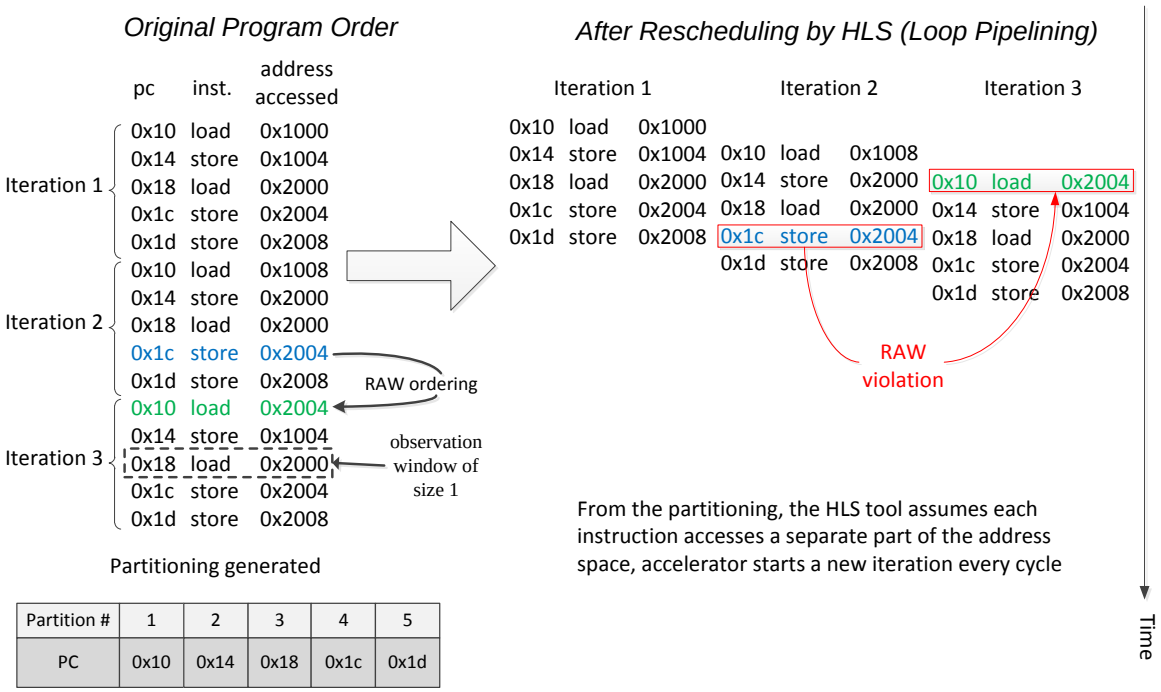

Figure 3.2. Incorrect Scheduling of Instructions Caused by Small Observation Window

the conservative assumption we have made so far, described above, provides the most efficient observation strategy.

Since this partitioning process is profile-based, despite the effort we have put in to ensure its validity, it is still possible that the behavior of the program is different when a different set of input is supplied. To ensure correct program execution, protection mechanisms are needed in case the independence between memory partitions is violated. The implementation details of these mechanisms are described in section 4.2

Chapter 4

Multi-Cache Architecture

Our multi-cache architecture is specifically designed to improve the performance of CPU+accelerator systems implemented on FPGA platforms. In figure 4.1, each accelerator is a hardware version of a computationally intensive segment in the original program. The main difference between this architecture and the conventional one (figure 2.1) lies in the application-specific memory access network. The use of multiple caches is the key technique for exploiting the memory-level parallelism in each application. It is also apparent that the network encompasses multiple accelerators as well as the processor. A built-in coherence mechanism, described in section 4.2, allows the exchange of data between all the caches.

4.1 Application-specific Memory Access Network

The application-specific memory access network is synthesized based on the partitioning of memory accesses, as detailed in section 3. Shown in figure 4.2 is an example network with one accelerator and the processor. Multiple cache ports are exposed to the accelerator, providing large memory bandwidth during execution. The accelerator can reach data in multiple disjoint parts of the address space simultaneously, alleviating the restriction imposed by the conventional memory interface. On the other side, the caches are connected to two shared buses, one for memory requests, the other for memory responses. The internal memory request bus provides a single point where requests from potentially many caches are serialized. It also forwards the requests to a higher-level cache or off-chip RAM in the case that the desired data is not found in this multi-cache network. The internal memory response bus is used to feed the response from the external memory or sibling caches to the requesting cache. If one or more cache misses occur in a single cycle during the execution, the accelerator is stalled. Only when all the caches have obtained the required data, whether from a sibling cache or external memory, the accelerator execution resumes.

Corresponding to the multiple memory partitions associated with each accelerated code segment, there are multiple caches for each accelerator connected to the network. Each of these caches has its own associativity, line size and the index bits, all based on the observed memory access pattern of the instructions in the corresponding partition. As compared to a conventional architecture

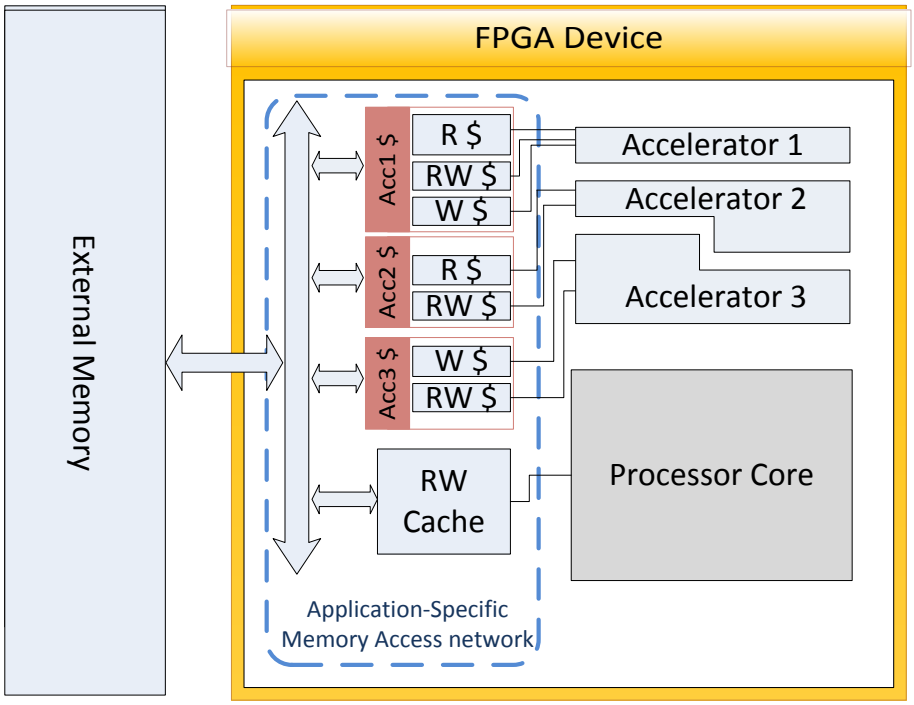

Figure 4.1. CPU+Accelerators System with Multi-cache Architecture

where all the memory accesses go to the same cache, the customization in our system is easier and more effective. The amount of interference between streams of addresses generated by memory instructions is greatly reduced by the partitioning. Thus some very obvious access patterns can be isolated and optimized for in-cache implementation.

Also illustrated in figure 4.2, each accelerator would have a mix of R, W and RW caches. These three cache types reflect the nature of the instructions in each partition. More specifically, for a partition containing either load or store instructions, its associated cache only needs to have either a read or write port. On the other hand, a partiion containing both load and store operations, would require a normal read/write cache. For the write-only and read-write caches, the write policy is write-back, also for write misses, write allocate is used.

4.2 Handling Inter-Partition Memory Dependence

Using the recorded memory addresses to create memory access partitions, our approach builds on the assumption that the captured profile of a program provides an accurate prediction for its future behavior. When the observation data are obtained by running the program for a long time with multiple different input datasets, we can be rather confident how the program would act when a new set of input is given. However, unexpected events, no matter how unlikely, still need to be provisioned for. The implementation of the multi-cache architecture must ensure that their occurrence would only result in a performance degradation rather than an incorrect program outcome.

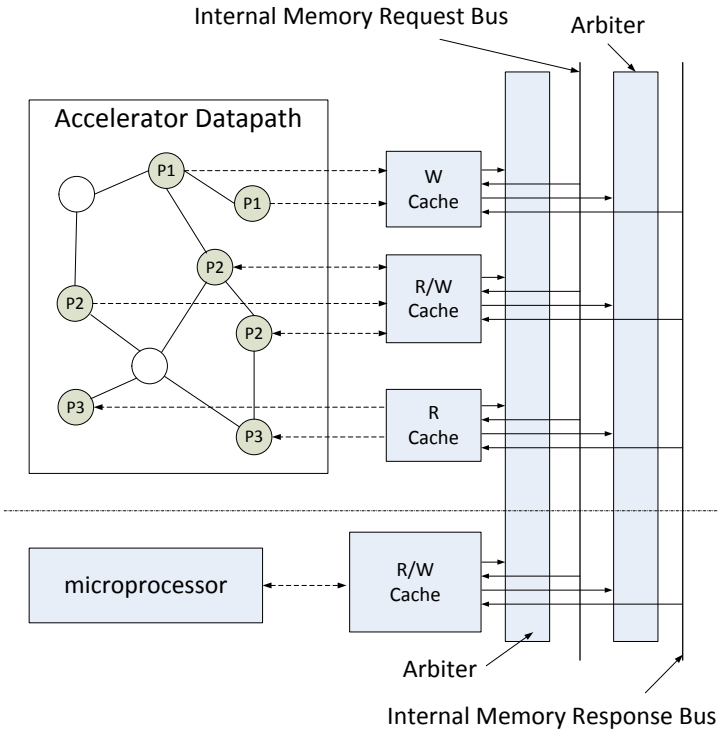

Figure 4.2. Structure of Application-specific Memory Access Network

Before describing what kind of infrastructure is in place to guarantee correct program execution, we look at two different scenarios where the memory partitions created are not actually independent from each other. In the first, it is possible that the accesses by two different memory operators from two partitions to a common piece of data are close temporally, such that the reordering of memory operations in our generated accelerator would have already caused a RAW, WAR or WAW violation. This is shown as scenario I in figure 4.3. As the store at 0x1c is rescheduled to before the load at 0x18 from the same iteration, a WAR violation occurs when they access the same memory address. In the other scenario, the second memory operation from a different partition hit the common data long after the first did. This conflict does not invalidate the reordering we have performed in the accelerator generation, and in fact is not unexpected given how we limited the size of our observation window during the partitioning process. This is scenario II in figure 4.3. The store at 0x14 accesses the same address as the store at 0x1c from the previous iteration. However, the access happens after the completion of the first iteration and thus no WAW violation is resulted. These two scenarios are handled by different schemes, which will be described in section 4.2.2 and 4.2.3.

4.2.1 Vulnerability Window

To distinguish the two scenarios described, we introduce a concept called vulnerability window for the reordering of memory partitions. It captures how far a memory access is rescheduled with respect to its predecessors in the original program order. The size of this window, as opposed to the observation window we discussed in section 3.2.2, is determined by the actual hardware synthesized,

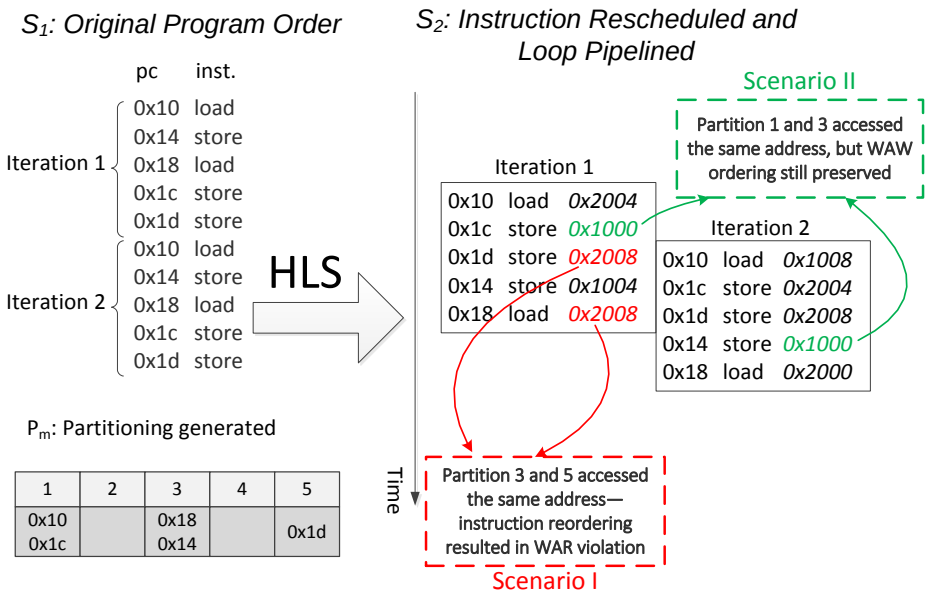

Figure 4.3. Two Scenarios for Inter-partition Dependence

which encodes the scheduling of memory operations. Therefore, the information about the worst case instruction movement the synthesis has introduced is available. Given the original program order S_1 and the rescheduled instruction order S_2, we can determine the vulnerability window for each partition as follows.

1. in each partition P_m, we find the set of instructions A_m, each of which has been rescheduled in S_2 to before its preceding memory instructions in S_1.
2. for each instruction I_i in A_m:
   - find the set of instructions preceding it in S_1. Among these instructions, pick the one instruction I_l which comes the latest in S_2.
   - find the minimal size of the window covering I_i and I_l in S_2, move to the next instruction in A_m.
3. the largest window obtained in the previous step is used as the vulnerability window for P_m

As exemplified in figure 4.4, in the final schedule S_2, the instruction at pc 0x1c has been scheduled to precede instruction at pc 0x18 in the generated datapath, thus its vulnerability window would be from A to C. The other instruction in partition 1, 0x10 is rescheduled to before 0x18 of the previous iteration, and it should therefore have the window covering B to C. The larger window of the two (A to C) is chosen for the partition. In a similar fashion, the vulnerability window for partition 5 is from B to C. As execution progresses, these windows slide forward, with the statically determined size remaining constant. On the other hand, instructions in partition 3 do not have any of their predecessors in S_1 rescheduled to after them in S_2, thus this partition does not have a vulnerability window.

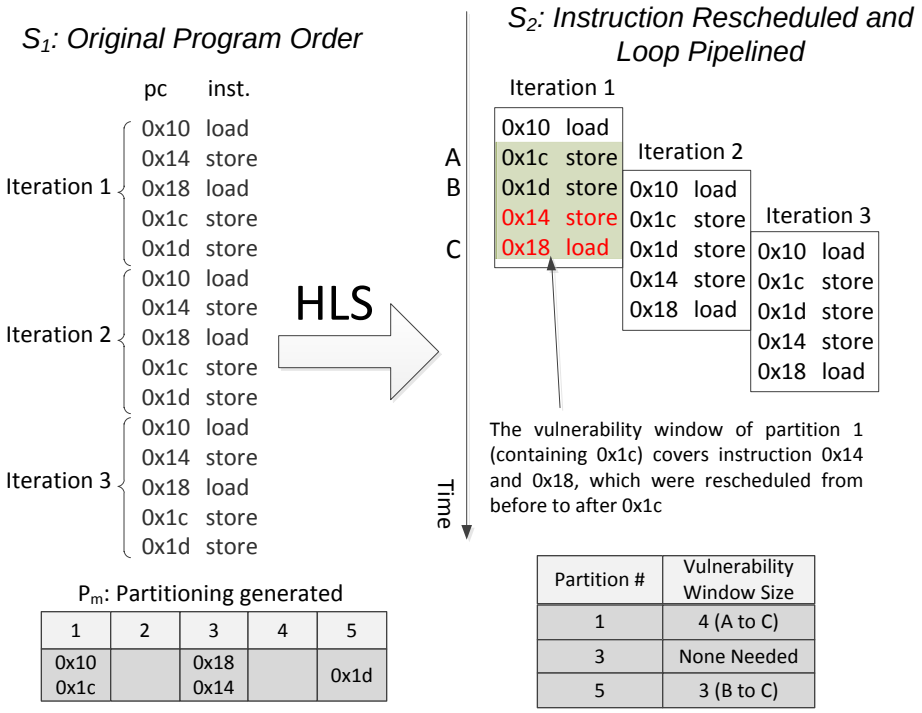

Figure 4.4. Vulnerability Window for Memory Operations

If a different partition's access to the same data falls outside of this window, the reordering of memory operations in the accelerator has not caused any errors. This scenario is called the violation outside the vulnerability window (VOVW). It requires a cache coherence scheme to move the data item such that the newest access would find it in its partition's corresponding cache. On the other hand, if the second memory access falls within the vulnerability window, the reordered memory operations would have already resulted in a wrong execution. In the example shown in figure 4.4, if within the same iteration, the load operation at pc 0x14 is targeting the same address as the store at pc 0x1c, the final schedule S_2 would produce a different result than the original program S_1. To correct the error resulted, we handle this violation in the vulnerability window (VIVW) with an exception scheme. The processor would take over the execution after the violation is detected. This is a high cost error, but should be very rare. Two instructions using memory for communication in a short temporal distance should have already been observed, given that the generated hardware only captures the same sequence of basic blocks executed in the original profile. The partitioning process, which always has an observation window larger than the vulnerability window, would have assigned the communicating instructions to the same partition. Nevertheless, we want to ensure that should this type of error occur, we are still safe from wrong program outcome. Mechanisms for periodic commits and restoration are built into the accelerators and the memory network, such that the processor can start execution from checkpoints.

4.2.2 Cache Coherence Scheme

The cache coherence scheme required by VOVW, described in section 4.2.1, is built on top of the two internal memory buses. This bus snooping protocol ensures only one valid copy of data is in the cache network unless all the owner partitions contain only load instructions. Different caches would behave differently when a miss is placed onto the internal memory request bus by a sibling cache, as detailed in table 4.1.

Local Cache Type	Request Source Cache Type	Local Cache Action
Read-only	Read-only	respond with data[1], keep the local copy valid
Read-only	Write-only Read-Write	respond with data, invalidate local copy
Write-only Read-Write	Read-only	respond with data, invalidate local copy
Write-only Read-Write	Write-only Read-Write	respond with data, invalidate local copy

Table 4.1. Actions of Caches Possessing the Requested Data

[1] when multiple caches respond, the arbiter would choose one

With these rules, it is guaranteed that no caches can share a valid copy of data with a write-only or read-write cache. It is possible, however, that when a cache miss is placed onto the internal memory request bus, multiple read-only caches have the requested data and respond simultaneously. An arbiter is implemented to control the muxing of all the responses to the internal memory response bus, which fans out to all caches.

Although the coherence scheme takes care of VOVW, it is apparent that if the violation occurs too often, the performance would be negatively impacted. It thus becomes necessary to employ another process, partition tuning, described in section 5.1, to adjust the partitioning under these circumstances.

4.2.3 Exception Scheme for Violation Inside Vulnerability Window

Detection of VIVW

The exclusive ownership of data by the caches, as imposed by the cache coherence scheme, guarantees that any cache hit would not result in any data inconsistencies in the system. Consequently, the detection of VIVW would just involve comparing the cache misses on the request bus with each caches most recent memory accesses. Since all the cache misses are serialized on the internal memory request bus, and their addresses are broadcasted to all the other caches, the only addition for the detection mechanism is a local store of a few past memory locations. Physically, this access history buffer is implemented by adding a shift register to every cache. It records the address of each access, as well as the original value that is overwritten by each write access. Accompanying each entry in this buffer, a comparator monitors the request bus and signals a VIVW when there is

a match of addresses. The structure is shown in figure 4.5. To avoid visual clutter, the enable and valid signals are omitted in the diagram.

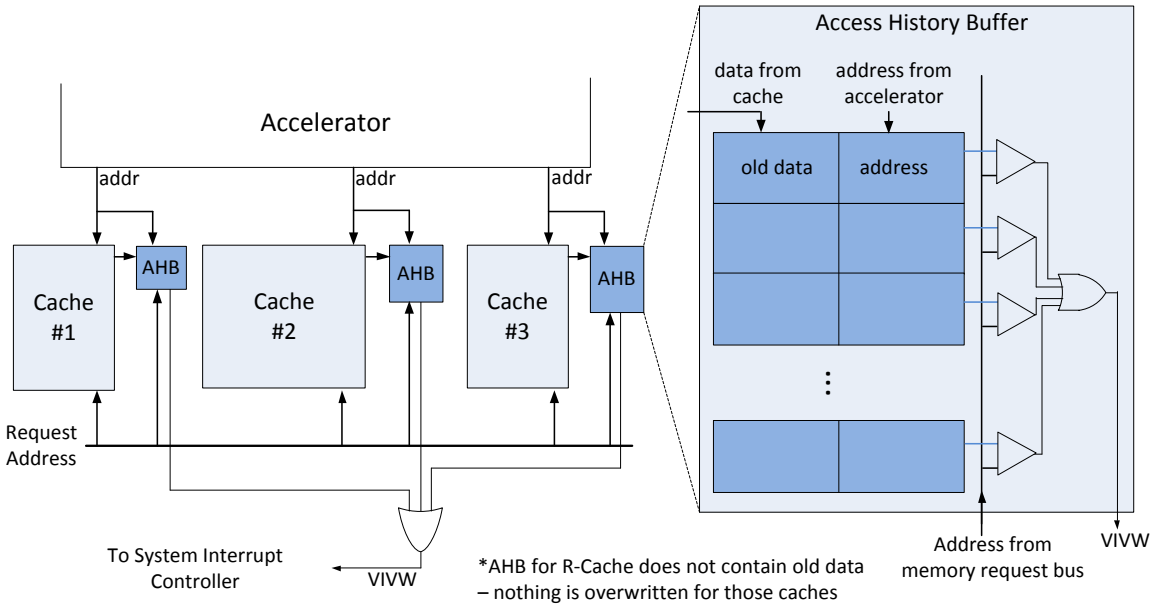

Figure 4.5. Access History Buffer

Each of the buffers, updated by every cache access, has a number of entries equal to the size of the particular partition′s vulnerability window. In reality, the generated schedule in the accelerator is usually wide and short as the instructions are heavily parallelized. As a result, the size of the vulnerability window for each partition is rather small. The access history buffer can therefore be implemented with a reasonable hardware cost.

Checkpointing and Restoration

After detecting VIVW, it is necessary for the processor to take over execution from a point with a known good machine state. More specifically, the memory and registers should be the same as in the case when the entire program was executed in the processor. To achieve this, we have devised a checkpointing mechanism such that memory writes and register value changes are committed only when they are known to be valid. For the innermost loop, one commit is made after each iteration finishes. For the outer loops, a commit is made when the execution of an inner loop starts, or when the current loop exits.

For the register values to be committed, in the generated accelerators, explicit store operations are inserted. Whenever a loop is entered/exited, or a new iteration of the innermost loop is started, the values corresponding to the original processor registers are written into a set of special caches, addressable by the processor. This addition has very little effect on the overall throughput of the synthesized accelerator, which is predominantly determined by the minimum initiation interval of the pipelined loops. Since nothing depends on these new store operations, the iterations of the loops can still fire at the same rate. Also, these special caches are small and not directly connected to the

application-specific memory access network. Thus they have little effect on the operating frequency of our system.

Meanwhile, to ensure correctness of the memory state, two issues must be resolved. First, the memory writes since the last checkpoint should be undone when the exception occurs, and second, the evicted cache lines should not be dumped to the main memory until the next commit occurs. To satisfy the first requirement, we make use of the stored data in the access history buffer. When a VIVW is detected, these old values can be used to undo memory writes. A commit would involve discarding the oldest entries in the access history buffers, which is automatically done as the shift registers take in newer entries. To satisfy the second requirement, the evicted cache lines would need to go into a victim buffer instead of the main memory. The victim buffer delays the commit of these cache lines so they do not pollute the memory.

Given the described infrastructure, when a VIVW exception occurs, the following steps would be performed for the processor to continue execution from a good machine state.

1. for every cache, a small hardware engine fetches the entries in the victim buffer and writes them back to the cache.
2. for each access history buffer, traverse the entries, from the newest to the oldest, write the stored data value to the internal memory request bus, invalidating any cachelines
3. committed register values are copied into the processor
4. processor continues execution

In essence, we have devised a distributed version of the buffers present in out-of-order processors to recover from mis-speculation about memory independence. The size of these buffers are typically small ($\sim$5 entries) since the aggressive parallelization has shorten the latency of each loop iteration drastically.

Chapter 5

Optimizations

5.1 Partition Tuning

As described in section 4.2, we have implemented mechanisms to handle interpartition memory dependencies. Although the correctness of execution is guaranteed, these mechanisms, when exercized, would introduce performance degradation. Therefore it is sometimes beneficial to merge two memory partitions at the cost of reducing utilizable memory-level parallelism. Besides, as the size of a typical cache line is greater than a word, the small spatial granularity we have used in the partitioning process might have also caused an overall performance degradation. To ensure we have a better trade-off in taking advantage of memory-level parallelism and reducing cache misses, a partition tuning step is devised. The exact steps are as follows:

1. execute the program using the partition generated, monitoring each cache's coherence miss rate and the number of misses served by each of its sibling caches.
2. when a cache C_m's coherence miss rate goes above 0.1, list the top five sibling caches who served most of the coherence misses.
3. traverse the list generated in the previous step, pick one sibling cache at a time, merge its partition with that of C_m, and regenerate the accelerator. The new schedule in the accelerator can be used to calculate the loss in performance of the datapath. If this loss is less than the cost of the eliminated coherence misses, the merge is committed.
4. go back to step 1) unless we have converged to a stable partitioning.

This tuning process is based on the existing profile and thus its effectiveness is limited to the accuracy of the profile. In our benchmarking experiment, we did discover candidate partitions for merging using this process. The final result is reported in chapter 7.

5.2 Cost-Performance Trade-offs

In our study, the partitioning process has been rather aggressive. All partitionable memory accesses are partitioned, without giving much consideration to the actual hardware cost. As the cache coherence is bus-based, a large number of partitions would inevitably stress the device resources and subsequently affect the operating frequency of the system. Therefore, if we consider the hardware cost, and the secondary effect on the clock frequencies, the partitioning we have produced, even after the tuning process, may not be the optimal. Thus to achieve the best trade-off, for every pairwise separation of memory accesses, the following factors would need to be considered.

1. the effect on the datapath throughput.
    - how does the initiation interval of the pipelined inner loop change?
    - how does the latency of the outer loops change?
2. the effect on the miss rate of the customized cache network
    - how much interference can be removed to facilitate per-cache customization
    - the extra coherence traffic introduced
3. hardware cost
    - number of slots in the buffers for each cache
    - number of memory ports occupied (important for FPGAs since each BRAM can only support dual port)
    - the cost of the internal memory request/response buses
4. the secondary effect on the clock frequencies of the system

It should be apparent this is a large exploration space. The infrastructure for tackling this optimization task is not in the scope of this work, and may be left for future research.

Chapter 6

Accelerator Generation

To evaluate the benefit of our methodology and architecture template, we have developed a tool flow to convert captured instruction traces to hardware implementation. We have employed several common techniques used in high level synthesis, such as loop pipelining, branch predication etc. in our tool flow. Figure 6.1 outlines the steps performed in our hardware synthesis program.

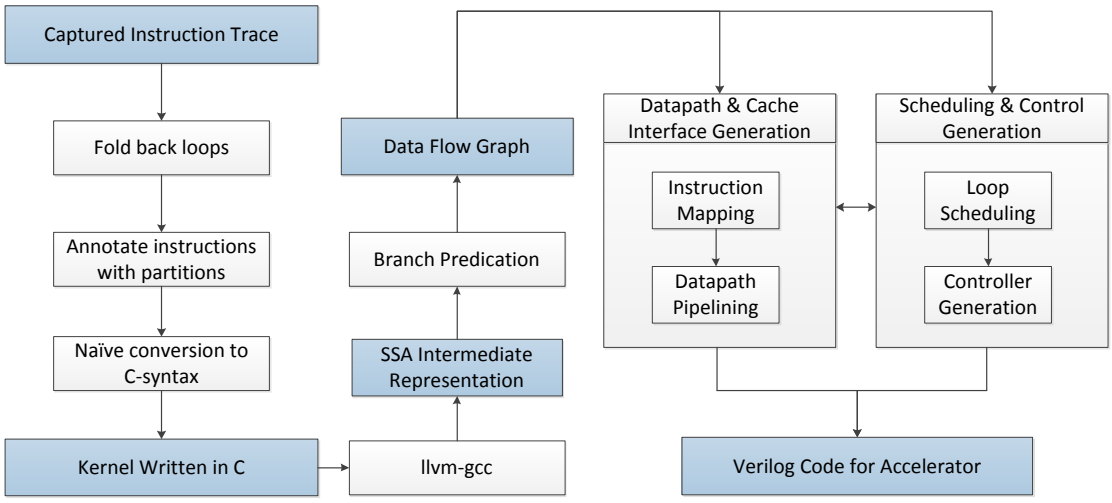

Figure 6.1. Accelerator Synthesis Flow

The instruction trace is essentially a subset of the original assembly code unfolded in time. Many iterations of the loops are captured, with possibly different execution paths in each. The first step in our flow is to fold the loop iterations back into its original form, which is necessary for a resource-efficient accelerator. During this process, we also identify the most frequently executed paths in the program, which would be selected to undergo the transformations into hardware accelerator. This naturally results in exclusion of certain rarely executed paths from hardware acceleration, a technique that was adapted from VLIW compilers [12] and has been used in reconfigurable com-

puting [13]. Then, with the memory access partitions, generated and optimized as described in the previous chapters, we can annotate each of the memory instructions in the assembly code obtained from the trace. The next step is a naive conversion of the code to C syntax. This is performed because we can then leverage the front end compiler (`llvm-gcc`) of the LLVM project [14] to create a single static assignment (SSA) [15] intermediate representation (IR).

The LLVM IR encodes a data control flow graph, where llvm instructions are grouped into basic blocks. We then perform branch predication on this graph in a way similar to partial predicated execution described in [16]. All control transfers and memory operations are also conditioned with the result of the previous branch instruction. Consequently, we have a data flow graph where all the dependencies are between individual instructions and the boundaries between basic blocks are eliminated. Meanwhile, to enforce WAW, WAR and RAW ordering in the same partition, memory dependency edges are added between instructions.

With all the dependencies created, scheduling of instructions can be carried out on the data flow graph, where each node corresponds to a single instruction in the IR. The approach we have taken is similar to that described in [17]. Every instruction in the IR has a delay assigned to it, and the longest cycle in this graph can be calculated accordingly. This cycle, together with the requirement on the sharing of the cache interfaces, determines the initiation interval with which we can pipeline the loop iterations. Meanwhile, to create the datapath, our tool performs a mapping of the instructions to a pre-built library of hardware primitives. Finally, pipeline registers are inserted, and a controller is synthesized. This controller coordinates the hardware execution by activating levels of pipeline registers according to the schedule.

There are some other optimization techniques, e.g. loop unrolling, which can be incoorporated into our hardware synthesis tool. Our implementation here is used to evaluate our methodology in parallelizing memory accesses. As our experimental results show, the memory accesses do become a performance bottleneck given the set of techniques we employed in accelerator generation. With more aggressive approaches, the constraint set by the traditional memory model would only become more severe, making the exploitation of memory-level parallelism more important.

Chapter 7

Experiment Result and Analysis

We implemented our prototype system with a Virtex-5 FPGA (XCV5LX155T-2) on a BEE3 (Berkeley Emulation Engine) platform. Nine applications from Spec2006 and MiBench were run through our flow. Ten hot regions are identified and synthesized into accelerators. As a baseline system, a conventional processor platform was implemented with Xilinx's Microblaze technology. The accelerators using a multi-cache memory access network were compared against their counterpart with a single cache (single memory partition), both of which are implemented on the same device.

7.1 Accelerator Performance

Shown in table 7.1 are the performance numbers of the accelerators measured using our system prototype. T_d refers to the time when the generated accelerators are actually running, T_c refers to the time when the accelerators are stalled for cache misses. The baseline represents the amount of time spent by the processor-only platform in executing all the instructions in the selected region.

From the table, it can be observed that when the multi-cache network is used to service accelerators' memory requests, the time in datapath (T_d) is reduced, demonstrating the benefit of exploiting memory-level parallelism. However, the total time for cache misses (T_c) went up, with two main contributing factors. First, as the same amount of storage is divided up to multiple smaller caches, capacity misses occur more frequently. Customizations such as those mentioned in section 4.1 can help alleviate the negative impact by reducing conflict misses, but there is still an overall increase in the miss rate. Meanwhile, the communication between caches also caused some performance degradation. However, all these costs are outweighed by the benefit of better accelerator throughput, resulting in an overall gain in performance.

Another observable effect is the decrease in the minimal clock period when single cache is replaced with the multi-cache network. This is due to the accelerator internalizing muxing of memory requests when only a single cache interface is available. In the multi-cache network case, the extent of this internal muxing is greatly reduced. Meanwhile, the muxing of multiple cache misses onto the

Benchmark	Accelerator w/ Single Cache						Accelerator w/ Multi-cache Network					
	% of trace.	clock period (ns)	T_d $\times 10^6$ (ns)	T_c $\times 10^6$ (ns)	% time compute	Speedup vs. Baseline	clock period (ns)	T_d $\times 10^6$ (ns)	T_c $\times 10^6$ (ns)	% time comput	Speedup vs. Single $	Overall Speedup
libquantum	99.98	5.1	0.67	0.84	44.45	2.29	5.1	0.33	1.17	22.22	1	2.29
hmmer	92.81	7.608	2.95	0.70	80.90	2.08	7.6	0.49	0.96	33.61	2.52	5.24
h264	92.39	9.444	4.68	0.26	94.83	3.27	7.612	2.44	0.52	82.42	1.67	5.46
susan	98.50	5.691	5.54	0.072	99.87	4.18	5.283	2.60	0.0076	99.71	2.13	8.90
dijkstra1	72.64	7.284	0.76	0.029	96.36	10.85	7.284	0.76	0.029	96.36	1	10.85
dijkstra2	17.14	4.773	0.49	0.14	77.86	3.37	4.773	0.49	0.14	77.86	1	3.37
qsort	87.05	5.647	3.70	1.55	70.49	4.20	5.16	2.67	1.53	63.61	1.25	5.25
jpeg	80.71	4.142	1.45	0.44	76.67	8.72	5.248	0.52	0.52	50.01	1.83	15.92
adpcm	98.77	7.55	3.22	0.091	99.72	5.25	7.545	2.30	0.0091	99.61	1.40	7.34
crc	99.40	6.005	1.93	0.09	99.54	7.70	5.211	0.84	0.0079	99.06	2.29	17.65

Table 7.1. Performance Improvement Breakdown

internal memory request bus happens on the other side of the cache. Therefore it can be decoupled from the clock used inside the accelerator, giving better performance as cache hits occur most of the time.

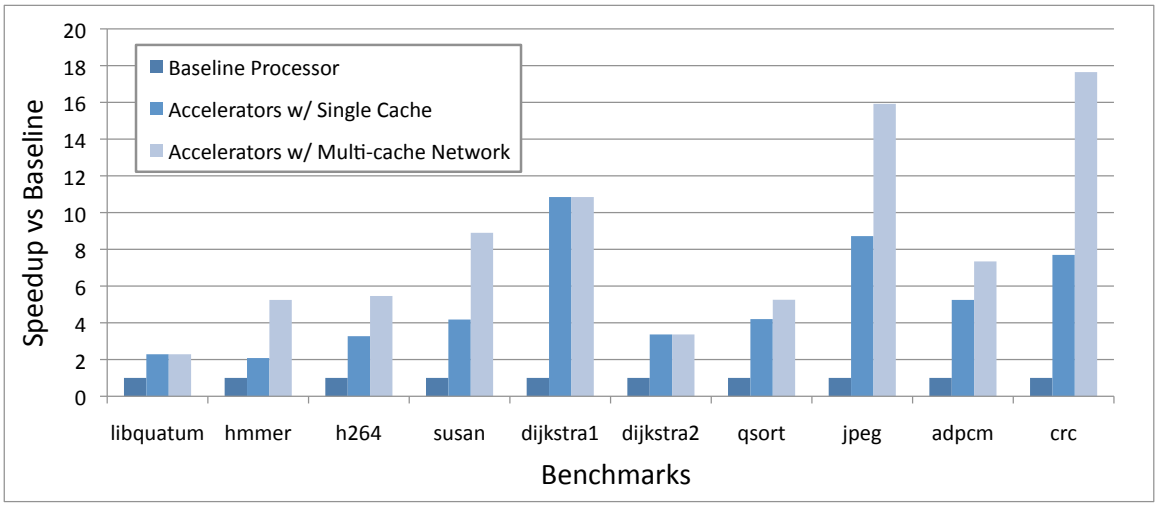

Figure 7.1. Performance Comparison between Different Implementations

Some of the benchmarks did not demonstrate any performance benefit when our approach was applied. Libquantum showed significant improvement in T_d when memory partitions were applied. However, that was accompanied by a big increase in T_c, which cancelled out the benefit. The accelerated code has two accesses in the inner loop which were placed into different partitions. This parallelism resulted in the reduction in T_d. However, in each iteration, the addresses the two instructions accessed were always one word away from each other. Also, there were no repeated accesses to the same element. Consequently, we cannot use a big cache line and every access produced a cache miss. The net effect is a draw in performance between the two implementations. For dijkstra, the first accelerator had multiple partitions that were eventually combined to one during

partition tuning, and the second accelerator has only one memory access. As a result, in both cases, the final implementations and performance were the same, with or without applying our new approach.

On average, we observed performance improvement of 4.5x over the baseline implementation when single cache accelerators are used. When memory level parallelism is identified and exploited for each of the accelerators, the overall performance improves another 51%, to 6.9x of the baseline implementation. Figure 7.1 shows the graphical comparison of each implementation over the entire benchmark set.

7.2 Resource Consumption

We have normalized the amount of memory used in the caches of the various implementations to be equivalent to a 64KB direct mapped cache—a typical cache implementation used by Microblaze. The actual silicon area consumed for storing data is thus kept roughly constant across the design points. However, as the multi-cache memory network provides many more memory ports for the accelerator datapath, the BRAM usage is in general much greater than that of a single cache. Besides, the cache coherence scheme and the exception mechanism for memory access collisions also took up a significant area.

Benchmark	**Accelerator w/ Single Memory**			**Accelerator w/ Multi-cache**			
	LUTs	FFs	BRAMs	Partitions	LUTs	FFs	BRAMs
libquantum	2623	2598	15	3	5016	3863	18
hmmer	8161	9873	15	16	19646	11281	40
h264	15534	45062	15	13	25261	37595	35
susan	7736	24486	15	6	12373	35222	32
dijkstra1	3308	6382	15	1	3308	6382	15
dijkstra2	1260	809	15	1	1260	809	15
qsort	4503	12933	15	6	9109	12136	32
jpeg	3111	4505	15	4	6177	5626	23
adpcm	4629	9169	15	5	8700	9055	27
crc	2807	1978	15	6	7480	4417	32

Table 7.2. Resource Consumption of Accelerators.

As shown in table 7.2, the ratio in number of BRAMs in the two implementations can be as high as 2.7X. Meanwhile, the LUT count increase in the multi-cache network is most apparent in the smaller benchmarks with large number of partitions. The FF count presented in the table, on the other hand, showed some of the benchmarks having smaller number when using multi-cache network. This is mainly due to the saving resulted in the accelerator datapath. A smaller initiation interval and a reduction in levels of operators counteracted the effect of the multi-cache network, resulting in a net decrease.

Chapter 8

Conclusion

For many memory-intensive embedded applications, sustaining high peak memory access bandwidth is the key to maximizing their computing performance. In this work, a novel approach is developed to parallelize application memory accesses, using the abundant block RAMs and hardware flexibility of FPGAs. We have shown that the new multi-cache architecture and its complementary tool flow is effective in helping to overcome the performance bottleneck imposed by the traditional accelerator memory model. An improvement of 51% is achieved in experiments using the proposed hardware template and methodology .

This improvement, however, came at the cost of a larger area, partially due to the coarse granularity of the block RAMs on FPGAs. As the total number of memory bits used is normalized, the increase in BRAM usage is caused by the large number of memory ports needed in the new architecture. Our approach would definitely benefit from an FPGA fabric where the block RAMs' bandwidth to capacity ratio is larger. Another issue with our approach is the large number of LUTs involved in implementing the memory access network and the exception mechanism. To alleviate this, a possible extension of our current project can explore the incorporation of static memory alias analysis in the generation of multi-cache network. By eliminating some of the communication infrastructure between subsets of caches, the area overhead in the new architecture can be reduced.

There are several other possible directions for further exploration. The cost and benefit of our approach may change when a different network topology is used to connect the caches. Also, with multiported memory or banked cache, we can have different microarchitectures which may have advantages in certain aspects. Furthermore, it would be interesting to look at some benchmarks with highly dynamic memory access patterns, e.g. frequent use of linked list, to have a better understanding of the limitations in our approach.

References

[1] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XI. New York, NY, USA: ACM, 2004, pp. 14–26. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024396

[2] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[4] Altera, "Nios II C2H Compiler User Guide," 2009.

[5] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to fpga circuits," *Computer*, vol. 41, no. 7, pp. 40 –46, 2008.

[6] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and power of cache-based reconfigurable computing," *SIGARCH Comput. Archit. News*, vol. 37, pp. 395–405, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555804

[7] Alter, "SOPC Builder User Guide."

[8] Xilinx, "Embedded System Tools Reference Manual."

[9] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, "Marc: A many-core approach to reconfigurable computing," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, dec. 2010, pp. 7 –12.

[10] "QEMU: a generic and open source machine emulator and virtualizer," http://wiki.qemu.org/Main_Page.

[11] T. Ball and J. Larus, "Efficient path profiling," in *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, Dec. 1996, pp. 46 –57.

[12] J. R. Ellis, "Bulldog: a compiler for vliw architectures (parallel computing, reduced-instruction-set, trace scheduling, scientific)," Ph.D. dissertation, New Haven, CT, USA, 1985, aAI8600982.

[13] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *In Proc. International Workshop on Field Programmable Logic*. Springer-Verlag, 1998, pp. 248–257.

[14] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, October 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[16] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. mei W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *IN PROCEEDINGS OF THE 22TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, 1995, pp. 138–150.

[17] T. J. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '00. New York, NY, USA: ACM, 2000, pp. 57–64. [Online]. Available: http://doi.acm.org/10.1145/354880.354889