Automatic synthesis of parallel memories for VLIW ASIPs

Xu, P.

Award date:
2014

Technische Universiteit Eindhoven University of Technology

Automatic synthesis of parallel memories for VLIW ASIPs

By

Peng Xu

Final Project Thesis
Eindhoven University of Technology
Department of Mathematics and Computer Science

Student:
Peng Xu (0771511)
p.xu@student.tue.nl

Supervisors:
Lech Jozwiak, EE, TU Eindhoven
Rosilde Corvino, EE, TU Eindhoven

10-02-2014

Abstract

Nowadays, people's life is closely related to an extensive use of electronic systems, which have to satisfy increasing demands related to their small size, energy efficiency and high performance. Application-specific instruction-set processors (ASIPs), featured as customized instruction processors, are suitable for the realization of such highly-demanding electronic systems, because they are able to meet stringent demands for size, power consumption and performance.

However, the stringent physical and economic requirements on hardware design combined with the increasing complexity of modern applications, result in the extremely complex problem of ASIP hardware design and application mapping, when guaranteeing a low power consumption and high performance.

The research described in this Master report addresses the automatic synthesis of parallel memories for Very Long Instruction Word (VLIW) ASIPs and data to memory mapping problem for ASIPs. The research activities are applied to the Intel/Silicon Hive VLIW ASIPS. The main aim of this Master project is to propose methods to reduce memory access conflicts and thereafter, propose and implement an automatic memory mapping tool to deal with the data mapping problem. The tool is able to automatically explore and decide an adequate number of memories needed for a specific application, as well as, perform the solution space exploration for the data mapping problem and simulation of the results. With the help of the tool, the hand-designed, error-prone and time consuming mapping method became automatic, and an optimal solution for the data mapping problem can be obtained.

Contents

Abstract 1
Table of Figures 4
1. Chapter 1: Introduction 5
1.1 Project background 5
1.2 ASAM project 6
1.2.1 Project introduction 6
1.3 Intel/Silicon Hive ASIPs 8
1.4 Problem statement 10
1.5 Report organization 12
2. Chapter 2: Automatic synthesis of parallel memories for ASIPs 13
2.1 Introduction 13
2.2 Related work 15
2.3 Proposed solution 15
3. Chapter 3: Data conflict problem 18
3.1 Problem analysis 18
3.2 Reduce data conflicts 19
3.2.1 Self-conflicts 19
3.2.2 Parallel - conflicts 19
3.2.3 API for data transfer 20
3.3 Code optimization 21
3.3.1 Loop fusion 22
3.3.2 Software Pipelining 24
3.4 Influence of data to memory mapping strategy 24
4. Chapter 4: Memory mapping tool design 26
4.1 Introduction 26
4.2 Memory mapping tool construction 27
4.2.1 Input 27
4.2.2 Memory mapping tool 27
4.2.3 Output 29
4.3 Memory reduction 30

4.4 Solution space reduction 30
5. Chapter 5: Experimental results 33
5.1 Data conflicts elimination 33
5.2 Application of the tool to benchmark applications 37
6. Chapter 6: Conclusion 39
6.1 Work and contribution 39
6.2 Ideas for further work 40
Bibliography 41
Appendix 42
Memory mapping algorithm 42
Header file for 3mm 43

Table of Figures

Figure 1-1 ASAM design flow 6
Figure 1-2 ASIP level DSE 7
Figure 1-3 ASIP architecture template 9
Figure 1-4 ASIP instance used in the project 10
Figure 1-5 Schematic of issue slot a) issue slot 1 b) issue slot 2 and 3 10
Figure 2-1 System performance variances due to different mapping strategies 14
Figure 2-2 Arrays assignments of mapping strategies 1 and 5 14
Figure 2-3 Memory accesses and array size of mapping strategies 1 and 5 14
Figure 2-4 Memory performance normalized with memory 3 15
Figure 2-5 3mm a) conflict graph b) colored conflict graph 17
Figure 3-1 Partial code segment of 3mm 18
Figure 3-2 Partial code segment of 3mm after applying register files to reduce self-conflicts 20
Figure 3-3 Assign arrays in application 3mm to reduce parallel-conflicts 20
Figure 3-4 Original code segment of 3mm 22
Figure 3-5 Improvement of data locality for array E in 3mm a) original b) after loop fusion 23
Figure 3-6 Reduction of loop overhead for partial code segment of 3mm 23
Figure 3-7 Subset of data mapping strategies for 3mm 25
Figure 4-1 System flow of memory mapping tool 27
Figure 4-2 Sub-parts of the memory mapping tool 28
Figure 4-3 Process flow of memory mapping algorithm 29
Figure 4-4 3mm a) Partial output of memory mapping tool b) Colored conflict graph according to mapping strategy 1213321 29
Figure 4-5 conflict graph of 3mm a) original b) reduced 30
Figure 4-6 Performance comparison between the original conflict graph and the reduced conflict graph 31
Figure 4-7 Grouped conflict graph a) 3mm b) 2mm 32
Figure 4-8 Performance comparison between the original simulation method and the grouped simulation method 32
Figure 5-1 Data mapping strategies for 3mm, 2mm and atax to eliminate parallel-conflicts 33
Figure 5-2 3mm a) Execution time b) Energy consumption 34
Figure 5-3 2mm a) Execution time b) Energy consumption 34
Figure 5-4 atax a) Execution time b) Energy consumption 35
Figure 5-5 Memory access for applications after each optimization step (normalized value) 35
Figure 5-6 Resource utilization for 3mm 36
Figure 5-7 Issue slot utilization of application 3mm, 2mm and atax 37
Figure 5-8 Improvements of code optimization after data conflicts are eliminated 37
Figure 5-9 Performance comparison between optimal mapping strategy and worst case mapping strategy a) Execution time b) Energy consumption. 38
Figure 5-11 Memory performance normalized with memory 3 38

1. Chapter 1: Introduction

This chapter introduces the project and presents relevant background information. It starts with background of the reported research activities. Next, a brief introduction to the Intel/Silicon Hive ASIP technology is given, consisting of discussion of general ASIP architecture template and the ASIP instance used in this research activity. Finally, the main problem addressed during this research activity and the organization of this report are described.

1.1 Project background

Nowadays, people's daily life is closely related to an extensive use of electronic systems, which have to satisfy increasing demands related to their small size, energy efficiency and high performance. Application-specific instruction-set processors (ASIPs), featured as customized instruction processors for specific applications, are suitable for such electronic systems, because they are able to meet the stringent demands for size, power consumption and performance.

ASIPs are an intermediary solution between the Application-specific integrated circuits (ASIC) and general purpose (GP) processors, having both the flexibility of the general purpose processor and the high performance of ASICs. ASICs are able to efficiently realize any kind of functionality, but they are extremely costly and not programmable. ASIPs deliver greater computational efficiencies than GP processors and more flexibility than fixed-function logic designs. As such, they are an appropriate technology to consider for performance and power sensitive designs in next-generation SoCs – particularly where flexibility provides a competitive advantage [1].

However, the stringent physical and economic requirements on hardware design combined with the increasing complexity of modern applications, result in an extremely complex problem of ASIP hardware design and application mapping when guaranteeing a low power consumption and high performance. The ASIP design problem involves application analysis, code optimization, scheduling and mapping, as well as, efficient processor design with customized data paths and memory subsystem. The design of memory subsystem, which involves the choice of the appropriate number and size of parallel local memories and of the mapping of arrays into the local memories, influences the performance of the processor to a high degree. This influence is even increased for data dominated applications, i.e. applications processing large amount of data. A good strategy for data memory mapping should make the full use of the allocated memory and should optimize the system performances. The problem of finding a good data to

memory mapping strategy and deciding a minimum number of different memory locations where to map the data, so that all possible conflicts in data access are removed, can be referred to as the data mapping problem. The subject of the Master project reported here is the automatic synthesis of parallel memories for VLIW ASIPs and data to memory mapping problem for ASIPs. The project was performed as a part of the European research project ASAM.

In the rest of this chapter, we describe the context of the Master thesis, i.e. the ASAM project, and SH technology and give the problem statement.

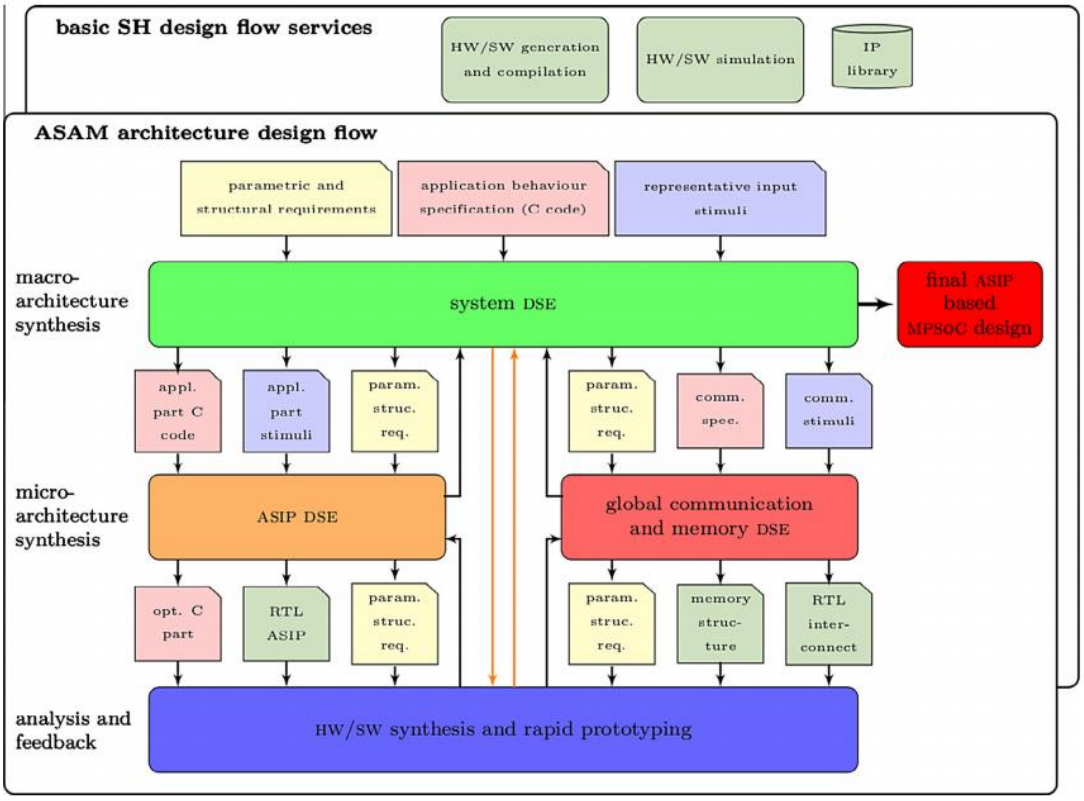

Figure 1-1 ASAM design flow

1.2 ASAM project

1.2.1 Project introduction

ASAM stands for Automatic Architecture Synthesis and Application Mapping. It is a European research project conducted in the framework of the ARTEMIS Program [2]. ASAM targets a uniform process of automatic architecture synthesis and application mapping for heterogeneous multi-processor embedded systems based on adaptable and extendable VLIW ASIPs. It aims to provide a tool suite for automatic multi-ASIP system design. The new design environment allows a rapid exploration of the high-level

algorithm and architecture design spaces, as well as, an efficient automation of the final system synthesis, and in consequence, a quick development of high-quality designs [3].

System DSE takes as its inputs: an application C-code, parametric and structural requirements, and representative stimuli. ASIP DSE aims at the design of a single ASIP and its associated software for the execution (of a part of) the whole application. GC&M DSE aims at the exploration and optimization of the global communication and memory structures for a multi-ASIP system. HW/SW synthesis accepts as input service requests from the System DSE, ASIP DSE and GC&M DSE. It also takes as input the abstract architecture description of the designed MPSoCs or their parts, and the corresponding restructured application C-code [3]. It produces a corresponding actual hardware and software. Rapid prototyping performs a simulation on FPGA evaluation of the synthesized HW/SW design.

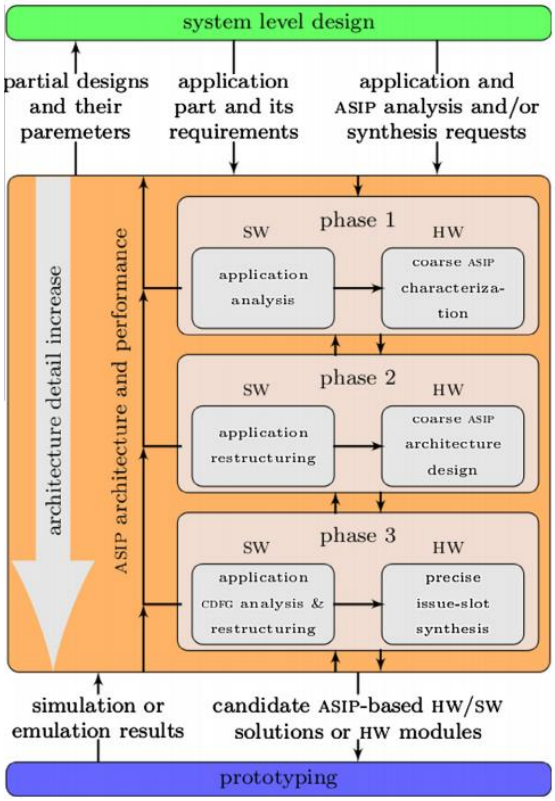

Figure 1-2 ASIP level DSE

This Master project is related to ASIP DSE. The ASIP level DSE in the ASAM project is subdivided into the following three phases, as shown in Figure 1-2:

- Application analysis
- Application optimization and ASIP coarse design

- Refinement

This Master project is part of the refinement phase. It explores and proposes a solution to the problem of parallel memory design and data to memory mapping for ASIPs to optimize system performances. ASAM project targets the VLIW ASIP technology of Intel Benelux (former Silicon Hive).

1.3 Intel/Silicon Hive ASIPs

Intel/Silicon Hive (SH) is a company of Intel's Mobile Communications Group (MCG), providing generic customizable ASIPs, developing flexible system intellectual properties (IP) modules based on ASIPs, as well as, for handling media processing in consumer electronics and mobile terminals chipsets, designing application-specific solutions for image, video processing and communications. SH provides a hive processor description language and a hive system description language. TIM language, the processor specification language, is a basis for SH's programming and processor generation tools. HSD language, which is the system description language, can be used to write customized system descriptions and plug-in customized device into the system. Moreover, a software development kit (SDK) is provided, including toolsets and application libraries to allow users to create fully programmable systems on chips that can be adapted to different application fields.

SH ASIP consists of one or more interconnected *Cells*. A single cell defines a Very Long Instruction Word (VLIW) machine that is capable of executing parallel software with a single thread of control. A cell template is shown in Figure 1-3, taken from [4].

A cell consists of a *Core* that performs computations under software program control, including:

- Datapath, which contains several *function units,* organized in a number of parallel *issue slots* to realize functional operations.
- Sequencer, which is a simple state machine containing a program counter register as well as a status register, can enable special processor modes under software control

Another part is a *CoreIO* that provides subsystems of memories and I/O allowing the core to be easily integrated in any system. *CoreIO* contains:

- Local data memories, comprised of one or more storage or I/O devices, allowing the function units' access to local physical memory in *CoreIO* or to perform memory mapped I/O with the system.

- Interfaces, providing communication and easy integration of a processor in a wide variety of system architectures
- FIFO, used with a stream interface to deal with a data stream.

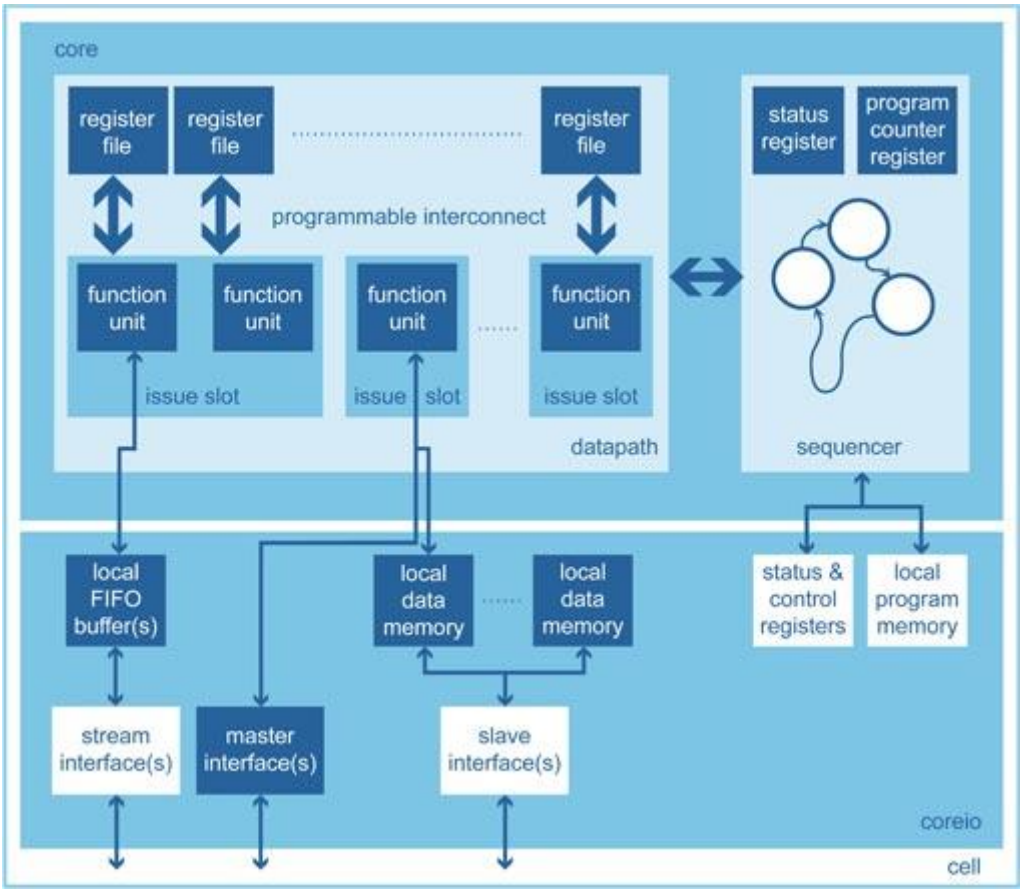

Figure 1-3 ASIP architecture template

The processors used in the scope of this project are instances of the ASIP architecture template of Figure 1-4. Every issue slot has an associated register file. The register files are part of storage elements in the VLIW architecture. The read ports of a register file are connected with relevant Function Units (FUs) in the issue slot the register file is deployed in. For the write ports, they are fully connected with each other through buses. Memories are single ported, which cannot be accessed concurrently. For the purpose of this project, we considered that maximally one memory is connected with one issue slot.

FUs are used for different operations. Issue slot 1 consists of 9 FUs, as shown in Figure 1-5a. A BRanch Unit (bru) and a Status Update Unit (suu) form the sequencer, which are used for program control. ARithmetic Unit (aru), Bit Management Unit (bmu), LoGic Unit (lgu), Multi Accumulate (mac), PaSs Unit (psu) and SHift Unit (shu) are able to provide the instructions for general purpose processing. Load Store Unit (lsu) is used for data transmission. Issue slot 2 and issue slot 3, as shown in Figure 1-4b, have the same

structure, consisting 6 FU's: ARithmetic Unit (aru), Bit Management Unit (bmu), LoGic Unit (lgu), Multi Accumulate (mac), PaSs Unit (psu) and SHift Unit (shu).

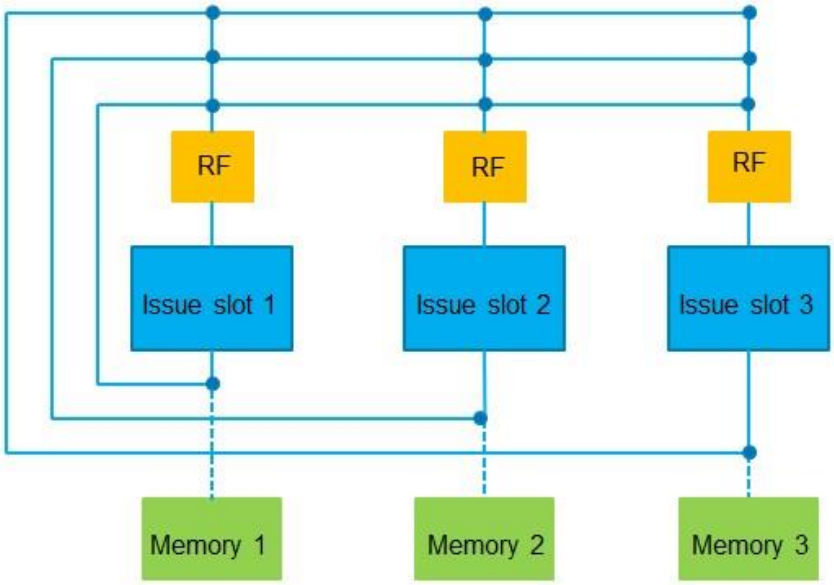

Figure 1-4 ASIP instance used in the project

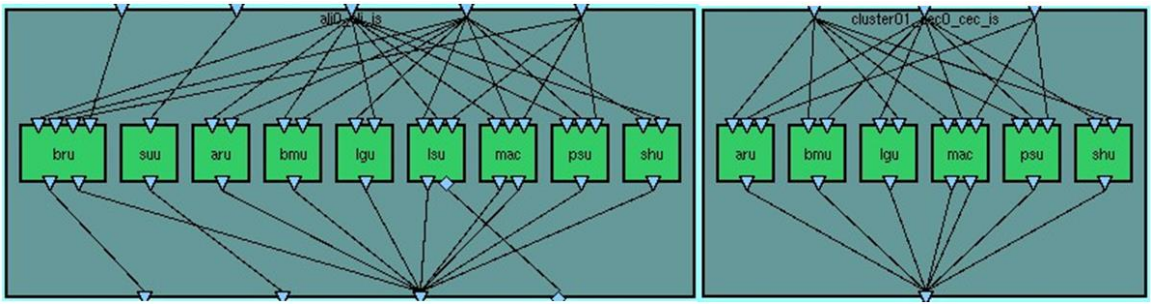

Figure 1-5 Schematic of issue slot a) issue slot 1 b) issue slot 2 and 3

1.4 Problem statement

Memory subsystem design and data to memory mapping are main concerns in ASIP design, especially for data-intensive applications. To realize these applications with high performance and high energy efficiency in a small size ASIP, the following two main problems, being the subject of this Master project have to be solved:

- **Elimination of data conflicts when accessing memories.** Conflicts in data access appear, if different parts of same data array or different data arrays mapped to the same memory are accessed simultaneously. To solve these conflicts, data accesses are rescheduled and some of them are delayed in time. This introduces stall cycles in the execution of processes waiting to be fetched with data, and it

degenerates the overall system performance. Adequate methods need to be found to reduce the data conflicts. There are two kinds of conflicts when accessing data into memory: self conflicts and parallel conflicts. Self conflicts appear when there are simultaneously accesses to the same array variable. Parallel conflicts appear when there are simultaneously accesses to different array variables mapped into the same memory.

- **Deciding the proper number of parallel memories and appropriate data mapping to memories.** Data mapping involves exploration of various possible placements of data in memories. This is an error-prone and time consuming process, requiring a lot of iterative analysis passes and a long experimental time, due to large solution space simulation. Automation of this process is therefore very important.

The research reported in this Master report addresses the automatic synthesis of parallel memories for VLIW ASIPs and data to memory mapping problem for ASIPs, focusing on the problems mentioned above, the aims of the project are the followings:

- To analyze the problem of parallel memory synthesis and data mapping for VLIW ASIPs, when focusing on the above listed two main sub problems of this problem.
- To propose and implement a method to reduce the memory access conflicts.
- To develop an automatic memory mapping tool to deal with the data mapping problem. The tool has to automatically find an appropriate number of memories needed for a specific application, as well as, an optimal data mapping strategy.
- To perform experimental research to analyze how the data conflicts elimination method improves the system performance and to analyze the efficiency of the automatic memory mapping tool.

The research activity builds upon these aims and is developed as described below.

Applications used in this project are ported onto an ASIP instance and their performance is evaluated. Then methods are proposed and implemented to deal with self-conflicts and parallel-conflicts. Thereafter, certain code optimizations are performed to obtain the power efficiency and high performance. Subsequently, an automatic memory mapping tool is designed and implemented to find the minimum number of local memories required for a specific application and an optimal data mapping strategy, which improves both the performance and power consumption. Finally, a grouping method is proposed and implemented to reduce the data memory mapping solution space, and in this way, to speed up the exploration time. In parallel to the above listed activities and after finalizing implementations of particular methods, an extensive

experimental research has been performed which is discussed in the corresponding sections of the report.

1.5 Report organization

The reminder of this thesis is structured as follows.

In Chapter 2, the introduction for automatic synthesis of parallel memories for ASIPs is given. Then, a motivational example is used to explore the influences of different data to memory mapping strategies on system performance. Thereafter, related works are summarized and one method to deal with the data to memory mapping problem is proposed.

In Chapter 3, the data conflict problems of data mapping onto local memories are analyzed, and strategies are proposed to reduce the data conflicts. Moreover, some code optimizations are applied to make better use of the VLIW architecture in order to improve the system performance.

In Chapter 4, a memory mapping tool proposed by us and its implementation are discussed. At first, the motivation to the design of this tool and its general design are given. Then, the implementation details of the memory mapping tool and also the algorithm used for the tool are presented. Thereafter, the memory mapping tool is applied to test benches to explore how the tool benefits the selection of a mapping strategy. Finally, the bottleneck of the tool is discussed, and a grouping method is proposed to reduce the solution space of data mapping to overcome the bottleneck.

In Chapter 5, the experiments results for data conflicts elimination and memory mapping tool are described. First part describes how the system performance improved by eliminating data conflicts and then, the efficiency of the memory mapping tool is illustrated.

In Chapter 6, this report is concluded providing a summary and discussion of the presented work and a discussion. Also, several ideas for future research and development are discussed.

2. Chapter 2: Automatic synthesis of parallel memories for ASIPs

In this chapter, the introduction for automatic synthesis of parallel memories for ASIPs is given. Then, a motivational example is used to explore the influences of different data to memory mapping strategies on system performance. Thereafter, related works are summarized and one method to deal with the data to memory mapping problem is proposed.

2.1 Introduction

In order to realize data-intensive applications with high performance and high energy efficiency on a small sized ASIP, the following two main problems have to be solved: 1) Elimination of data conflicts when accessing memories, since data conflicts introduces stall cycles in the execution of the overall process. 2) Selection of the appropriate number of parallel memories and appropriate data to memory mapping.

We will use the 3mm application for benchmark to illustrate our discussion. The 3mm is 3 matrixes multiplication distributed in three independent loops, each realizing one of the following matrix multiplications: E = A*B; F = C*D; G = E*F.We explain the impact of memory mapping on system performance by using the simplified models of application execution time and energy consumption taken from [5] [6] and reported below:

Estimated execution time calculation formula:

- Execution time

$$\propto \sum access(memory\_i) + \text{access}(\text{pmem})$$
$$\propto \text{access}(\text{memory}_1) + \text{access}(\text{memory}_2) + \text{access}(\text{memory}_3) + \text{access}(\text{pmem})$$

Estimated energy consumption calculation formula:

- Energy consumption

$$\propto \sum access(memory_i) * size(memory_i) * \varepsilon + \text{access}(\text{pmem}) * \text{size}(\text{pmem}) * \varepsilon$$
$$\propto \text{access}(\text{memory}_1) * \text{size}(\text{memory}_1) * \varepsilon + \text{access}(\text{memory}_2) * \text{size}(\text{memory}_2) * \varepsilon + \text{access}(\text{memory}_3) * \text{size}(\text{memory}_3) * \varepsilon + \text{access}(\text{pmem}) * \text{size}(\text{pmem}) * \varepsilon$$

$\text{Size}(\text{memory}_i)$ equals to the total size of arrays mapped to memory_i and ε is the normalized value for energy consumption per cycle(8.98642E-08 J/cycle). Access time can be get from application simulation result.

Figure 2-1 shows the variances in system performance due to different mapping strategies. From the figure, we explain strategies 1 and 5 with respect to the previous formula. Figure 2-2shows the arrays assignments of mapping strategies 1 and 5.

Strategy	Memory 1	Memory2	Memory3	Execution time	Energy consumption
1	BDG	AF	CE	733684	80470
2	CDG	EF	AB	733620	80560
3	AF	BCG	DE	700884	77280
4	AE	DF	BCG	700852	77250
5	CE	BF	ADG	700852	77170

Figure 2-1 System performance variances due to different mapping strategies

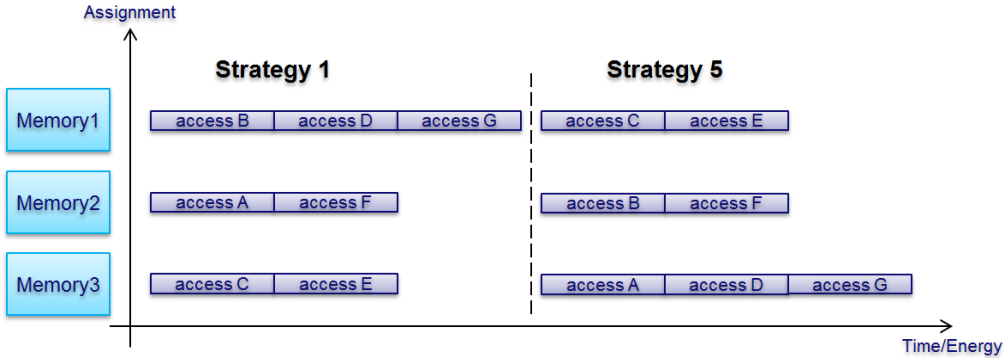

Figure 2-2 Arrays assignments of mapping strategies 1 and 5

Strategy	Accesses		Size		
	Data memory	Program memory	Memory 1	Memory 2	Memory 3
1	396288	730516	3096	2048	2048
5	396288	698708	2048	2048	3096

Figure 2-3 Memory accesses and array size of mapping strategies 1 and 5

According to the above estimated calculation formulas for execution time and energy consumption. The execution times for strategies 1 and 5 are 1126804 and 1094996. The energy consumptions for strategies 1 and 5 are 85.4+0.065*size(pmem) and 85.4+0.063 *size(pmem). The variances on system performance are caused due to the influence of the sequencer, which is used to realize control related interactions, as well as data accesses into the local memory. The interactions create a delay in accessing the memory connected with the sequencer. Consequently, the local memory connected to the sequencer is always slower than the others. We say that an architecture allocating memories with different speeds is asymmetric. The delay can be modeled by assigning

different speeds to the different local memories, as shown in Figure 2-4. The figure shows the ratio of each memory's performance compared to memory 3 and in this case, memory 1 is connected with the sequencer. The factor is a normalized value with respected to the value obtained from memory 3. Memory 1 is the slowest memory and has the highest energy consumption, so when arrays are mapped to memory 1, the system performance will be degenerated. This is the main reason why the system performance varies with different data mapping strategies.

Performance	Memory 1	Memory 2	Memory 3
Accessing time	1.35	1	1
Energy consumption	1.28	1	1

Figure 2-4 Memory performance normalized with memory 3

2.2 Related work

Memory subsystem plays a dominant role in the design of electronic systems, and it is a major contributor to the overall energy consumption of the entire system. As applications get more and more complex, the number of local memories used in the system and data to memory mapping strategy influence the system performance significantly.

One idea that can be applied to reduce energy consumption is memory partition, which is to divide the address space and to map the blocks to local memory. This idea has been applied to several prior works. Kandermir et al. [7] proposed a compiler-controlled dynamic on-chip scratch-pad memory management framework that uses both loop and data transformations to maximizing the reuse of data portions. Memory space partitioning strategy is applied to utilize the memory space efficiently. Benini et al. [8] proposed a recursive partitioning of the on-chip SRAMs address space into multiple banks and achieved an exploration of banking solution. Angiolini et al. [9] optimized the solution in [8], the cost function was shown to exhibit properties that allow applying a dynamic programming paradigm. Prior work illustrates that the memory partitioning is an effective method to reduce energy consumption. In this Master project, loop transformation – loop fusion, is applied to improve the reuse of memory locations. Moreover, memory partition is also applied to partition the address space of memory to fit the size of different arrays that mapped into local memory, in order to deal with the data to memory mapping problem.

2.3 Proposed solution

One solution to deal with the data to memory mapping problem is using graph coloring. Graph coloring is a way of coloring the nodes of a conflict graph such that no two

adjacent nodes in the graph share the same color. The graph coloring problem can be defined as follows: For a given conflict graph G = (V, E), and m colors, color all the nodes in all possible ways, so that no adjacent nodes have the same color. By apply the graph coloring algorithm, data mapping problem can be solved. Data mapping strategy can be found after completing the conflict graph coloring. It is possible to iterate the solution of the graph coloring problem, while iterating the number of used colors. This allows for exploring the impact of the number of memories on the improvements of the application mapping.

The input graph of the memory mapping analysis is a conflict graph among the arrays of an application. The conflict graph is able to model the conflict relation among different arrays in the application. The conflict graph can be built according to the following rules:

- The nodes of the conflict graph represent the arrays being used in a given application.
- The edges of the graph indicate that two arrays can be used in parallel, and therefore, they should not share the same memory.
- An array assigned into a certain memory location corresponds to a node assigned with a color, corresponding to this memory. Thus, two adjacent nodes should never be assigned with the same color.

In this way, the data mapping problem has been translated into the graph coloring problem, with the following attributes:

- The number of nodes in the conflict graph equals to overall number of arrays used in the application.
- The number of colors that can be used to coloring the conflict graph is the same as the overall number of memories provided in the processor.
- Two adjacent nodes, i.e. with edges between them indicate that two arrays access one memory simultaneously. Therefore, they should not be colored with same color.

In order to complete the conflict graph coloring, the conflict graph needs to be colored with the provided colors. Different graph coloring strategies indicate different solutions for data mapping.

3mm application is taken as an example to illustrate the building of the conflict graph. There are 7 arrays in the 3mm application. So the number of nodes in the conflict graph is 7. In our case, the arrays with data conflicts have to be distributed into 3 memories, which indicate that 3 colors have to be used to color the conflict graph. Nodes are used

to represent arrays that can be accessed simultaneously, and then edges are added indicating that these arrays have possible data conflicts.

Figure 2-5a shows the conflict graph for the 3mm application. There are 7 indexed nodes in the conflict graph, corresponding to 7 arrays used in the application. 3 memories are used and they are represented by 3 colors. Color blue, green and pink are used to represent memory 1, memory 2 and memory 3, respectively. The edges corresponding to the sunset of arrays that have parallel conflicts, such as array E, array A and array B, have edges between them indicating that node E, node A and node B cannot be colored with the same color. To complete the conflict graph coloring, the conflict graph needs to be colored with 3 provided colors without breaking constrains of the graph coloring problem. Figure 2-5b shows a coloring example. It is shown that node A, node B and node E are colored with different colors. Similarly, node C, node D and node E are colored with different colors, and the same for node E, node F and node G. This coloring strategy indicates that array A, array C and array G are mapped to memory 1; array B and array F are mapped to memory 2; array E and D are mapped to memory 3.

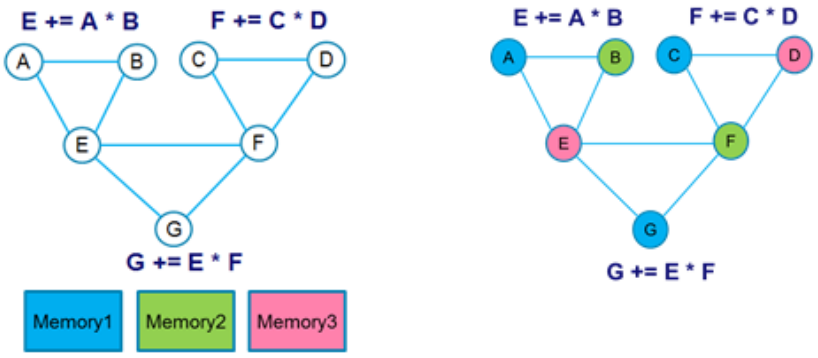

Figure 2-5 3mm a) conflict graph b) colored conflict graph

3. Chapter 3: Data conflict problem

In this chapter, data conflict problem arising when mapping data onto local memories is introduced and the strategies to reduce data conflicts are described. Thereafter, some code optimizations are discussed which were applied to make better use of the VLIW architecture in order to improve the system performance. Three applications are used as benchmarks to see how good the system performs after reducing the data conflicts and applying the code optimizations. The performances after each optimized steps are compared in terms of the execution time and energy consumption.

3.1 Problem analysis

The arrays used in the application are stored in the memory, so the memory will be accessed every iteration during data reading and writing. There are two kinds of conflict problems when accessing the memory: self conflicts problem and parallel conflicts problem. Self conflicts refer to simultaneous access of the same data array and parallel conflicts refer to simultaneous access of two or more different data arrays mapped into the same memory. A code segment taken from 3mm application is used as an example of data conflicts.

The code segment performs the computation of *E+=A*B*, which can also be written as *E=E+A*B.* It is shown that array *E* is self-conflicted, because it accesses the same memory both in reading and writing. Moreover, array E, A and B need to be accessed simultaneously to complete the computation, leading to the parallel-conflicts if the arrays are stored in the same memory.

3mm_partial	
1	/* E := A * B */
2	**for** (i = 0; i < 32; i++)
3	**for** (j = 0; j < 32; j++)
4	{
5	E[i][j] = 0;
6	**for** (k = 0; k < 32; ++k){
7	E[i][j] += A[i][k] * B[k][j];
8	}}

Figure 3-1 Partial code segment of 3mm

The conflict relations between arrays of an application can be captured by a $n \times n$ conflict matrix, where n is the number of arrays used in the application. The elements on diagonal position represent self-conflicts and the other elements represent possible parallel-conflicts for particular arrays. The conflict matrix below shows the conflicts among arrays E, A and B. Rows and columns of the matrix represent E, A and B in order.

In the conflict matrix, taking first row as an example, it is shown that E conflicts 32768 times with itself, while it conflicts 32768 times both with A and B. The total number of conflicts for arrays E, A and B are 98304, 65536 and 65536, respectively.

$$C = \begin{pmatrix} 32768 & 32768 & 32768 \\ 32768 & 0 & 32768 \\ 32768 & 32768 & 0 \end{pmatrix}$$

3.2 Reduce data conflicts

3.2.1 Self-conflicts

The memory in the ASIP instance considered is single ported, which means it can only have one single access at a time, either in reading or in writing. To solve the data self-conflict problem, data with self-conflicts need to be placed into a multi-ported memory to allow data to be read and written simultaneously. Due to the single port attribute of the memory, it is not possible to solve this kind of conflict from the hardware side alone, so we take advantage of the register files to deal with the data self-conflict problem. The calculations in the application use accumulation methods, to reduce the memory accesses to arrays that have self-conflicts are replaced with integers stored in the register file, and assign only the final computation results to the arrays stored in the memory.

As shown in Figure 3-2, integer *temp_E* is used instead of array E to store the intermediate computation result derived the iterations of the recursive process. By taking advantage of the register files, the number of memory accesses are reduced since the memory has no need to be accessed every time to read and write elements in array E in each iteration. Also, the overhead of data is reduced because self-conflicts of array E are eliminated, which benefits the execution time and energy consumption.

3.2.2 Parallel - conflicts

Parallel-conflicts appear when two or more different data arrays are mapped to the same memory and are accessed simultaneously. Application execution time will increase duo to the extra cycles cost for waiting. The parallel-conflicts can be reduced through distributing arrays that have parallel-conflicts in different local memories, so that they can be accessed simultaneously.

```
3mm_partial_register
1     /* E := A * B */
2     for (i = 0; i < 32; i++){
3       for (j = 0; j < 32; j++)
4         {
5             temp_E = 0;
6             for (k = 0; k < 32; ++k){
7                 temp_E += A[i][k] * B[k][j];
8             }
9             E[i][j] = temp_E;
10    }}
```

Figure 3-2 Partial code segment of 3mm after applying register files to reduce self-conflicts

Normally, the number of memories in the ASIP is fixed by the allocation, and it is equal to or fewer than the overall number of arrays having parallel-conflicts in an application. In our case, the number of memories is also constrained by the issue slots' number. Therefore, memory re-use is needed. For application 3mm with conflict matrix in page 20, one strategy to distribute arrays with parallel-conflicts to different memories is shown in Figure 3-3 below. It is shown that array E, array A and array B are assigned to different memories, as well as array F, array C and array D, and array G, array E and array F are assigned to different memories. With this mapping, arrays that have parallel-conflicts can be accessed in parallel in the execution.

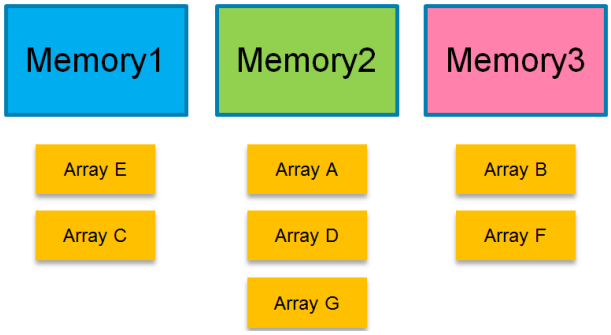

Figure 3-3 Assign arrays in application 3mm to reduce parallel-conflicts

3.2.3 API for data transfer

To map data from host memories to memories in the processor, the Hive Run-Time (HRT) Application Program Interface (API) can be used. HRT API can be used to control SH processor from a host processor. With the help of HRT, the host processor can upload and execute programs on SH processor. Also, data can be passed and stored in memories of SH processor [10]. A common reason for accessing variables in a processor's memory is to pass data to the application running on the processor. The following two functions can be used for data passing:

- **Pass data from host to processor**
  - void hrt_mem_store(hive_cell_id *cell*, hrt_mem_id *mem*, hrt_address *dst*, void* *src*, hive_uint *size*)
- **Pass data from processor to host**
  - void hrt_mem_load(hive_cell_id *cell*, hrt_mem_id *mem*, hrt_address *src*, void* *dst*, hive_uint *size*)

The first API function can be used to pass data from the host into arbitrary memory locations in the processor, meaning it copies copy *size* bytes from *src* into *dst* in memory *mem* of processor *cell*. The source address, *src*, is a host address, and the destination address, *dst* is a processor memory address. In the same way, the other API function can be used to copy data back from an arbitrary memory location of a processor into the host. The examples below show the usage of the API, for data transfer between host memory and processor memory.

- hrt_mem_store(c_2mm_fully, cluster01_cec0_dmem_mem, 0x100, A, 32*32* sizeof(int))

In this command, array E of size *32*32*sizeof(int)* is passed from *address E* in host to memory *cluster01_cec0_dmem_mem* in processor *c_2mm_fully* in address *0x100*.

- hrt_mem_load(c_2mm_fully, cluster02_cec0_dmem_mem, 0x100+5*sizeof(int) + 6*32*32+sizeof(int), G, 32*32* sizeof(int))

In this command, array G of size *32*32*sizeof(int)* is passed from address *0x100 + 5*sizeof(int) + 6*32*32 + sizeof(int)* in memory *cluster02_cec0_dmem_mem* processor of processor *c_2mm_fully* back to *address G* in host.

By setting the source and destination memory address, the needed data can be mapped in the appropriate location and realize arbitrary data mapping. These two data transfer functions are the basis for the automatic data mapping discussed in Chapter4.

3.3 Code optimization

After the above described methods were applied to eliminate the data conflicts in an application, several application code optimizations are applied in order to increase the instruction level parallelism. The original application is written in a sequential C format, which may prevent the scheduler to find optimized schedules and to exploit the ILP, for instance, due to too many functional calls and recursive functions. The code optimizations allow to remove some redundant overhead and to better exploit the VLIW architecture to obtain power efficiency and high performance. Also, for a specific

allocation, the code optimizations help to find optimized data mapping solution on a VLIW ASIP.

3.3.1 Loop fusion

Loop fusion is used to change the execution order of iterations or data accesses in the application where the iteration space is traversed. The loop fusion can be applied by merging adjacent loops with identical control into one loop or tiling the iteration space or inverting the execution order of a given loop, etc. Application 3mm is taken as an example.

```
3mm_orignal
1   /* E := A * B */
2   for (i = 0; i < 32; i++)
3     for (j = 0; j < 32; j++)
4       {
5           E[i][j] = 0;
6           for (k = 0; k < 32; ++k){
7               E[i][j] += A[i][k] * B[k][j];
8           }}
9
10  /* F := A * B */
11  for (i = 0; i < 32; i++)
12    for (j = 0; j < 32; j++){
13          F[i][j] = 0;
14          for (k = 0; k < 32; ++k){
15              F[i][j] += A[i][k] * B[k][j];
16           }}
17
18  /* G := E * F */
19  for (i = 0; i < 32; i++)
20    for (j = 0; j < 32; j++){
21          G[i][j] = 0;
22          for (k = 0; k < 32; ++k){
23              G[i][j] += E[i][k] * F[k][j];
            }}
```

Figure 3-4 Original code segment of 3mm

Originally, the matrix operations *E+=A*B* and *G+=E*F* are distributed in two independent loops, since there control are the same, these two loops can be merged into to one loop. The execution of *E+=A*B, F+=C*D, G+=E*F* are sequential in original code, so maximally, only 3 arrays can be accessed in parallel, and the execution time is long due to latencies in computation of intermediate data. After loop fusion is applied, there are 5 arrays that can be accessed in parallel. Loop fusion gives more parallelization possibilities, and these potential possibilities can be actually enhanced when more memories are allocated to the processor. If new memories are not allocated, the

performance of the system is not decreased by the application of loop fusion, if the total amount of data conflicts stays the same as original, no extra conflicts are generated.

The original application has high storage and bandwidth requirements, since elements in array E need to be written into a memory during the execution of the first loop and to be read back from the memory during the second loop executing when calculating the value of array G. This is repeated by Figure 3-5b, when the production consumption of the whole array E are degenerated. After loop fusion is applied, data locality is improved by combining loops references to the same array locality. As shown in Figure 3-5b that array E is consumed shortly after it has been produced. This optimizes the locality of data and reduces the requirements of memory and bandwidth.

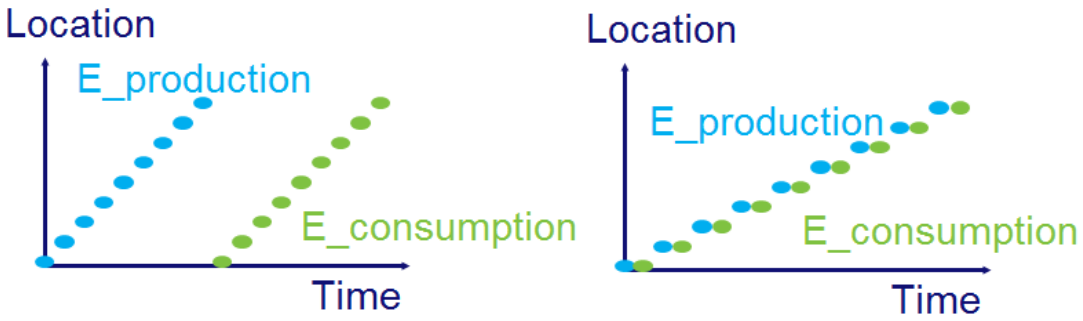

Figure 3-5 Improvement of data locality for array E in 3mm a) original b) after loop fusion

Also, the loop overhead can be reduced, since the loop overhead is cut down due to more compact code. Take array E as an example, loop fusion optimizes the loop nest by removing redundant overheads generated by the initialization, comparison and self-increment of variables. From Figure 3-6, it can be seen that the loop overhead decreased from 206115 times for the original code to 137379 times for the merged loops, which is 33.3% reduction.

index	initialization	comparison	self-increment
i	$1*3$	$32*3$	$32*3$
j	$32*3$	32^2*3	32^2*3
k	32^2*3	32^3*3	32^3*3
loop	3171	101472	101472
overhead	**206115**		

index	initialization	comparison	self-increment
i	$1*2$	$32+31$	$32+31$
j	$32+31+32$	$32^2+31*32+32$	$32^2+31*32+32$
k	$32^2+31*32+32$	$32^3+31*32^2+32^2$	$32^3+31*32^2+32^2$
loop	2145	67617	67617
overhead	**137379**		

Figure 3-6 Reduction of loop overhead for partial code segment of 3mm

3.3.2 Software Pipelining

One of the important features of VLIW is allowing for software pipelining, which reduces cycles per instruction (CPI) by assigning operations that can be executed in parallel to resources of the targeted processor. With software pipelining, the processor allows instructions of the next iterations to be fetched and partially executed while the processor is performing the instruction of the current iteration. As a result, instructions are allowed to be executed in parallel.

Instruction scheduling determines which instructions to execute in parallel. Silicon Hive compiler, HiveCC, is an instruction scheduling complier, which can group the operations that are able to be execute in parallel into one instruction, and the VLIW hardware executes the instruction containing parallelized operations. The principle of software pipelining is to schedule the code of the loop body and determine an integer number called initiation interval (II), which is the minimum interval between the beginnings of two successive loop iterations. HiveCC supports automatic software pipelining and it can be enabled by adding software pipelining pragma before the closing curly brace at the end of the loop in the code. For example:

- *#pragma hivecc pipelining=0*
    - *The complier tries to initiate a new loop iteration every cycle. Otherwise, it finds the minimum number of cycles for the II.*

3.4 Influence of data to memory mapping strategy

In section 3.3.2, data matrixes with parallel conflicts are assigned to different local memories to reduce the parallel conflicts. After data conflicts have been eliminated, loop fusion and software pipelining are applied to better exploit the VLIW architecture in order to further increase the energy efficiency and performance. However, for an application, many possible data mapping strategies may exist. For example for 3mm application mapped on an ASIP with 3 issue slots, there are 24 possible mapping strategies. A subset of possible mapping strategies is shown in Figure 3-7. Different data mapping strategies result in different system performance and energy consumption, due to the speed variance and different memory access times, as shown in Figure 2-4. In the previous work to eliminate parallel conflicts, a hand-designed mapping strategy assigning data to a specific memory is used, to distribute each array of the application manually onto a chosen memory. Thereafter, the application is simulated and the execution time and power consumption data are obtained. This method can be iterated in an exploration approach to discover the best mapping strategy with respect to system performance. However, it is inefficient and ineffective, as data need to be mapped

manually to the targeted memory and the application has to be simulated repeatedly. Consequently, this method requires a lot of iterative analysis passes and a long experimental time, which is very time-consuming and error-prone. Also, the code optimizations are applied based on the result after parallel conflicts are reduced, so the automation of data to memory mapping process and find a mapping strategy giving better system performance are therefore very important.

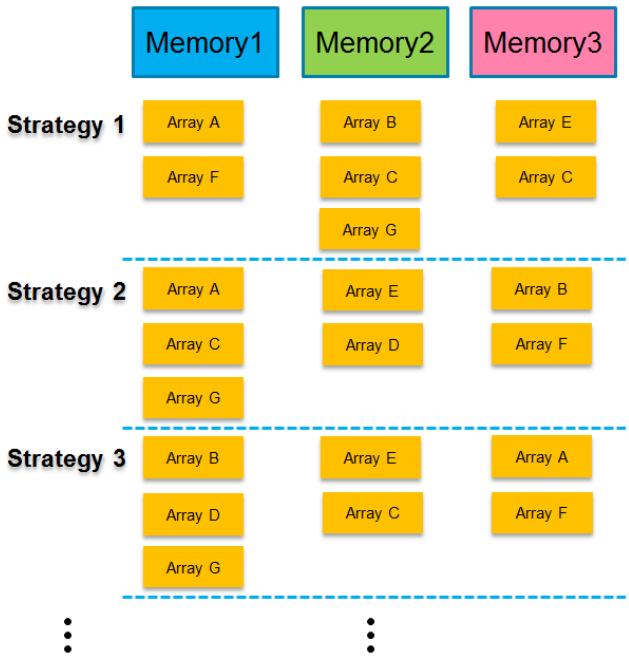

Figure 3-7 Subset of data mapping strategies for 3mm

4. Chapter 4: Memory mapping tool design

In this chapter, the proposed memory mapping tool is described. At first, the motivation for designing this tool is presented. Then, in section 4.2, the algorithm used for the tool and the implementation details of the memory mapping tool and are described. Thereafter, in section 4.3, the memory mapping tool is applied to applications from Polybench to explore how the tool performs the selection of optimal mapping strategy. Finally, a bottleneck of the tool is analyzed and a grouping method to reduce the solution space of data mapping is introduced, to eliminate the bottleneck.

4.1 Introduction

The research reported in this Master thesis explores and proposes a solution to the problem of automatic synthesis of parallel memories to VLIW ASIPs, and especially of data to memory mapping for ASIPs. Given a specific ASIP instance, with a fixed number and type of issue slots and an application C code, we want to find the minimal number of local memories and decide the mapping of arrays in the application into the ASIP local memories in order to make good use of the allocated memories and to get high performance and low energy consumption.

To efficiently and correctly deal with the data mapping problem, a memory mapping tool is proposed and implemented, that can automatically explore the solution space for the data mapping problem and simulate the application. After automating the solution space exploration, the proposed method can much more efficiently find Pareto optimal solutions of the memory mapping problem, with respect to the execution time and power consumption. The proposed design and exploration flow is shown as Figure 4-1. The input for the tool is an adjacency matrix, which can be computed from the data conflict graph of an application. The tool reads its input and based on a brute-force algorithm, computes the minimal number of colors needed to color the conflict graph without breaking the constrains. Given the minimal number of colors and the conflict graph, the possible mapping strategies are exhaustively enumerated and simulated. An output file that contains all the possible coloring strategies and relevant execution time and power consumption data is generated. According to the output, the Pareto mapping strategies with the highest performance and lowest energy consumption will be found. For each Pareto mapping strategy, the header file defining the source and destination address for each array in the SH APIs is generated. This defines the data mapping strategy, so that the application can be directly simulated.

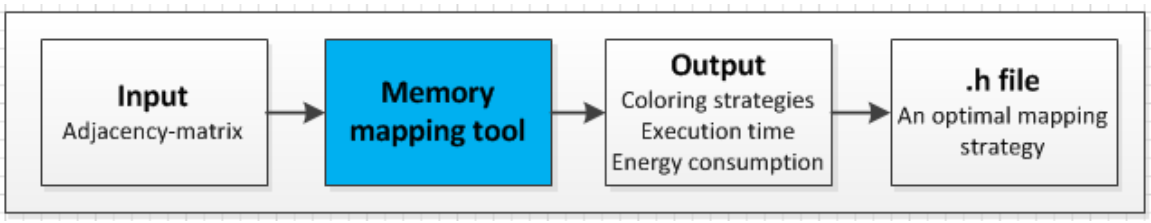

Figure 4-1 System flow of memory mapping tool

4.2 Memory mapping tool construction

4.2.1 Input

The input for the memory mapping tool is an adjacency matrix of the conflict graph, which indicates the adjacency relation among different nodes in the conflict graph. The adjacency matrix summarizes the information about the parallel conflicts. The elements of an adjacency matrix of a conflict graph are 0 and 1. The elements in the diagonal element are all 0, because there is no edge exists for one single node. If a graph has n nodes, the adjacency matrix can be given as by a $n \times n$ matrix M.

$$\mathrm{M_{ij}} = \begin{cases} 1, \text{if } (\mathrm{i},\mathrm{j}) \in E \text{ , where } E \text{ is the set of edges} \\ 0, \text{otherwise} \end{cases}$$

For Figure 2-5a, the adjacency matrix is as following:

$$\mathrm{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Rows and columns of the adjacency matrix represent node A, B, C, D, E, F and G in order. For example, the first row represents the relation between node A and all other nodes. Since node A is in conflict with node B and node E, in the position of node B and E, the values are set to 1 and the rest of values are set to 0.

4.2.2 Memory mapping tool

The memory mapping tool consists of two sub-parts: solution space exploration and application simulation, as shown in Figure 4-2. For the solution space exploration part, the tool reads the input adjacency matrix, and using the memory mapping algorithm,

finds the minimal number of colors needed to color the conflict graph without breaking the constrains. Subsequently, the solution space of coloring strategies is explored and all possible coloring strategies (data mappings) are found. The results are stored in a file. For the application simulation part, at beginning, a header file containing one mapping strategy extracted from the solution space is automatically generated that can be used to simulate the application. Thereafter, the application is automatically simulated according to the mapping strategy contained in the header file generated before, and subsequently, a power estimation tool is run, delivering information about the execution time and power consumption for the coloring strategy. Finally, the information on the possible solutions and related execution time and power consumption characteristics is stored in the output file. This process is iteratively executed until all the strategies in the solution space are simulated.

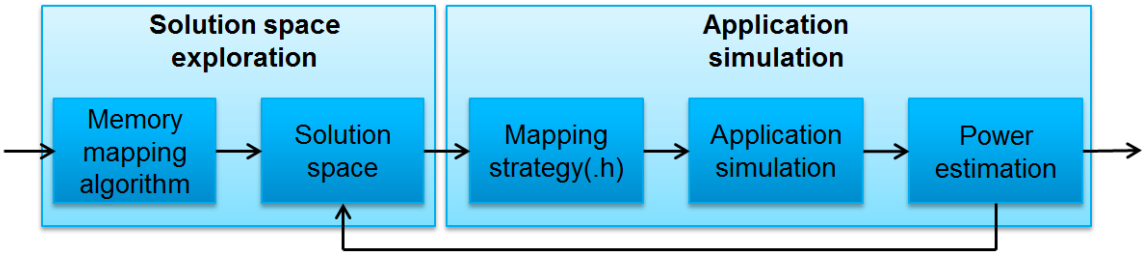

Figure 4-2 Sub-parts of the memory mapping tool

To find the solutions of the conflict graph coloring, the brute force algorithm is applied. To simplify the process, numbers instead of letters are used to index the nodes in the conflict graph and colors to color the graph. The following steps are taken to color the conflict graph:

- Choose the first node as the starting point and color the node with color 1.
- For the rest of nodes, it is checked if a node can be colored with color 1, in other words, if the adjacent nodes of this node are not colored or colored with a color different from 1, then, this node is colored with color 1.
- If the node cannot be colored with color 1, a new color – color 2 is introduced to color this node. If color 2 is not allowed to color the node, then color 3 is introduced and so on. Repeatedly, new color is introduced until the node can be colored properly.
- The above steps are repeated until all the nodes in the conflict graph are colored.

Figure 4-3 shows the design flow of the memory mapping algorithm and detailed processes are shown in appendix.

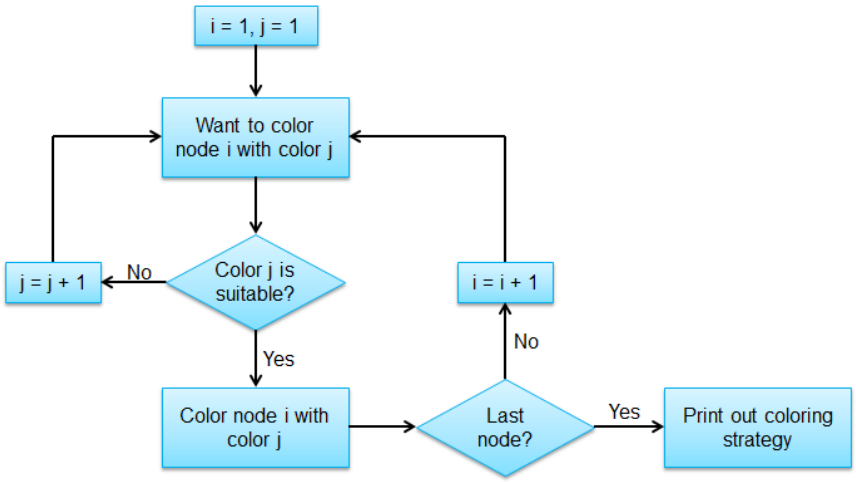

Figure 4-3 Process flow of memory mapping algorithm

4.2.3 Output

The output of the memory mapping tool is stored in an excel file. The file contains the following 3 parameters: mapping strategy, execution time and energy consumption. Figure 4-4a below gives a part of the output for the expectation of Figure 2-5a in section 3.2. As mentioned before, numbers are used instead of letters to index nodes and colors. For example for the first mapping strategy 1213321, it means that array A, array C and array G are mapped to memory 1; array B and array E are mapped to memory 2; array C and array D are mapped to memory 3. The colored conflict graph corresponding to this strategy is shown in Figure 4-4b.

strategy	ABCDEFG	#cycles	power consumption
1	1213321	281873	33510
2	1223312	280849	33420
3	1231321	312593	36560
4	1232312	282897	33610
5	1312231	344432	39730
6	1321231	309586	36270
7	1323213	281939	33510
8	1332213	279891	33320
9	2113321	312593	36570

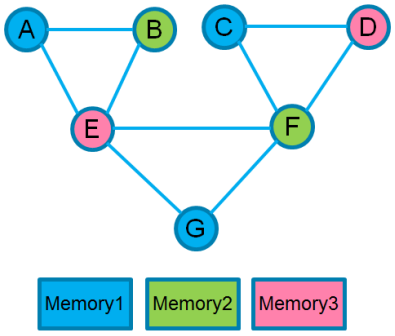

Figure 4-4 3mm a) Partial output of memory mapping tool b) Colored conflict graph according to mapping strategy 1213321

Based on the output, an optimised mapping strategy with high performance and low energy consumption can be found. Then, according to the index of the optimal strategy, the coresponding header file containing the memory mapping differences can be generated. The header file can be used to customize the SH ASIPs and for the simulation the application, detailed information can be found in Appendix.

4.3 Memory reduction

As we know, using more memories brings extra energy consumption, and at the same time, increases the size of the processor, so we want to reduce the number of memories allocated to the processor without influencing the system performance. One possible method is to take advantage of register files. As described in section 3.2.1, register files are used to eliminate the data self-conflicts. At the same time, the self-conflicts reduction can also reduce the number of used memories. For example, let's consider the operation *E = A*B*, after eliminating self-conflicts, the operation becomes *temp_E = A*B*. Now, instead of three memories, only two memories are needed to further eliminate the parallel-conflicts. Therefore, the total number of needed memory is reduced.

Reducing the memory's number also influences the relation among different conflict graph nodes, leading to changes in the conflict graph. For application 3mm taken as an example, the reduced conflict graph is show in Figure 4-5b. By applying the memory mapping tool on the reduced conflict graph, the minimal number of colors needed to color the conflict graph is two instead of three, what corresponds to one memory less than for the original conflict graph.

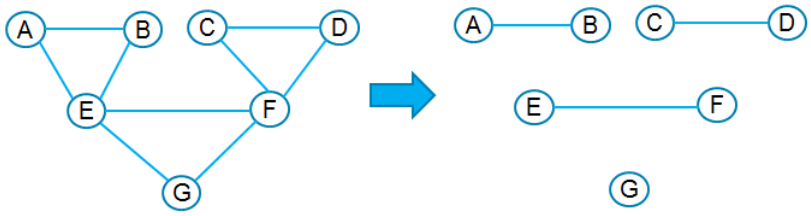

Figure 4-5 conflict graph of 3mm a) original b) reduced

This method is applied to application 3mm, 2mm and atax to explore the performance of the system after the number of memories allocated in the processor has been reduced. Figure 4-6 shows that reducing the memory's number substantively benefits the system's total area of the processor and can also benefits its execution time and energy consumption.

4.4 Solution space reduction

When using the memory mapping tool, the whole solution space of data mapping strategies can be explored and all possible solutions found. Thereafter, to test the performance of each mapping strategy, the application needs to be repeatedly simulated to get the corresponding execution time and energy consumption. For reasonably simple applications, if we only focus on the minimal number of memories,

the solution space can be small, and the optimized mapping strategy can be found easily. However, if we consider complex applications or larger than minimum number of memories, the solution space of data mapping becomes large, and the simulation time becomes the bottleneck of the whole exploration, since it takes time to simulate many different application mapping versions with different mapping strategies. One method to decrease the simulation time is proposed by Roel Jordans[1], being a member of the ASAM project team. The method reuses the statistics.txt file, which contains the cycle count per instruction and overall program cycle count. The statistics.txt file is an output of application simulation and also can serve as an input to power estimation tool. This method can be used to predict simulation results when the control flow of the application is not modified. It is usefully for example to shrink down the ASIP HW to the minimum requirements of a given version of the SW. In this case, it can save up to 60% of the simulation time. However, this method cannot be applied in our case, as the SW mapping changes, because different data mapping strategies influence the application's instruction scheduling, and in a consequence, the cycle count per instruction. For this reason, another solution is proposed.

Application	#memory	Execution time(#cycles)	Energy consumption(nJ)	Total area
3mm	2	280910	32710	97123944
	3	278898	33220	145230419
	Ratio	**1.007:1**	**0.98:1**	**0.67:1**
2mm	2	182647	19800	97123944
	3	210298	23140	145230419
	Ratio	**0.87:1**	**0.86:1**	**0.67:1**
atax	2	5488	634.4	97123944
	3	5490	653.3	145230419
	Ratio	**1:1**	**0.97:1**	**0.67:1**

Figure 4-6 Performance comparison between the original conflict graph and the reduced conflict graph

To reduce the solution space, the nodes of the conflict graph are partitioned into groups and partial optimal mapping strategy for each group is found. Figure 4-7a shows the partitioned groups of Figure 4-5b. First, the memory problem for group 1 is solved, mapping all the other arrays in a default memory. Then, keeping the optimal strategy of group 1 as constraint and keeping the arrays of group 3 and group 4 mapped in the default memory, the optimal strategy for group 2 is found. First, the optimal mapping strategy for group 1 is found. Then, mapping strategy of group 1 is kept as a constraint and the optimal strategy for group 2 is explored. This process is repeated to find the mapping strategy for the whole graph.

[1] http://www.es.ele.tue.nl/~rjordans/

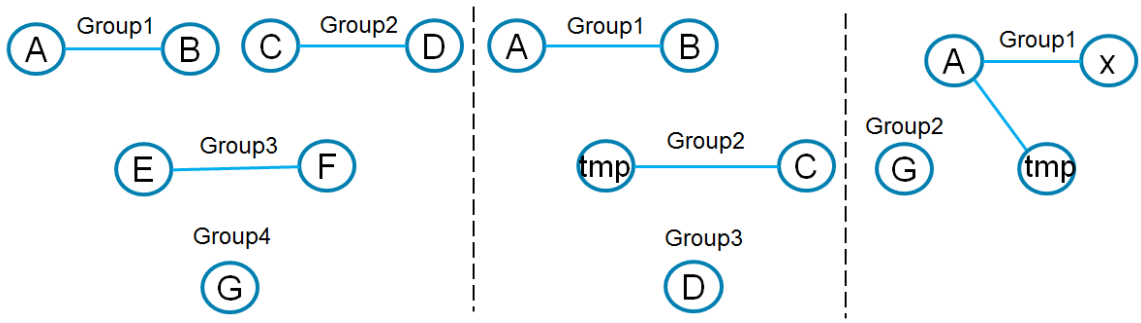

Figure 4-7 Grouped conflict graph a) 3mm b) 2mm

Application of this method reduces the solution space of the data mapping and makes the number of solutions limited. This method allows us to explore the system performances for more complex applications and when the number of memories in the processor increases. This method is applied to applications 3mm, 2mm and atax for different number of memories, to explore how this method reduces the solution space and computes the relevant system performances. Figure 4-8 shows the results for applications mapped to different number of memories and related number of solutions. The execution time and energy consumption in the table is the value of the optimal mapping strategy for each case. Due to a long simulation time, we only simulated the 2mm, 3mm and atax with the reduced conflict graph, mapped to three memories. The solution space is reduced to a large extent and the performance of the optimal mapping strategy found by grouping method is almost the same as the results found by simulating all the solutions.

Application	conflict graph	#solution	Execution time (#cycles)	Energy consumption (nJ)
2mm	original	108	181561	20240
	grouped	15	181561	20240
	Ratio	**7.2:1**	**1:1**	**1:1**
3mm	original	648	276849	33220
	grouped	21	278801	33220
	Ratio	**30.8:1**	**0.99:1**	**1:1**
atax	original	36	5488	649.5
	grouped	15	5488	649.5
	Ratio	**2.4:1**	**1:1**	**1:1**

Figure 4-8 Performance comparison between the original simulation method and the grouped simulation method

5. Chapter 5: Experimental results

This chapter describes the experiments results for data conflicts elimination and memory mapping tool. First part describes how the system performance improved by eliminating data conflicts and then, the efficiency of the memory mapping tool is illustrated.

5.1 Data conflicts elimination

In this section, 3 applications are used as benchmarks to apply the described strategies in Chapter 3 to eliminate data conflicts and to apply code optimizations to get better performance for the ASIP system. The applications are chosen from the Polyhedral Benchmark suite [11]: PolyBench/C, including 3mm, 2mm and atax.

- **3mm**. Three matrix multiplications distributed in three independent loops.

  E = A*B; F = C*D; G = E*F
- **2mm**. Two matrix multiplications distributed in two independent loops.

  tmp = A*B; D = C*tmp
- **Atax.** Matrix transposition and vector multiplication distributed in two independent loops.

  tmp = A*x; y = A*tmp

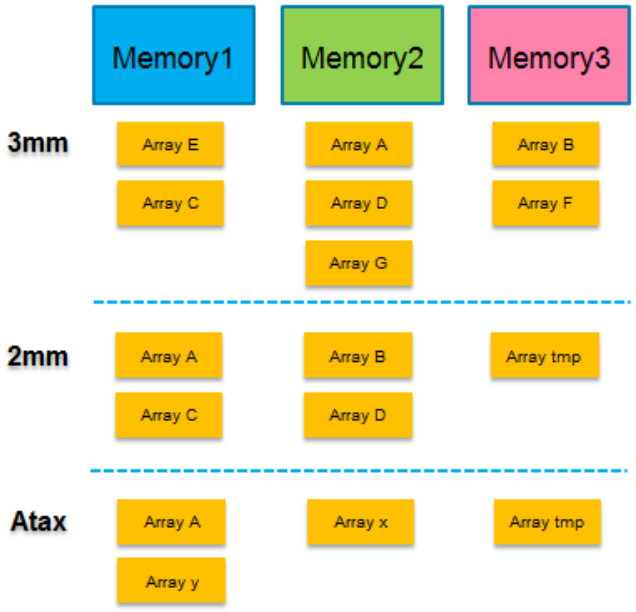

Figure 5-1 Data mapping strategies for 3mm, 2mm and atax to eliminate parallel-conflicts

For each application, the earlier discussed strategies used to eliminate the data conflicts and code optimizations are applied. Both execution time and energy consumption of each optimization step are compared. The data mapping strategies for dealing with

parallel-conflicts and applying loop fusion and software pipelining are shown in Figure 5-1. These mapping strategies assign data with conflict to different memories. For simplicity, *Memory 1*, *memory 2* and *memory 3* are used to represent real names of memory *ali0_dmem_mem*, *cluster01_cec0_dmem_mem* and *cluster02_cec0_dmem _mem*, respectively.

Execution time

We take advantage of the register files to reduce the data self-conflicts and we distribute data that have parallel-conflicts to different local memories to reduce parallel-conflicts. In this way, the stall cycles caused by conflicting simultaneous data accesses to the same memory are eliminated. This causes the execution time decrease. Also, loop fusion and software pipelining are applied to the code to optimize the data locality and instruction level parallelism. This increases the amount of parallelism available among instructions and gives the complier more flexibility to schedule the operations in a more effective way. Therefore, the number of clock cycles for execution drops. Figure 5-2a, Figure 5-3a and Figure 5-4a shows the execution time and improvements after optimization each step.

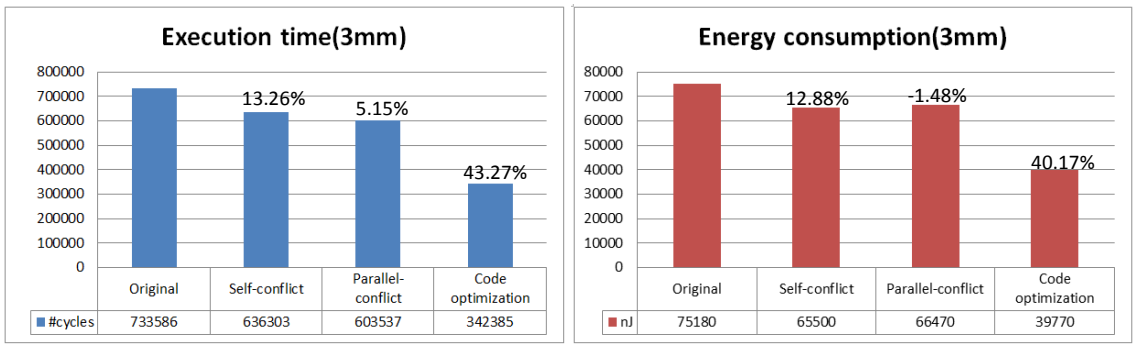

Figure 5-2 3mm a) Execution time b) Energy consumption

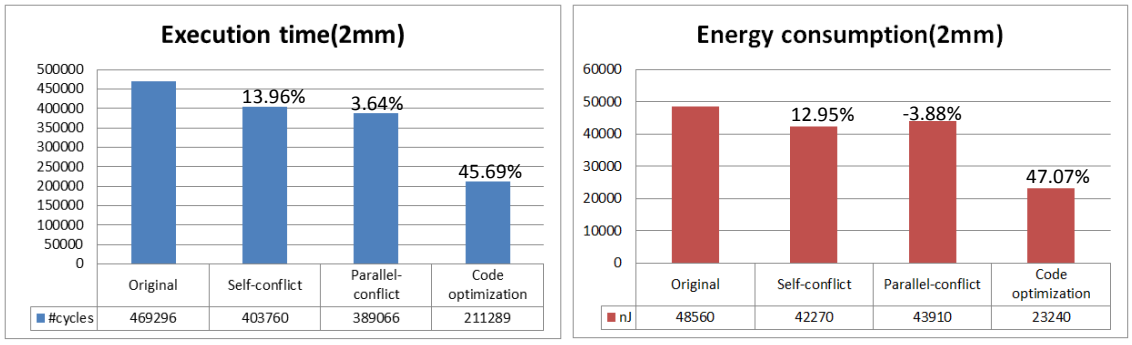

Figure 5-3 2mm a) Execution time b) Energy consumption

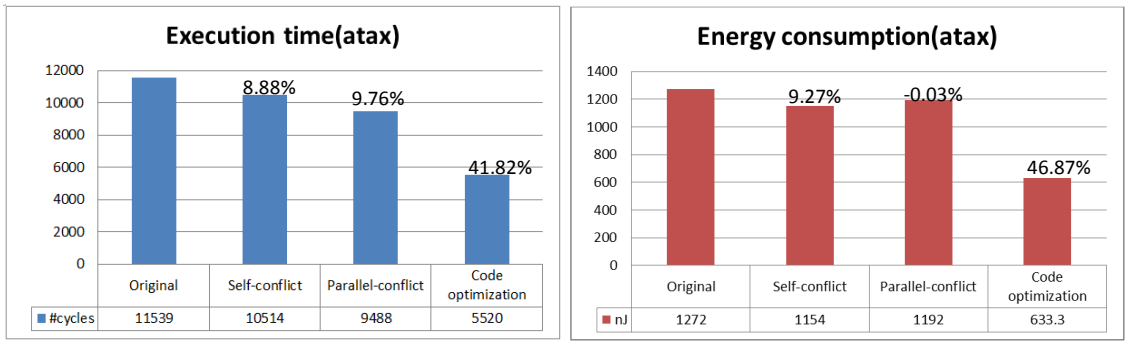

Figure 5-4 atax a) Execution time b) Energy consumption

From the figures one can conclude that removing the data conflicts and applying loop fusion and software pipelining benefit the execution time. Eliminating the data self-conflicts and parallel-conflicts gives 17.73%, 17.1% and 17.77% improvement on execution time for applications 3mm, 2mm and Atax, respectively.

With the decrease of the number of accesses to memories, as shown in Figure 5-5, the execution time decreases at the same time. After applying the code optimizations, 43.27%, 45.69% and 41.82% extra improvements are achieved in the execution time for applications 3mm, 2mm and atax, respectively.

step	total access to memory(normalized value)		
	3mm	2mm	atax
1. Original	1	1	1
2. Self-conflict	0.74	0.73	0.83
3. Parallel-conflict	0.71	0.73	0.78
4. Code optimizations	0.54	0.48	0.78

Figure 5-5 Memory access for applications after each optimization step (normalized value)

Energy consumption

As shown in Figure 5-3b, Figure 5-4b and Figure 5-5b, reducing data conflicts also substantially decreases the energy consumption. The reduction in energy consumption for applications 3mm, 2mm and atax, is 11.57%, 9.58% and 17.53%, respectively. Also, after applying of the code optimization, 40.17%, 47.07% and 46.87% extra improvements on execution time are achieved for applications 3mm, 2mm and atax, respectively. We notice that for application 2mm and 3mm, after data parallel-conflicts are eliminated, the energy consumption increases, which is different from expectation. One reason for this is two extra memories added to reduce parallel-conflicts, which brings extra dynamic and static energy consumption. Although elimination of the parallel-conflicts decreases the memory accesses and overheads, which reduces the energy consumption, the extra energy consumption generated by additional memories

exceeds the energy consumption reduction, leading to the increase of total energy consumption.

Another aspect that will influence the system performance is hardware utilization. Hardware utilization indicates the system design efficiency. High architecture utilization corresponds to a low waste of energy. In this Master thesis, hardware utilization is considered and measured as the utilization of the processor Issue Slots. There also exist other utilization aspects and measures, such as the interconnected network utilization, register file write-port and read-port utilizations. The Issue Slot utilization considered in this thesis is the average utilization in each cycle during the whole execution process of one application. Therefore, the issue slot utilization is the sum of utilization of all issue slots for all cycles against the total number of cycles. The corresponding utilization calculation formula is shown as below:

$$\text{utilization} = \frac{\sum \text{utilization for each cycle}}{\text{total number of cycles}}$$

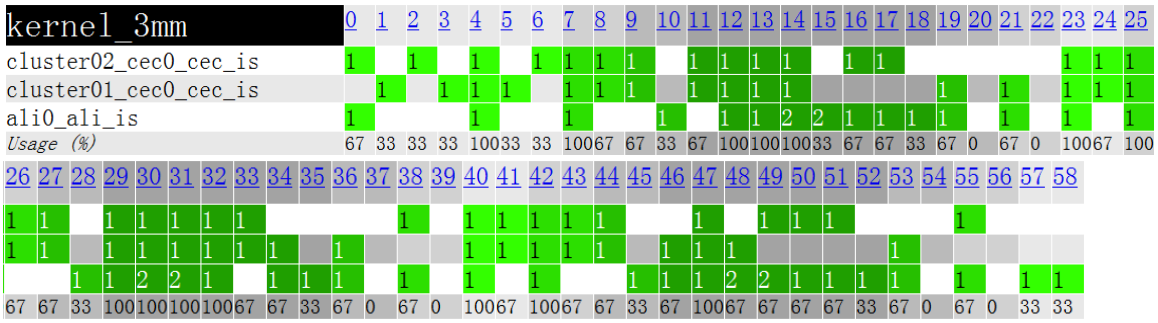

Figure 5-6 Resource utilization for 3mm

We take the original 3mm application to show how the utilization is calculated. The utilization for each cycle is shown in Figure 5-6. For one cycle, the utilization is 33%, 67% and 100% for the usage of one issue slot, two issue slots and three issue slots, respectively. The utilization for this application can be calculated as below:

$$\text{utilization} = \frac{\sum \text{utilization for each cycle}}{\text{total number of cycles}} = \frac{67 + 33 + 33 + \cdots + 33}{59} = 56.95\%$$

The issue slot utilization benchmark results are shown in Figure 5-7. The red bar is the final result after reducing the data conflicts and applying the code optimization. The improvement of the hardware utilization indicates the more efficient hardware usage, which is beneficial for the system performance.

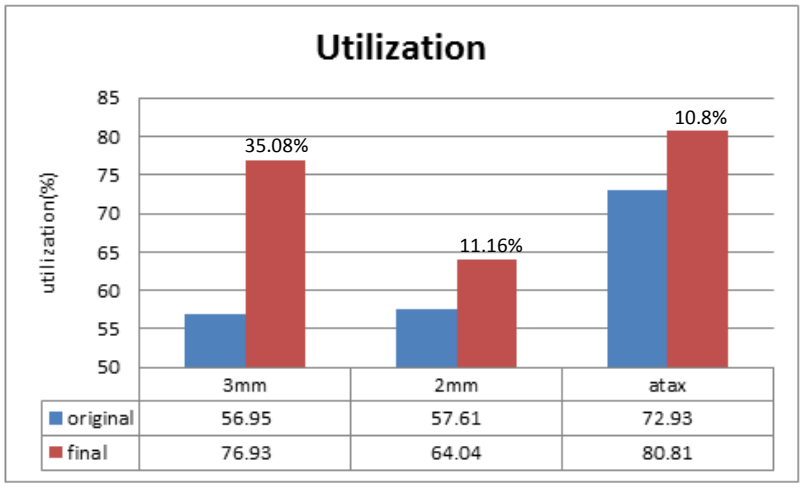

Figure 5-7 Issue slot utilization of application 3mm, 2mm and atax

Also, reducing the data conflicts allows the code optimization to result in better performance, as shown in Figure 5-8.

Application	Execution time			Energy consumption		
	with data conflicts	no data conflicts	**ratio**	with data conflicts	no data conflicts	**ratio**
3mm	22.62%	53.32%	**1:2.4**	20.84%	47.11%	**1:2.6**
2mm	28.08%	54.97%	**1:1.9**	23.62%	52.14%	**1:2.2**
atax	9.13%	52.16%	**1:5.7**	13.13%	50.21%	**1:3.8**

Figure 5-8 Improvements of code optimization after data conflicts are eliminated

5.2 Application of the tool to benchmark applications

The memory mapping tool is applied to benchmark applications 3mm, 2mm and atax to analyze and evaluate the influence of different data mapping strategies on the system performance and energy consumption. The ASIP instance is the same as used before in relation to eliminate the parallel-conflicts elimination. The best case and worst case values used to compare the execution time and energy consumption were computed for the same data mapping strategy for each application. Figure 5-9 shows that the execution time and energy consumption are both substantially influenced by the mapping strategy. For the execution time, compared to the worst cases, the values of the best cases have 19.03%, 23.77% and 0.56% improvements for 3mm, 2mm and atax, respectively. For energy consumption, the values of best cases have 16.93%, 27.73% and 1.74% improvements for 3mm, 2mm and atax respectively, versus worst case. With the help of the memory mapping tool, the hand-designed, time-consuming and error-prone

process is automated and the optimized mapping strategy can be found among all possible mapping strategies.

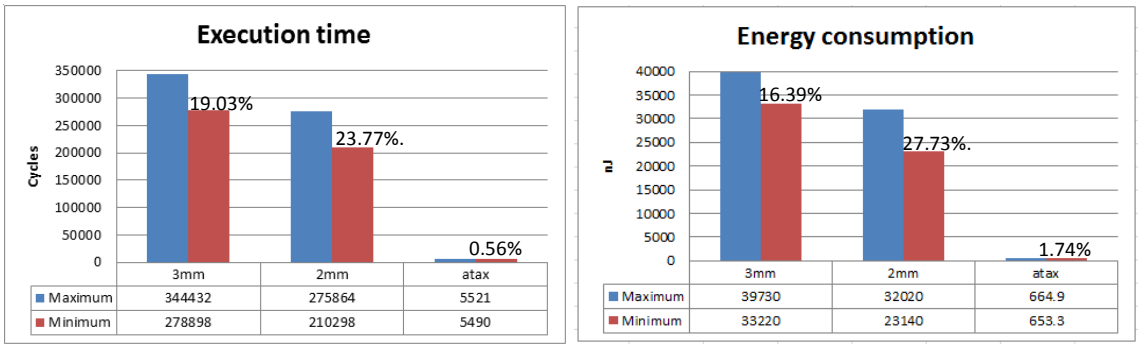

Figure 5-9 Performance comparison between optimal mapping strategy and worst case mapping strategy a) Execution time b) Energy consumption.

The variance in the results of different mapping strategies is due to the asymmetry of the ASIP architecture, i.e. the issue slots connected to the different memories are not equivalent. In the case of 3 issue slots ASIP, issue slot 2 (cluster01_cec0_cec_is) and issue slot 3 (cluster02_cec0_cec_is) are uniform, with same function units. However, issue slot 1 (alio_ali_is) is different, with more function units. Issue slot 1 is the sequencer and it needs to be connected to the program memory. This influences the performance when accessing memory 1 (ali0_dmem_mem). To test the speed of each memory, all the data of the application are mapped firstly on memory 1, then on memory 2 and then on memory 3, respectively, to evaluate the system performance. Figure 5-11 shows the ratio of each memory's performance compared to memory 3. The factor is a normalized value with respected to the value obtained from memory 3. Memory 1 is the slowest memory and has the highest energy consumption, so when arrays are mapped to memory 1, the system performance will be degenerated. This is the main reason why the system performance varies with different data mapping strategies.

Application		Memory 1	Memory 2	Memory 3
3mm	time	1.35	1	1
	energy	1.28	1	1
2mm	time	1.17	1	1
	energy	1.27	1.05	1
atax	time	1.18	1	1
	energy	1.15	1	1

Figure 5-10 Memory performance normalized with memory 3

6. Chapter 6: Conclusion

This chapter concludes the report. In the first part, the work and contributions of the Master project are summarized. The second part provides several ideas for future research and development.

6.1 Work and contribution

The research reported in this Master project addressed the automatic synthesis of parallel memories for VLIW ASIPs and data to memory mapping problem for ASIPs, when focusing on the problems mentioned in section 1.4. The aims of the project were the following:

- To analyze the problem of parallel memory synthesis and data mapping for VLIW ASIPs, when focusing on the above listed two main sub problems of this problem.
- To propose and implement a method to reduce the memory access conflicts.
- To develop an automatic memory mapping tool to deal with the data mapping problem. The tool has to automatically find an appropriate number of memories needed for a specific application, as well as, an optimal data mapping strategy.
- To perform experimental research to analyze how the data conflicts elimination method improves the system performance and to analyze the efficiency of the automatic memory mapping tool.

All the above aims have been satisfactorily realized. In particular, the project produced the following results:

- Data conflicts when data accessing to memory are reduced.
  There are two kinds of data conflicts when accessing the memory: self-conflicts and parallel-conflicts. The self-conflicts have been reduced through placing the data that have self-conflicts into the register files instead of placing them in the local memory. The parallel-conflicts have been reduced through distributing arrays that have parallel-conflicts into different local memories. Thereafter, some code optimizations were applied to make a better use of the VLIW architecture in order to improve the system performance. Benchmark results show that elimination of the data conflicts saves up to 18% of the execution time and 13% of the energy consumption. After the application of the code optimization, 46% and 47% extra improvements are achieved in the execution time and energy consumption, respectively.
- The automatic memory mapping tool is designed and implemented.

The tool is able to automatically explore and decide an adequate number of memories needed for a specific application, as well as, perform the solution space exploration for the data mapping problem and simulation of the results. Benchmark results show that among the different solution for data mapping problem, there are up to 27% and 28% variances on system performance between the best case solution and the worst case solution. With the help of the tool, the hand-designed, error-prone and time consuming mapping method became automatic, and an optimal solution for the data mapping problem can be obtained.

- A method to deal with the simulation time bottleneck is proposed.
  If the solution space is large, the simulation is the bottleneck of the whole exploration. To deal with the bottleneck, a grouped method is applied to reduce the solution space in order reduce the simulation time. This method considers only a limited number of solutions. Benchmark results shows that up to 96% of the solution space can be reduced, while roughly maintaining the same system performance.

6.2 Ideas for further work

The input for the memory mapping tool is an adjacency matrix of the conflict graph, which indicates the adjacency relation among different nodes in the conflict graph. For the purpose of this project, the adjacency matrix is constructed by hand. To improve the efficiency, one possible method is to use the C-to-Array-OL tool to automatically generate the adjacency matrix from Array_OL of the application.

In section 4.5, to reduce the solution space of mapping strategies, the nodes in the reduced conflict graph are partitioned into groups and an optimal mapping strategy by is found by searching for partial optimal mapping strategies of each group. However, for some applications, the reduced conflict graph cannot be partitioned into groups. One possible solution is to redefine and solve this problem as the Clique problem [12]. The nodes in the conflict graph can be partitioned by finding maximal cliques.

Bibliography

[1] ASIPs [Online]. Available: http://deep3.pkl.net/Files/Research%20Documents%20and%20Papers/ASIPs/ASIPs.doc

[2] "ARTEMIS," [Online]. Available: http://www.artemis-ju.eu/.

[3] "ASAM - Automatic Architecture Synthesis and Application Mapping," 2010 [Online]. Available: http://www.asam-project.org/.

[4] Jozwiak, Lech, Menno Lindwer, Rosilde Corvino, Paolo Meloni, Laura Micconi, Jan Madsen, Erkan Diken et al. "ASAM: Automatic Architecture Synthesis and Application Mapping." In Digital System Design (DSD), 2012 15th Euromicro Conference on, pp. 216-225. IEEE, 2012.

[5] Corvino, Rosilde, and Abdoulaye Gamatié. "Abstract Clocks for the DSE of Data-Intensive Applications on MPSoCs." Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on. IEEE, 2012.

[6] Florin Balasa, Noha Abuaesh, Cristian V. Gingu, Ilie I. Luican, Doru V. Nasui. "Energy-Aware Scratch-Pad Memory Partitioning for Embedded Systems"American University in Cairo, Fermilab, Microsoft, Inc., American International Radio, Inc.

[7] Kandemir, Mahmut, et al. "Dynamic management of scratch-pad memory space." Design Automation Conference, 2001. Proceedings. IEEE, 2001.

[8] Benini, Luca, et al. "Layout-driven memory synthesis for embedded systems-on-chip." Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 10.2 (2002): 96-105.

[9] Angiolini, Federico, Luca Benini, and Alberto Caprara. "An efficient profile-based algorithm for scratchpad memory partitioning." Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24.11 (2005): 1660-1676.

[10] Intel Benelux (Silicon Hive), "Silicon hive software development kit user manual".

[11] PolyBench/C: the Polyhedral Benchmark suite [Online]. Available: http://www.cs.ucla.edu/~pouchet/software/polybench/

[12] Clique problem [Online]. Available: http://en.wikipedia.org/wiki/Clique_problem/

Appendix

Memory mapping algorithm

The main part of the memory mapping algorithm is shown below. In the table, *matrix* means the adjacency matrix of the conflict graph and array *color* store the color for each node. Integer m is the number of color needed to color the graph and n is total number of nodes. In line 2, a flag is used to identify if the number of current colors is enough. In line 5, the function *colorchecking()* is used to check whether color *t* can be used to color the current node. If color *t* can be used to color current node, graph coloring process is continued to color the rest of the nodes in the graph. The function *graphcolor()* is called again in line 8 to perform graph coloring for the next node. In line 9, if all existing colors fail to color the node, then, the index of color will be increased, introducing a new color. The *colorchecking()* function is used to check if the current color can be applied to color the node, in other words, the function check coloring conflicts. If there is a coloring conflict, the return value is *false*, meaning current color cannot be used to color the node. If there is no coloring conflict, the returned value is *true*, and the current color is used to color the node.

```
1   def graphcolor(matrix,i,m,color):
2       flag = 0
3       if i<n:
4           for t in range(0,m):
5               if colorchecking(matrix,i,t,color):
6                   flag=1
7                   color[i]=t
8                   graphcolor(matrix,i+1,m,color)
9           if(flag!=1):
10              m+=1
11              color[i]=m-1
12              graphcolor(matrix,i+1,m,color)
13      else:
14          for j in range(0,n-1):
15              print(color[j])
16
17  def colorchecking(matrix,i,t,color):
18      for j in range(0,i+1):
19          if (matrix[i][j]==1 and color[j]==t):
20              return False
21      return True
```

Header file for 3mm

In the head file, information in Line 1 to Line 7 represents the indexes of memories where arrays are allocated and information in Line 8 to Line 14 represents the indexes of memories where arrays are mapped to.

```
1  #define m1 ali0_dmem_mem
2  #define m2 cluster01_cec0_dmem_mem
3  #define m3 ali0_dmem_mem
4  #define m4 cluster02_cec0_dmem_mem
5  #define m5 cluster02_cec0_dmem_mem
6  #define m6 cluster01_cec0_dmem_mem
7  #define m7 ali0_dmem_mem
8  #define M1 c_2mm_fully_ali0_dmem_mem
9  #define M2 c_2mm_fully_cluster01_cec0_dmem_mem
10 #define M3 c_2mm_fully_ali0_dmem_mem
11 #define M4 c_2mm_fully_cluster02_cec0_dmem_mem
12 #define M5 c_2mm_fully_cluster02_cec0_dmem_mem
13 #define M6 c_2mm_fully_cluster01_cec0_dmem_mem
14 #define M7 c_2mm_fully_ali0_dmem_mem
15 //1,2,1,3,3,2,1
```