Imperial College of Science, Technology and Medicine
Department of Computing

Optimising Runtime Reconfigurable Designs for High Performance Applications

Xinyu Niu

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Department of Computing of Imperial College London

January 2015

Abstract

This thesis proposes novel optimisations for high performance runtime reconfigurable designs. For a reconfigurable design, the proposed approach investigates idle resources introduced by static design approaches, and exploits runtime reconfiguration to eliminate the inefficient resources. The approach covers the circuit level, the function level, and the system level. At the circuit level, a method is proposed for tuning reconfigurable designs with two analytical models: a resource model for computational and memory resources and memory bandwidth, and a performance model for estimating execution time. This method is applied to tuning implementations of finite-difference algorithms, optimising arithmetic operators and memory bandwidth based on algorithmic parameters, and eliminating idle resources by runtime reconfiguration. At the function level, a method is proposed to automatically identify and exploit runtime reconfiguration opportunities while optimising resource utilisation. The method is based on Reconfiguration Data Flow Graph, a new hierarchical graph structure enabling runtime reconfigurable designs to be synthesised in three steps: function analysis, configuration organisation, and runtime solution generation. At the system level, a method is proposed for optimising reconfigurable designs by dynamically adapting the designs to available runtime resources in a reconfigurable system. This method includes two steps: compile-time optimisation and runtime scaling, which enable efficient workload distribution, asynchronous communication scheduling, and domain-specific optimisations. It can be used in developing effective servers for high performance applications.

Statement of Originality

The work described in this thesis was carried out in the Custom Computing Group, at Imperial College London, between 2012 and 2014, under the supervision of Professor Wayne Luk. The work in this thesis is entirely original except where duly referenced. In particular:

1 The CPU implementation of the option pricing application used in Chapter 3 was developed by Dr. Qiwei Jin.

2 The CPU, GPU and FPGA implementations of the particle filtering application used in Chapter 4 was developed by Mr. Thomas C.P. Chau.

3 The compiler front-end of the development tool described in Section 1.3 was developed by Mr. Paul Grigoras.

Copyright Declaration

Acknowledgements

First of all, I would like to thank my supervisor Professor Wayne Luk for his great support and guidance for the research work presented in this thesis. Wayne is a very responsible and knowledgeable supervisor. He has always made time to discuss my work, provided resources and help whenever I had problems with my research, given detailed corrections and comments on my English writing and presentation, and been supportive to my ideas which often sound unrealistic at the beginning. I cannot imagine a better mentor for my PhD study.

I would also like to thank all my colleagues from the Custom Computing Research Group for the pleasant and inspiring working atmosphere: Brahim Betkaoui, Pavel Burovskiy, Tobias Becker Bridgette Cooper, Thomas Chau, Gabriel De Figueiredo Coutinho, Gary Chow, Kit Cheung, Stewart Denholm, Andreas Fidjeland, Andreea-Ingrid Funie, Ce Guo, Liucheng Guo, Paul Grigoras, Gordon Inggs, Qiwei Jin, Maciej Kurek, Adrien Le Masle, Nicholas Ng, Anson Tse, James Stanley Targett, Kuen Hung Tsoi, Timothy Todman, and Jinzhe Yang. They not only provided help and inspiration for my research work, but also made my PhD study enjoyable.

I would like to acknowledge UK EPSRC, the European Union Seventh Framework Programme under Grant agreement number 287804, 248976 and 257906, the HiPEAC NoE, Maxeler, Altera and Xilinx for their support in my research work.

Finally, I would like to thank my family: my wife Mengqiu (Rachel) Xiao, my parents Yuhe Niu and Chengyan Yin, my sister Xinhui Niu, and my brother-in-law Jidong Yan, who have always been there for me, and believed in me even when sometimes I didn't. I hope this thesis is a start of my returning to my family.

Glossary

We summarise the key concepts and commonly used terminologies in this thesis as follows.

- 3-D: Three-Dimensional.
- ALU: Arithmetic Logic Unit.
- ASIC: Application-Specific Integrated Circuit.
- BRAM: Block Random-Access Memory.
- CB: Connection Block.
- CLB: Configurable Logic Block.
- CMOS: Complementary Metal–Oxide–Semiconductor.
- CPU: Central Processing Unit.
- DDR3: Double Data Rate 3 (a DRAM interface specification).
- DRAM: Dynamic Random-Access Memory.
- DSP: Digital Signal Processing.
- DFG: Data-Flow Graph.
- DPGA: Dynamically Programmable Gate Array.
- FIR: Finite Impulse Response.
- FPGA: Field Programmable Gate Array.
- FR: Full Reconfiguration. The whole configuration file of an FPGA is updated during runtime reconfiguration.
- FF: Flip-Flop.
- FIFO: First In, First Out.
- GPP: General-Purpose Processor.

- GPU: Graphics Processing Unit.
- HLS: High-Level Synthesis.
- ICAP: Internal Configuration Access Port.
- INLP: Integer Non-Linear Programming.
- I/O: Input/Output.
- JTAG: Joint Test Action Group (interfaces for testing and configuration).
- LUT: Look-Up Table.
- MIMD: Multiple Instruction, Multiple Data streams.
- OP: Option Pricing. OP refers to a type of benchmark applications, including OP, Bond OP and Barrier OP.
- PDE: Partial Differential Equation.
- PR: Partial Reconfiguration. Only part of a configuration file of an FPGA is updated during runtime reconfiguration.
- PF: Particle Filtering. PF is a benchmark application.
- RDFG: Reconfiguration Data Flow Graph.
- RNG: Random Number Generator.
- RTM: Reverse Time Migration. RTM is a benchmark application.
- SB: Switch Block.
- SIMD: Single instruction, Multiple Data streams.
- SDR: Software Define Radio.
- SoC: System-on-Chip.
- SRAM: Static Random-Access Memory.
- SSE: Streaming SIMD Extensions.

General

- Reconfigurable design: a (or a group of) customised hardware design(s) for an application, mapped into reconfigurable devices. A reconfigurable design is capable of executing a complete application, while a hardware design may only support a part of an application.

- Hardware efficiency: the effectiveness of a hardware design. A resource unit is considered utilised if it contributes to the generation of computation results. In the optimal case, all available resources of a design are instructed / configured to work actively at each clock cycle, which leads to the theoretical peak performance of the design. The hardware efficiency is calculated as the ratio between measured throughput TH_{mes} and theoretical peak throughput TH_{the}. As a resource unit can refer to different resource types with different resource granularities, the inefficient use of resources is defined at different levels of granularities.
  - redundant logic gates (circuit level): the idle logic units in an arithmetic operator during runtime. This happens when some of input bits are fixed during computation, therefore some logic branches of the logic operator are never activated during runtime.
  - idle functions (function level): the function modules that are idle during runtime. A function module is idle when its dependent data are not available. This happens when the data are being computed in other modules, or the on-chip / off-chip communication channels cannot satisfy the communication bandwidth requirements.
  - runtime available nodes (system level): the computing nodes that are not available when an application is launched, but become available before the application is finished. This happens in reconfigurable systems with multiple computing nodes, where applications are launched from time to time. When not properly used, the runtime available nodes remain idle in the system.

- Runtime reconfiguration operations: operations to update the configuration of a reconfiguration design, i.e., the configuration file stored in on-chip SRAM arrays.
  - for partial reconfiguration, the operations include stalling the reconfigured modules, updating the partial configuration file, and starting the new module.

  - for full reconfiguration, besides the configuration update and execution control, off-chip memory data need to be preserved. The off-chip data are managed by on-chip memory controllers, which are not functional during reconfiguration. Therefore, for full reconfiguration, additional operations are needed to preserve the off-chip memory data during runtime reconfiguration.

- Runtime reconfiguration overhead: time to finish runtime reconfiguration operations.

- Runtime scenario: a period of runtime when application requirements or hardware resource status stay the same.

- Configuration / configuration file: the synthesised configuration file for a hardware design customised for a specific runtime scenario. A configuration file can be downloaded into FPGAs to define the implemented operations.

- Static design: a reconfigurable design that handles all the runtime scenarios in a single configuration file. The implemented circuits remain static during runtime. No reconfiguration takes place during runtime.

- Dynamic design: a reconfigurable design that handles each runtime scenario with a customised design. Each design is synthesised as a configuration file, and a dynamic design uses runtime reconfiguration to switch between different configuration files.

- *well-behaved* data-path: a hardware implementation of an algorithm that that generates one set of output data for each input data set, with input data fed into the data-path one set per clock cycle (the concept of *well-behaved* data-paths is discussed in detail in Section 2.2.3).

- Idle resource unit: a circuit necessary to support a given application which can become inactive during runtime.

- Parallelism: the number of replicated *well-behaved* data-paths in a hardware design.

Circuit-Level

- General operators: arithmetic operators that can process any input data.
- Constant operators: arithmetic operators that can only be used when some input values are constant over time.
- Algorithm instance: an instance of an algorithm, with initial algorithm parameters.
- Algorithm design space: for an algorithm instance, the range of algorithm parameters where these parameters (and thus constant coefficients) can vary without compromising algorithm mathematical correctness.
- Constant coefficient set: a point in an algorithm design space that specifies constant values used in constant operators.
- Runtime scenario: a period of time when the used constant coefficient set stays the same.

Function-Level

- Segment: it contains a group of application functions that can be executed concurrently, without any idle application function.
- Configuration: it contains a group of segments that are optimised together to achieve the maximum parallelism, which are synthesised as a configuration file.
- Partition: it contains a combination of configurations and is capable of accomplishing the application functionality. Each application partition is a valid way of executing the application, i.e., a complete reconfigurable design.
- Runtime scenario: a period of time the active functions in an application stay the same.

System-Level

- Resource availability: whether a computing node is free to accommodate the target reconfigurable design.
- Runtime scenario: a period of time when the computing node availability of a reconfigurable system stays the same.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ever-increasing demand for high-performance computing solutions has exceeded the clock frequency scaling of General-Purpose Processors (GPPs) for about a decade by now [Sut05]. GPPs, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), control data movements and arithmetic operations by instructions. Computer applications are mapped into such processors as compiled instructions, to reuse the processing units of the processors as much as possible. Each instruction typically goes through fetch, decode, read, execute, and write stages to complete its operation, which can limit the efficiency of program execution.

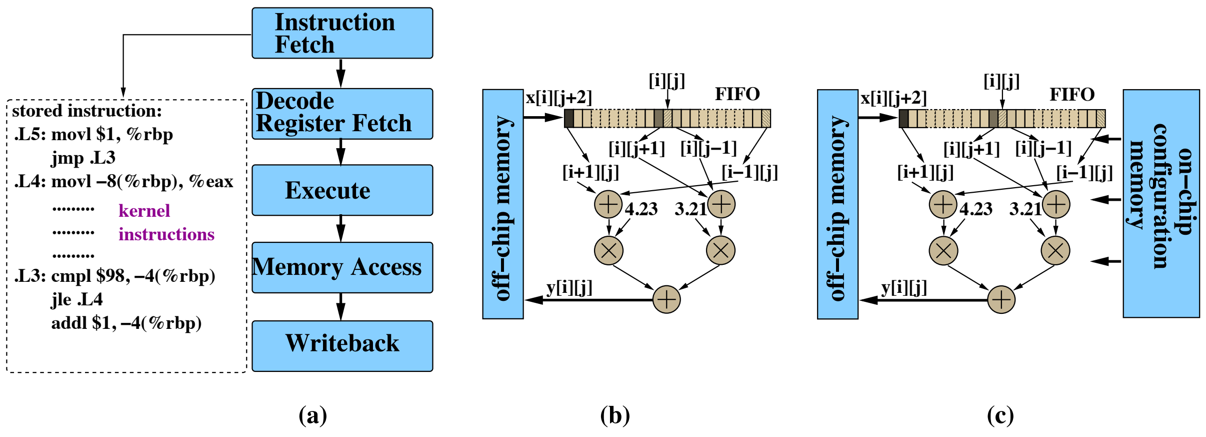
As an example, consider a 2-D convolution problem in Algorithm 1, Figure 1.1(a) shows the compiled instructions for this algorithm. Traditionally, the users of GPPs, from small-scale embedded application developers to large-scale datacentre operators, relied on performance improvement in GPPs. Take Intel CPUs between 1970 and 2010 as an example. While transistor count over this period follows Moore's Law, clock frequency and performance per clock cycle have been stagnant since 2004 mainly due to power density and heat dissipation issues [Sut05]. On the other hand, based on a report from International Data Corporation (IDC) [IDC], the volume of data in the world doubles every two years. With data size increasing, high-performance applications become more and more time-consuming. For Reverse-Time Migration (RTM), an application with 3-D stencil computational kernel (similar to the 2-D convolution in Algorithm 1), simulating a wave propagation process within a 25 km^2 area with 5000 m depth for

Algorithm 1 Example algorithm to demonstrate hardware efficiency for different architectures.

Input: Input arrays x with data size ds*ds.
Output: Output array y

```
for i = 1 → ds-1 do
    for j = 1 → ds-1 do
        y[i][j] = (x[i][j+1] + x[i][j-1]) * 3.21 + (x[i+1][j] + x[i-1][j]) * 4.23;
    end for
    tmp = y; y = x; x = tmp;
end for
```

a period of 0.4 s requires 63.4 tera floating-point operations, which would consume 1.4 hours to execute on an Intel Xeon X5650 CPU with 6 cores running in parallel. As simulation time and data size (e.g. larger simulated area, higher simulation granularity) increase, the execution time would be even longer. This situation calls for new architectures and techniques that can meet the continuous increase in demand for high-performance computing.



```
stored instruction:
.L5: movl $1, %rbp
     jmp .L3
.L4: movl −8(%rbp), %eax
     .........   kernel
     .........   instructions
     .........
.L3: cmpl $98, −4(%rbp)
     jle .L4
     addl $1, −4(%rbp)
```

Figure 1.1: Implementation of Algorithm 1 in (a) General-Purpose Processors (GPPs), (b) ASICs, and (c) reconfigurable architectures. GPPs decompose a computing program into general instructions. ASICs and reconfigurable architectures construct customised data-paths.

Application operations in Application-Specific Integrated Circuit (ASIC) designs are often implemented as customised hardware. Typically, ASIC designs use pipelined data-paths to execute communication and computation operations, with communication and computation operations respectively implemented as hard wires and dedicated operators. Figure 1.1(b) shows an example ASIC design for Algorithm 1. A customised memory architecture shifts in one datum each clock cycle, and connects the data to be used in current clock cycle into a data-path. The memory architecture provides enough on-chip memory bandwidth to keep the data-path active throughout runtime, achieves maximum data reuse ratio as data shifted out of on-chip memory are no longer needed in the future, and has no cache miss as new data are streamed into the memory cycle by cycle. The arithmetic operators in the data-path are pipelined. In each clock cycle, this ASIC design streams input data from data memory, and generates one output data set. The necessity for instruction fetch and decode is eliminated. However, such ASIC designs sacrifice generality since the customised hardware, once fabricated, cannot be reused by other applications. Given the rapidly evolving computing applications and the substantial ASIC development cost, fabricating an ASIC chip for a specific application only makes economic sense if it can be sold for a large volume.

Field-Programmable Gate Arrays (FPGAs) balance the requirements for high-performance customised designs and the necessity to adapt to application evolvement. FPGAs use configuration memory to store post-fabrication circuit configurations. By updating the stored configuration data, the implemented circuits can be reconfigured in accordance to application specifications. The ability to implement customised circuits for various applications without going through chip fabrication process drives the use of FPGAs as hardware accelerators, which offload the computationally intensive functions in applications. Figure 1.1(c) shows a common FPGA design for Algorithm 1. Compared with software implementations, FPGA designs often achieve higher efficiency as customised data-paths are implemented. However, in order to support circuit reconfigurability, FPGAs suffer from overhead in area and delay since additional configuration memory and reconfigurable routing / logic fabric need to be accommodated on-chip. [KR07] compared two circuits synthesised from the same design, with one mapped to an FPGA, and the other one synthesised as an ASIC, in the same 90-nm CMOS technology. Compared with ASIC designs without the overhead to support reconfigurability, implementing hardware designs in FPGAs on average increases design area by 18 times, and reduces clock frequency by 3 times. The gap can be even bigger if ASIC designs are manually optimised rather than automatically synthesised. In average, FPGA designs use 15 times to 25 times larger area, and run at 3 times to 10 times slower clock frequencies.

1.1 Motivation

The example in Figure 1.1 shows the optimal scenario for a customised hardware design:

- a relatively long data-path with operators deeply pipelined;
- an optimal memory architecture provides required data each clock cycle with maximum data reuse ratio and no cache miss;
- vectorisable design as data-paths can be replicated and the customised memory architecture can be modified to process multiple data per clock cycle (for the example in Figure 1.1, P data-paths can be replicated, and at clock cycle 1, $y[1, ..., N]$ will be calculated instead of just $y[1]$);
- large data volume for the same computational kernel to be applied iteratively.

This requires linear data access pattern and simple control-flow operations. However, in practice, a large portion of applications does not meet these requirements. With complex control operations (e.g. *if-else* operations) and dynamic data access operations (e.g. dynamic pointers, array indices that depend on runtime variables), these applications often perform better on GPPs such as CPUs and GPUs.

In order to take a wide range of applications into consideration, we evaluate architectures in term of their hardware efficiency and generality. For a design mapped into a hardware architecture, we define its hardware efficiency as the ratio between its measured throughput and its theoretical peak throughput. We calculate the theoretical peak throughput by assuming all the used processing units are active through runtime. For example, the theoretical peak throughput of a CPU design is calculated assuming all the Arithmetic Logic Units (ALUs) used in this design (typically all the ALUs in a CPU) are actively processing data cycle by cycle. For a reconfigurable design, the used processing units refer to its consumed logic units (LUTs, DSPs, and FFs). Limited by the routing capacity of an FPGA, a reconfigurable design often uses less than 100% of the logic units in an FPGA [DeH99].

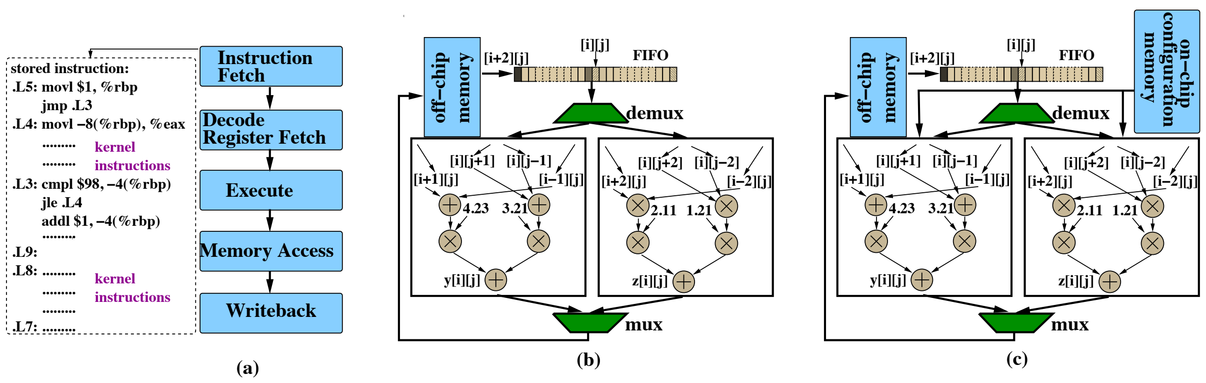
Algorithm 2 Example algorithm to demonstrate hardware efficiency for different architectures.

Input: Input arrays x, y with data size ds*ds.
Output: Output array z

```
for i = 1 → ds-1 do
    for j = 1 → ds-1 do
        y[i][j] = (x[i][j+1] + x[i][j-1]) * 3.21 + (x[i+1][j] + x[i-1][j]) * 4.23;
    end for
    tmp = y; y = x; x = tmp;
end for
for i = 1 → ds-1 do
    for j = 1 → ds-1 do
        z[i][j] = (y[i][j+2] + y[i][j-2]) * 1.21 + (y[i+2][j] + y[i-2][j]) * 2.21;
    end for
    tmp = z; z = y; y = tmp;
end for
```

For applications that meet the requirements listed above, customised hardware architectures can achieve the theoretical peak throughput of these applications. For the example reconfigurable design in Figure 1.1(c), at each clock cycle, first, one input datum is streamed from off-chip memory. Second, all data stored in on-chip memory architecture are shifted right by one position. Third, the data that appear at the output ports of the on-chip memory architecture steam into the connected data-path operators. Finally, the data-path generates one result, based on data streamed into the data-path in previous cycles. After an initial overhead to fill the on-chip memory architecture, each implemented operator executes one arithmetic operations per clock cycle, i.e., the hardware efficiency of this architecture reaches 100%. Experimental results show that for practical high-performance applications, customised architectures for seismic imaging [NCJ+13a], financial modelling [TL08a], and matrix processing [DRM14] can approximate optimal hardware efficiency (higher than 90%). For a RTM application, an optimised 6-thread CPU design running on Intel Xeon X5650 achieved 13.3 GFLOPS [NCJ+13a]. Since the peak throughput of Intel Xeon X5650 is 63.984 GFLOPS [Int], the CPU hardware efficiency for RTM is 21%. An Nvidia Tesla C2070 GPU achieved 83.4 GFLOPS for the same application. However, compared with its peak throughput of 1.03 TFLOPS [Nvi], the hardware efficiency is even lower.

While customised hardware architectures achieve higher hardware efficiency for certain applications, GPPs show higher generality for a wide range of applications (i.e. a wide range of applications can be supported without sacrificing performance). Algorithm 2 contains two convolution modules. Since the second module depends on the output from the last iteration of the first module, these two modules cannot run in parallel. Figure 1.2 shows the software and hardware implementations for Algorithm 2. For GPPs, the two modules are compiled as two instruction loops (*.L4* and *.L8* in Figure 1.2). The two instruction loops are executed one by one, to share computational resources in GPPs (such as ALUs in CPUs). When data size for each module (*ds* in Algorithm 2) is large enough, the impacts on hardware efficiency is negligible. For the previous RTM example, the same hardware efficiency (21%) can be achieved if two stencil modules need to be supported and data size is large enough. For the hardware designs, either ASIC or reconfigurable, one additional hardware module is implemented to cover possible runtime operations. In Figure 1.2(b) and (c), two hardware modules are implemented, one for each convolution module. Output data from the two modules are multiplexed based on cycle counters, and thus only one of the modules is active at each clock cycle. In this thesis, we define idle resource units as *circuits necessary to support a given application which can become inactive during runtime*. In this example, while the hardware efficiency of the software designs is still 21%, the hardware efficiency of the ASIC design and the reconfigurable design reduces to 50%, i.e., at each clock cycle, only 5 out of 10 arithmetic operators are active. As computing problem complexity (e.g. the number of branches in an *if-else* expression, or the number of modules that are not active at the same time) increases, the number of idle resource units increases, and the advantages of customised hardware architectures rapidly disappear.



```
stored instruction:
.L5: movl $1, %rbp
     jmp .L3
.L4: movl −8(%rbp), %eax
     .........   kernel
     .........   instructions
.L3: cmpl $98, −4(%rbp)
     jle .L4
     addl $1, −4(%rbp)
     .........
.L9:
.L8: .........   kernel
     .........   instructions
     .........
.L7: .........
```

Figure 1.2: The hardware efficiency of customised designs is reduced to ensure generality for complex operations. For an algorithm with two modules that cannot run in parallel, we present the algorithm implementations for (a) GPPs, (b) ASICs, and (c) reconfigurable architectures.

Runtime reconfiguration is a technique to update the configuration of a reconfigurable design, which provides FPGA designs another dimension to improve design efficiency and generality. Conventional reconfiguration designs keep configuration data static during runtime, and therefore the implemented circuits remain the same once initialised. With runtime reconfiguration, a reconfigurable resource unit can accommodate different operations by updating the configuration information. For the reconfigurable design in Figure 1.2(c), instead of implementing two data-paths, we can develop two designs, with each design handling one `if` branch. At each time, one of the two designs is active, and only the active design is configured into FPGAs. This brings two benefits: (1) improved efficiency as there is no need to statically implement all possible operations, and (2) improved generality as more operations can be efficiently supported.

Theoretically, runtime reconfiguration can significantly improve the hardware efficiency of a reconfigurable design, since configuration memories can dynamically update their content to keep all implemented circuits active during runtime. In practice, the use of runtime reconfiguration is limited by the overhead to switch between configurations. In addition, existing scenarios to apply runtime reconfiguration tend to be application specific. In this thesis, we aim to systematically identify general optimisation opportunities to to apply runtime reconfiguration, to integrate runtime reconfiguration support into reconfigurable designs, and eventually to exploit runtime reconfiguration to improve design performance. We divide the optimisation opportunities into three levels: circuit level, application level, and system level.

At the circuit level, runtime reconfiguration is used to support customised operators without sacrificing operator generality. Hardware designers develop general arithmetic operators, such as multipliers, adders, and ALUs, to handle arithmetic operations with variable input values. For the 32-bit unsigned multiplier shown in Figure 1.3, when some input bits are fixed (constant) during runtime, the corresponding logic gates in the multiplier become redundant. As an example, if all bits of b are '1', $a \cdot b = a$, and the multiplier can be implemented as wire connections with a. Therefore, the logic gates of this multiplier become redundant. This introduces a dilemma for hardware designers: on the one hand, implementing customised operators reduces design area, on the other hand, even ASIC designs cannot afford only supporting specific constants for an application. Runtime reconfiguration resolves this issue as

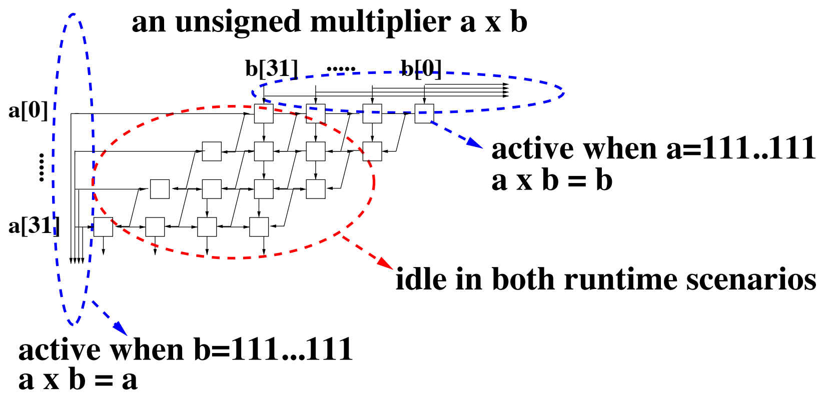


Figure 1.3: An example of idle resources at the circuit level. For a general multiplier with input a and b, there are idle resources if parts of a or b are constant. In the extreme case, if all bits of b are '1', the output is equal to a, the multiplier reduces to wire connections to $a[0] \sim a[31]$. Similarly, if all bits of a are '1', the operations can be implemented as wire connections to $b[0] \sim b[31]$. In both cases, all logic gates in the multiplier become redundant (outputs are directly to inputs).

deeply customised operators can be used for specific constants, and the implemented customised operators can be reconfigured during runtime to support different constants. Previous work [BS08, BJLW11, JBLT12a] has studied the use of runtime reconfiguration for arithmetic operators with constant input values. However, the constant operators can be further optimised. As demonstrated in Figure 1.3, the amount of redundant (idle) resources depends on constant value. A small variation in the constant value can lead to a significant reduction in resource usage. Some implemented circuits, while contributing to results, might not be needed by the target algorithm, i.e., removing the circuits will not compromise the algorithm quality. Therefore, constant coefficients in an application can be tuned to further reduce operator resource usage.

At the application level, runtime reconfiguration enables a reconfigurable design to only implement function modules that are active at the current runtime scenario. The reconfigurable design for an application often contains multiple function modules. During runtime, the function modules become idle from time to time, bounded by data dependencies or bandwidth constraints. These idle functions reduce hardware efficiency. Figure 1.4 shows an example application with four functions. The four functions are executed step by step in different runtime scenarios. For a conventional FPGA design, all the four functions are mapped into a single con-

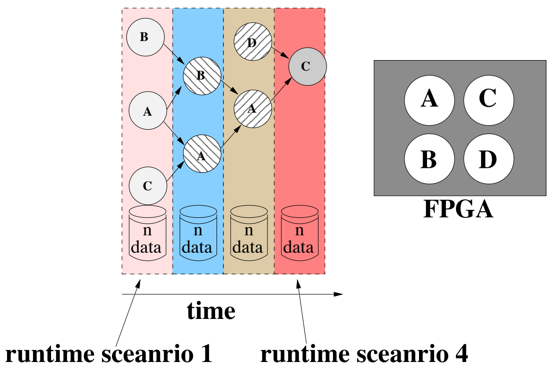


Figure 1.4: An example of idle resources at the function level. The example application contains four functions. For a static design, as the implemented functions cannot be changed during runtime, all of the functions must be implemented in the same configuration.

figuration file, and are activated based on runtime conditions. When all four functions cannot be activated concurrently all the time, this design approach introduces idle function modules.

Previous work that partitions reconfigurable designs is motivated by the fact that for some applications, not all application functions can fit into one FPGA. In this case, even when an FPGA can accommodate the whole application, the idle resources lead to inefficient use of FPGA resources. With runtime reconfiguration, the idle functions can be dynamically reconfigured into active functions.

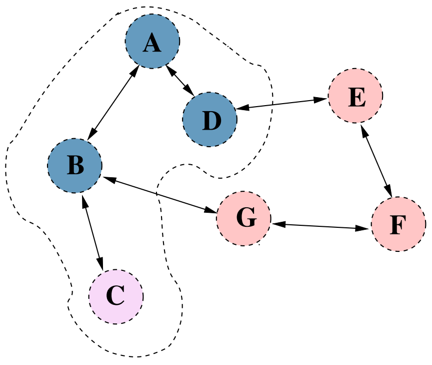


Figure 1.5: An example of idle resources at the system level. When an application is launched into a reconfigurable system, FPGA nodes A, B and D are available. During the execution of the application, node C, E, G and F become available. For a conventional FPGA application, the resources that become available during its execution time cannot be effectively utilised, leading to idle FPGA nodes.

At the system level, runtime reconfiguration adapts a reconfigurable design to the resource availability variations in a multi-FPGA system. Typically, a reconfigurable system contains multiple FPGAs, and is shared by various applications. Computing nodes in reconfigurable systems are provisioned and released by applications from time to time. At compile time, application developers are not aware of the amount of resources available to the developed designs at runtime. Figure 1.5 shows an example reconfigurable system with seven FPGAs. Three of the seven FPGAs are available when a reconfigurable design is launched in the system, and the other four FPGAs become available during the execution of the design. We name each FPGA as a computing node. With conventional design approaches, hardware developers can assume that a reconfigurable design only uses one computing node so it can start as soon as launched, or the developers can limit the design to use all the seven computing nodes. In the latter case, the design needs to wait until all the seven computing nodes are available during runtime, and will stay idle when some computing nodes are used by other applications. In both cases, idle resources occur in the system. With runtime reconfiguration, a reconfigurable design can dynamically reconfigure computing nodes in a system, to adapt to node availability variations.

1.2 Contributions

Given the runtime reconfiguration opportunities at the three design levels, we invent new design optimisation techniques and design flows for runtime reconfigurable designs. We introduce the basic design model and tool flow in Section 1.3, and elaborate the detailed design approaches in the following chapters. The summarised results in Chapter 6 show that in terms of overall throughput, the optimised runtime reconfigurable designs for Bond Option Pricing, Barrier Option Pricing, Particle Filtering, and Reverse Time Migration applications achieve up to 26 times improvements compared with the corresponding static designs, and orders of magnitudes improvements compared with software implementations. We refer to the conventional reconfigurable designs without runtime reconfiguration as static designs. The contributions of this thesis include:

Eliminating idle logic gates in arithmetic operators. (Chapter 3)

At the circuit level, we propose a design approach to (1) explore algorithm design spaces to find constant coefficient sets that are preferable to hardware implementations, (2) implement customised operators for the selected constant coefficients, and (3) integrate the customised operators into reconfigurable designs. The proposed approach uses runtime reconfiguration to switch between different customised operators when different constant coefficients need to be supported. We evaluate the efficiency of the proposed approach with two finite-difference applications. Experimental results show that compared with previous constant operators, the resource usage of the applications is further reduced by 50%. The deeply optimised arithmetic operators, when integrated into reconfigurable designs, lead to up to 7.8 times speedup over the corresponding static designs.

Eliminating idle functions in high-performance applications. (Chapter 4)
At the application level, we propose a partitioning approach for applications with idle functions. Reconfiguration Data Flow Graph (RDFG), a hierarchical graph structure, is defined. We develop design models, search algorithms and design rules to group functions active at the same time into the same configuration, based on the analysed idle cycles of each function. The grouped functions are optimised to fully exploit the resources previously used by idle functions. Applications in finance, control and seismic imaging are developed with the proposed approach. The runtime reconfigurable designs approximate the optimal hardware efficiency by eliminating idle functions, and are 1.31 to 2.19 times faster than optimised static designs. FPGA designs developed with the proposed approach are up to 26.7 times faster than optimised CPU reference designs and 1.55 times faster than optimised GPU designs.

Eliminating idle computing devices in reconfigurable systems. (Chapter 5)
At the system level, we propose an approach that optimises reconfigurable designs by constructing scalable designs. The scalable designs can adapt to the available runtime resources in a reconfigurable system. The proposed approach has two stages: compile-time optimisation and runtime scaling, and can be used in developing effective servers for high-performance computation. Two benchmark applications, Bond Option Pricing and Reverse Time Migration, are developed with the proposed approach. Experimental results show that dynamic designs can dynamically scale over computing nodes that become available during their execution. When statically optimised, the dynamic designs are 1.4 to 11.2 times faster and 1.8 to 17 times more

power efficient than reference CPU, GPU, MaxGenFD, Blue Gene/Q and Cray XK6 designs; when dynamically scaled, the hardware efficiency of the dynamic designs reaches 91%, which is 1.8 to 2.3 times higher than their static counterparts.

1.3 Overview

One of the holy grails in reconfigurable computing is runtime reconfiguration. While conventional design approaches optimise reconfigurable designs in space (i.e., exploiting reconfigurable fabrics in reconfigurable devices), runtime reconfiguration enables hardware designers to optimise reconfigurable designs in runtime. Instead of targeting general computing problems, reconfigurable designs are often customised for a specific application. This provides reconfigurable designs more optimisation opportunities since fewer problem scenarios need to be supported. However, even for a specific application, a reconfigurable design still needs to cover all runtime scenarios in this application. When a device is runtime reconfigurable, we can implement a hardware design optimised for each runtime scenario. This provides runtime reconfigurable designs more optimisation scope compared with static designs. We refer the reader to the Glossary which defines the terms used in this thesis.

Both industry and research communities seek application domains that exploit the benefits of runtime reconfiguration [HBB04, SFG06, KT11, BS08, BSPM09]. There are two obstacles in this research direction: (1) the developed runtime reconfiguration approaches are often application-specific, and cannot be applicable to other application domains, and (2) even for a specific application, the effectiveness of dynamic designs depends on application parameters (such as data size), and therefore dynamically reconfiguring designs reduces design performance in certain cases. In this thesis, we take a different direction in exploring the use of runtime reconfiguration. We start from investigating the static designs to identify the inefficient design units that can be improved by runtime reconfiguration. Once inefficient design units are found, new design approaches are proposed to replace the inefficient design units with design units customised for specific runtime scenarios. Finally, runtime overhead and benefits are modelled and evaluated to ensure the reconfigurable design with minimal execution time for a given application is executed. The executed design can be a static or a dynamic design, depending on evaluation results.

In this section, we present below an overview of this thesis, in four aspects:

- hardware efficiency to define the activeness of consumed resources in a reconfigurable design, and to indicate whether a reconfigurable design contains idle resource units.
- idle resource units to indicate the inefficient design units in a static design, divided into three design levels (circuit level, application level, and system level).
- design models to optimise idle resource units into efficient designs, with runtime benefits and overhead evaluated.
- a design flow to demonstrate how a reconfigurable design can exploit runtime reconfiguration step by step to improve design performance.

1.3.1 Hardware Efficiency

In a runtime scenario, a *well-behaved* data-path achieves the peak performance by keeping all implemented operators working in pipeline, generating one result set per clock cycle. An optimal reconfigurable design refers to the case where in each runtime scenario, all available resources are fully utilised to implement *well-behaved* data-paths. In practice, limited by conventional design approaches, hardware designers need to compromise hardware efficiency for design generality. Therefore, the measured performance of the developed design is often much lower than the peak performance. We define hardware efficiency E as the ratio between the measured design throughput TH_{mes} and the theoretical peak throughput TH_{the} of a reconfigurable design, and define throughput TH as the ratio between the number of processed arithmetic operations N_{ari}

and the overall execution time T.

$$TH_{mes} = \frac{N_{ari}}{T_{mes}} \quad TH_{the} = \frac{N_{ari}}{T_{the}} \tag{1.1}$$

$$E = \frac{TH_{mes}}{TH_{the}} \tag{1.2}$$

$$= \frac{T_{the}}{T_{mes}} \tag{1.3}$$

$E = 1$ indicates that the peak performance for a given application is achieved.

The execution time of an optimal reconfigurable design with R successive runtime scenarios can be calculated by accumulating the execution time in each runtime scenario,

$$T_{the} = \sum_{rf=1}^{R} \frac{ds_{rf}}{N_{dp,rf} \cdot f_{dp}} \tag{1.4}$$

where rf is the runtime scenario index, ds_{rf} is the number of output data sets in runtime scenario rf, $N_{dp,rf}$ is the number of *well-behaved* data-paths, and f_{dp} is the operating frequency of the data-paths. In this case, there are R runtime scenarios.

In practice, the performance of a reconfigurable design is reduced by various limitations, such as: (1) inefficient data-paths due to design approaches, such as unresolved data dependencies and unsatisfied communication bandwidth requirements, and (2) reduced design efficiency in certain runtime scenarios due to idle resource units. Therefore, we express T_{mes} as follows.

$$T_{mes} = U_{des} \cdot \sum_{rf=1}^{R} \frac{ds_{rf} \cdot U_{idle}}{N_{dp,rf} \cdot f_{dp}} \tag{1.5}$$

where U_{des} accounts for the design-related inefficiency, and U_{idle} accounts for the reduced performance due to the idle resource units. The design-related inefficiency U_{des} affects the design performance in all runtime scenarios, while the idle-resource inefficiency U_{idle} is specific to certain runtime scenarios. The objective of this thesis is to reduce the gap between TH_{the} and TH_{mes} by (1) developing *well-behaved* data-paths for a given application and (2) further optimising the reconfigurable design for each runtime scenario with the support of runtime reconfiguration.

1.3.2 Idle Resource Units

Idle resource units limit the performance of reconfigurable designs, and can be viewed as the opportunities to apply runtime reconfiguration. We define idle resource units as *circuits necessary to support a given application which can become inactive during runtime.* At different design levels, idle resource units refer to different resource types, and need to be optimised with different design approaches.

- **circuit level:** an idle resource unit refers to a logic gate in an arithmetic operator. When some input values are constant over time, parts of the logic gates become redundant. The proposed design approach removes the redundant logic gates in the runtime scenarios, thus reducing the resource usage of a data-path.

- **application level:** an idle resource unit refers to an application function in a reconfigurable design. For static designs, hardware designs need to map all application functions into reconfigurable devices, to ensure an application can be properly supported. Parts of the functions may only need to be executed in certain runtime scenarios, and therefore become idle in the other scenarios. The proposed approach replaces these idle functions with active functions. Therefore the active functions obtain more resources to exploit.

- **system level:** an idle resource unit refers to an available FPGA that is not utilised by a reconfigurable design. In a reconfigurable system with multiple FPGAs, FPGAs that are not available at the beginning of design execution can become available during certain runtime scenarios, after being realised by other reconfigurable designs. If each reconfigurable design only uses the FPGAs that are available when it is launched into a reconfigurable system, the overall system efficiency is reduced. The proposed approach dynamically reconfigures designs running in reconfigurable systems, to exploit the FPGAs that become available during runtime.

1.3.3 Design Models

With idle resources detected at various design levels, design models are developed to optimise reconfigurable designs under each runtime scenario. We divide the design models into three categories: design parameters, system resource constraints, and runtime benefits and overhead. We present the basic design models as follows, and elaborate the details at each design level.

Design parameters refer to the resource usage and execution time of a dynamic design for a runtime scenario rf. Resource usage includes on-chip logic resource usage $l(C, P)$, on-chip memory resource usage $m(C, P)$, and off-chip communication resource usage $c(C, P)$, where C indicates application characteristics, such as the number of arithmetic operations and communication patterns, and P indicates the number of replicated data-paths in a configuration. Resource constraints express the amount of available resources in a reconfigurable device / system. With the support of resource constraints, the design models ensure replicated data-paths are *well-behaved* under the current runtime scenario, i.e., $U_{idle} = 1$. Therefore, the execution time T can be calculated as the ratio between output data size ds and processing capacity $P \cdot f_{dp}$. To approximate the peak performance, the model objective is to minimise the execution time of all configurations in a reconfigurable design. For each design configuration, the model can be expressed as:

$$\textbf{minimise:} \ \frac{ds_{rf}}{P_{rf} \cdot f_{dp,rf}} \tag{1.6}$$

subject to:

$$l(C_{rf}, P_{rf}) \leq A_l \tag{1.7}$$

$$m(C_{rf}, P_{rf}) \leq A_m \tag{1.8}$$

$$c(C_{rf}, P_{rf}) \leq A_c \tag{1.9}$$

where A_l, A_m and A_c respectively represent the available logic, memory and communication resources in a reconfigurable system.

Runtime reconfiguration is a double-edged sword: it enables reconfigurable designs to be further customised for specific runtime scenarios, however, dynamically switching between different hardware designs will inevitably introduce overhead. Runtime benefits RT_{bne} refer to the reduction in the overall execution time T by dynamically updating FPGA configurations, and can be expressed as:

$$RT_{bne} = \sum_{rf=1}^{R} \frac{ds_{rf}}{P \cdot f_{dp}} - \sum_{rf=1}^{R} \left(\frac{ds_{rf}}{P'_{rf} \cdot f'_{dp}} + O_{rf} \right) \tag{1.10}$$

where $\sum_{rf=1}^{R} \frac{ds_{rf}}{P \cdot f_{dp}}$ indicates the execution time of a static design, and $\sum_{rf=1}^{R} (\frac{ds_{rf}}{P'_{rf} \cdot f'_{dp}} + O_{rf})$ indicates the execution time of a dynamic design. The design parameters of static and dynamic designs can be collected after design optimisation. O_{rf} is the reconfiguration time for switching from configuration rf to configuration $rf + 1$. The reconfiguration time includes the time to update circuit configuration information, and the time to preserve application context data (intermediate results).

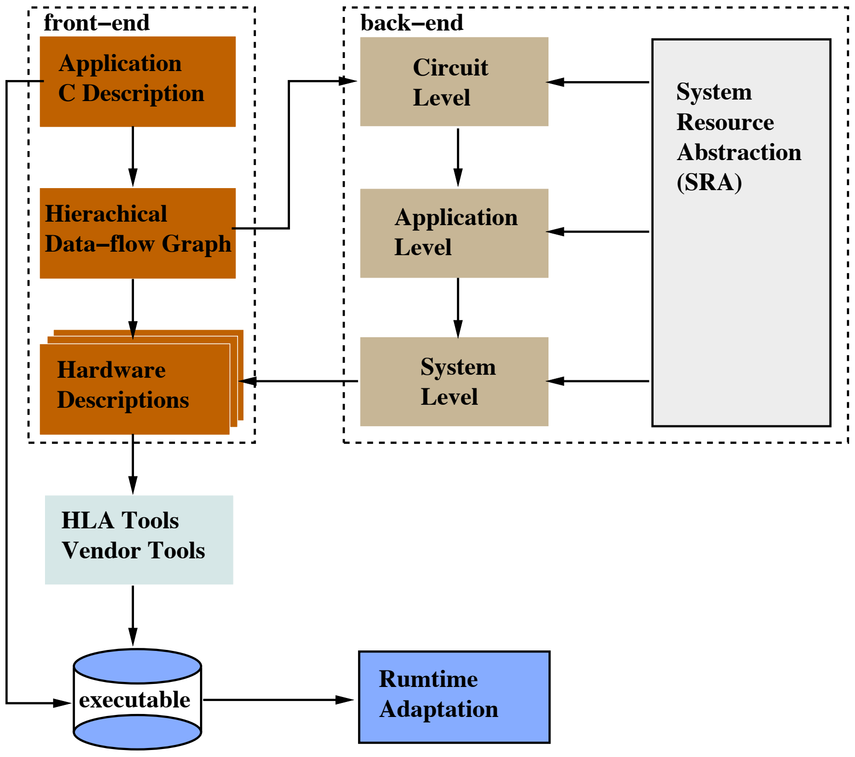


Figure 1.6: Overall design flow.

1.3.4 Design Flow

The objective of this thesis is to eliminate U_{des} and U_{idle} by optimising reconfigurable design in compile time and by dynamically reconfiguring the optimised designs. This mainly includes three steps:

- identifying the runtime reconfiguration opportunities in a reconfigurable design, i.e., finding the idle resource units in a reconfigurable design;
- modelling the impact of these runtime reconfiguration opportunities in a reconfigurable design, and optimising the design under the new optimisation opportunities;
- implementing the optimised design in FPGAs, and adapting the reconfigurable design during runtime.

Our approach starts with descriptions in the C language of an application, generates HLS-compatible hardware descriptions, and links host programs and synthesised configuration files as an executable. Figure 1.6 shows the tool flow. The front-end of the tool translates the C descriptions into high-level hardware descriptions, using a hierarchical Data-Flow Graph (DFG) as the intermediate representation of an application. In the back-end of the tool, the hierarchical DFG goes through three design levels, to exploit runtime reconfiguration opportunities step by step. We use a System Resource Abstraction (SRA) file to describe system-specific information such as available resources, inter-FPGA connections, and operator resource usage. Eventually, the back-end updates optimised design parameters after exploiting runtime reconfiguration, and the front-end uses the optimised design parameters to generate hardware descriptions.

The design flow generates a customised hardware design for each runtime scenario, with each design synthesised as a runtime configuration. We link the runtime configurations for an application as an executable. During runtime, the executable downloads the configuration that suits the current runtime scenario the most into available FPGAs, and adapts the implemented design when runtime scenarios vary. In this thesis, we build the tool with the ROSE infrastructure [Qui00]. The compiler front-end translates the computational kernels in the

C program into hardware descriptions, and the back-end implements the proposed optimisation approaches at the circuit, function and system levels. The tool can be downloaded from `http://www.doc.ic.ac.uk/~nx210/tools/irue.zip`.

1.4 Publications in Ph.D Study

Peer-reviewed Journal Publications

Chapter 3

[*J*1] **X. Niu**, Q. Jin, W. Luk and S. Weston. "A Self-Aware Tuning and Evaluation Method for Finite-Difference Applications in Reconfigurable Systems". ACM Transactions on Reconfigurable Technology and Systems (TRETS) 7(2):15(2014)

Chapter 4

[*J*2] **X. Niu**, Q. Jin, W. Luk, Q. Liu and O. Pell. "Automating elimination of idle functions by run-time reconfiguration" ACM Transactions on Reconfigurable Technology and Systems (TRETS) (accepted, to appear).

[*J*3] T.C.P. Chau, **X. Niu**, A. Eele, W. Luk, P.Y.K. Cheung and J. Maciejowski. "Extended Heterogeneous Reconfigurable System for Adaptive Particle Filters with Data Compression". ACM Transactions on Reconfigurable Technology and Systems (TRETS) (accepted, to appear).

Chapter 5

[*J*4] **X. Niu**, J.G.F Coutinho, Y Wang, and W. Luk. "EXPRESS: Exploiting Runtime Resources in Reconfigurable Systems", IEEE Design & Test (submitted).

Others

[*J*5] **X. Niu**, K.H. Tsoi, Q. Liu, A. H.T. Tse and W. Luk. "Adaptive Framework for Distributed Application in Heterogeneous Cluster With Wireless Networking", IEEE Transactions on Parallel and Distributed Systems (TPDS) (submitted).

[*J*6] Q. Liu, T. Mak, **X. Niu**, W. Luk and A. Yakovlev. "Power Adaptive Computing System Design for Solar Energy Powered Embedded Systems". IEEE Transactions on Very Large Scale Integration Systems (TVLSI) (accepted, to appear).

Peer-reviewed Conference Publications

Chapter 3

[*C*1] **X. Niu** and W. Luk. "A Dynamically Tuned Finite Difference Method For Reconfigurable Systems". 2012 Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS).

Chapter 4

[*C*2] **X. Niu**, T.C.P. Chau, Q. Jin, W. Luk and Q. Liu. "Automating elimination of idle functions by run-time reconfiguration". FCCM, pp. 97-104, 2013.

[*C*3] **X. Niu**, T.C.P. Chau, Q. Jin, W. Luk and Q. Liu. "Automating Resource Optimisation in Reconfigurable Design". FPGA, pp 275, 2013.

[*C*4] T.C.P. Chau, **X. Niu**, Alison Eele, W. Luk, Peter Y.K. Cheung and Jan Maciejowski. "Heterogeneous Reconfigurable System for Adaptive Particle Filters in Real-Time Applications". ARC, pp. 1-12, 2013.

[*C*5] **X. Niu**, Q. Jin, W. Luk and Q. Liu. "Exploiting Run-time Reconfiguration in Stencil Computation". FPL, pp. 173-180, 2012.

[*C*6] R. Cattaneo, **X. Niu**, C. Pilato, T. Becker, M.D. Santambrogio and W. Luk. "A Framework for Effective Exploitation of Partial Reconfiguration in Dataflow Computing". ReCoSoC pp. 1-8, 2013.

[*C*7] P. Grigoras, **X. Niu**, J.G.F. Coutinho and W. Luk. "Aspect Driven Compilation for Dataflow Designs". ASAP, pp 18-25, 2013.

[*R*1] K. Papadimitriou, C. Pilato, D.N. Pnevmatikatos, M.D. Santambrogio, C.B. Ciobanu, T. Todman, T. Becker, T. Davidson, **X. Niu**, G. Gaydadjiev, W. Luk, D. Stroobandt. "Novel Design Methods and a Tool Flow for Unleashing Dynamic Reconfiguration". CSE 2012: 391-

398.

Chapter 5

[C8] **X. Niu**, J.G.F. Coutinho, W. Yu and W. Luk. "Dynamic Stencil: Effective Exploitation of Run-time Resources in Reconfigurable Clusters". FPT, pp. 214-221, 2013.

[C9] **X. Niu**, J.G.F. Coutinho and W. Luk. "A Scalable Design Approach for Stencil Computation on Reconfigurable Clusters", FPL. pp. 1-4, 2013.

[C10] P. Grigoras, M. Tottenham, **X. Niu**, J.G.F. Coutinho and W. Luk. "Elastic Management of Reconfigurable Accelerators", ISPA 2014: 174-181

Others

[C11] N. Ng, N. Yoshida, **X. Niu**, K.H. Tsoi, and W. Luk. "Session Types: Towards Safe and Fast Reconfigurable Design". HEART, pp. 22-27, 2012.

[C12] **X. Niu**, K.H. Tsoi and W. Luk. "Self-Adaptive Heterogeneous Cluster with Wireless Network". RAW, pp. 306-311, 2012.

[C13] A. Bara, **X. Niu**, W. Luk. "A Dataflow System for Anomaly Detection and Analysis", FPT 2014

[C14] **X. Niu**, W. Luk, Y. Wang. "EURECA: On-Chip Configuration Generation for Effective Dynamic Data Access", FPGA 2015

Chapter 2

Background and Related Work

2.1 Introduction

This chapter presents the background information of this thesis. We divide the following parts of this chapter into five sections. Section 2.2 introduces the concepts of reconfigurable computing, data-flow programming, and runtime reconfiguration. In Section 2.3, the related work in runtime reconfiguration is introduced. We categorise the related work as: (1) scenarios previously proposed to utilise runtime reconfiguration, and (2) tools and approaches developed to apply runtime reconfiguration. Section 2.4 summarises the novel aspects in this thesis, compared with the related work, and Section 2.5 concludes this chapter.

2.2 Background

The concept of reconfigurable computing can be traced back to 1963, when a restructurable computer system was proposed to build special-purpose computers for given computational problems [EBTB63]. The term reconfigurable computing refers to the use of programmable hardware to construct customised circuits for a specific computational problem, and the term programmable hardware indicates that the information defining the customised circuits can be updated after chip fabrication to adapt the implemented circuits to problem requirements.

Modern reconfigurable devices, such as FPGAs, were initially proposed for circuit verification and rapid prototyping [CH02]. It was soon noticed that with large logic gate capacity, deeply pipelined circuits and flexible programmability, FPGAs can achieve better performance for signal processing applications [Her97], compared with software implementations. Nowadays, hardware accelerators based on FPGAs have achieved orders of magnitude improvements in absolute performance and energy efficiency for applications such as financial modelling [TL08a], DNA sequencing [ATLJ13], seismic imaging [NJL$^+$12a] and scientific computing [ZP05]. In this section, we present the background information for reconfigurable computing, with the use of runtime reconfiguration emphasised.

2.2.1 FPGA architecture

Reconfigurable devices, such as FPGAs, map hardware descriptions into underlying reconfigurable fabric. The basic operations in an application can be divided as data movement and data processing. Correspondingly, the basic components of an FPGA include I/O blocks, logic blocks and interconnect network, as shown in Figure 2.1. The I/O blocks of an FPGA connect on-chip resources to external devices, with communication infrastructures such as PCI-Express controllers and DDR3 memory controllers implemented on-chip to control data streams connected with the I/O blocks. The programmable logic blocks allow users to define logic and arithmetic operations for implementation on FPGAs, and the programmable interconnect network allows users to define the connections between the implemented operations. To improve performance, commonly used design blocks such as memory blocks and DSP operators are hardened in modern FPGAs.

Typically, the basic reconfigurable units in an FPGA include configuration memory, routing resources and logic resources [BRM02], as shown in Figure 2.2. Most of FPGAs store configuration information in Static Random-Access Memory (SRAM) cells. A 6-transistor SRAM cell uses two WL transistors to control the read and write operations, and store 1-bit configuration information in its internal inverters (see Figure 2.2(a)). FPGA designs use pass transistors and routing multiplexers to define connections between logic blocks, as shown in Figure 2.2(b). A

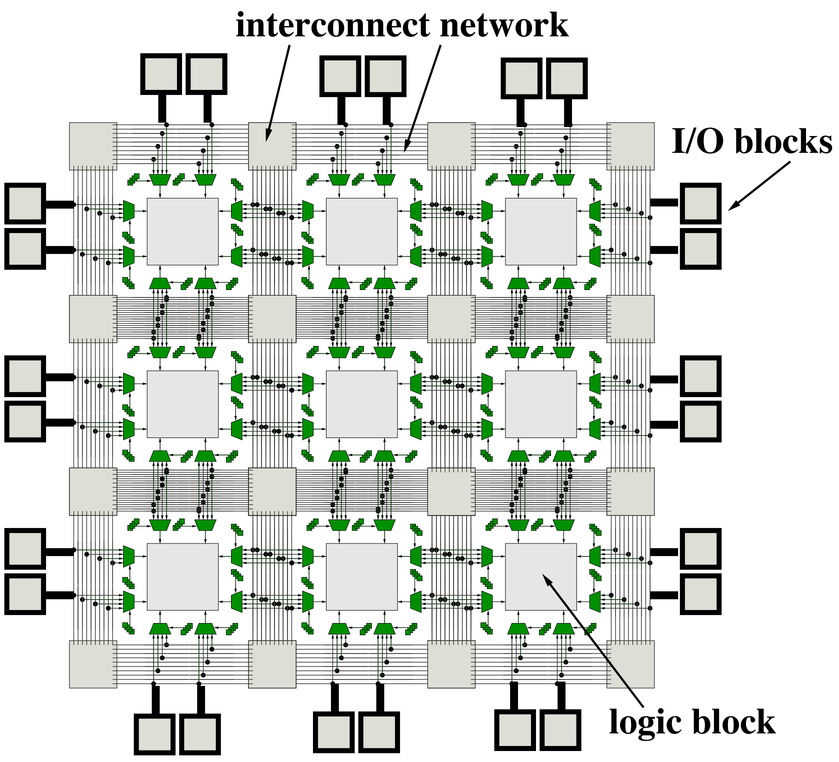


Figure 2.1: A generic FPGA architecture.

pass transistor connects its input and output nodes when its coupled SRAM cell is configured as ‘1’. A routing multiplexer connects one node to multiple possible input nodes, with the implemented connection defined by the coupled SRAM bits. Look-Up Tables (LUTs) accommodate logic operations in an FPGA design. A LUT is implemented as a multiplexer with fixed input values, with the selection signals of this multiplexer used as logic input. For the 3-input LUT in Figure 2.2(c), a 8-input multiplexer is used, and the 8 input wires are connected to pre-configured SRAM bits. The input signals for the 3-input LUT work as variable inputs of a truth table, and each combination of the variable inputs selects a specific SRAM bit to the LUT output. When the SRAM values of a LUT are updated, its implemented truth table (logic operation) is reconfigured.

A Configurable Logic Block (CLB) consists of LUTs, Flip-Flops (FFs), configuration SRAM cells, I/O ports, and a switch matrix. A CLB contains multiple LUTs (two 4-input LUTs in the CLB in Figure 2.3), and each LUT is paired with an FF to buffer its output. As shown in Figure 2.3, the switch matrix in a CLB defines the connections between CLB I/Os and the

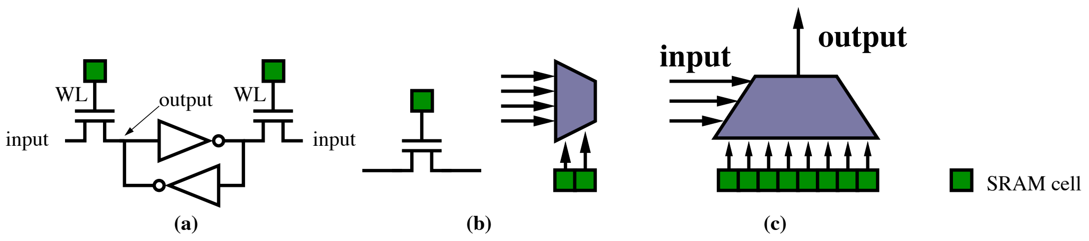


Figure 2.2: Basic units in a typical FPGA: (a) an SRAM bit, (b) a pass transistor and a routing multiplexer, and (c) a 3-input LUT.

LUT input pins. The routing multiplexers in the switch matrix are configured with SRAM cells, to determine the implemented connections inside a CLB.

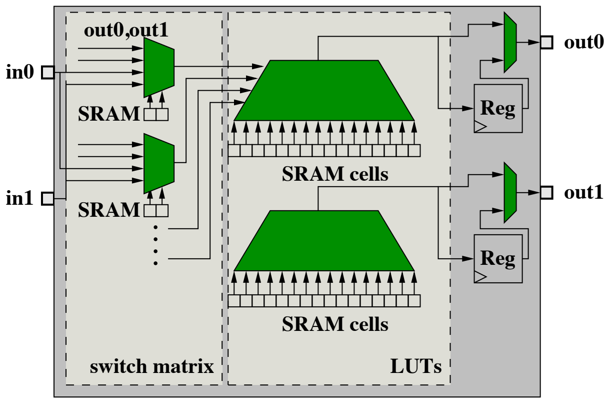


Figure 2.3: A Configurable Logic Block (CLB) of an FPGA.

The routing infrastructure of an FPGA includes Connection Blocks (CBs) and Switch Blocks (SBs). As shown in Figure 2.4, a CB defines the connections between a CLB and its surrounding wires, and an SB defines the connections between the routing wires in its neighbouring routing channels. A reconfigurable connection can be implemented by either pass transistors or multiplexers. As shown in Figure 2.4, inside a CB, multiple wires in a routing channel are multiplexed into a CLB input. The multiplexer selection inputs are connected to SRAM cells. Inside a SB, a routing wire is connected with wires in neighbouring routing channels via pass transistors. Each pass transistor is coupled with an SRAM cell. The connection is enabled when '1' is written into the SRAM cell.

During the synthesis procedure of a reconfigurable (FPGA) design, hardware descriptions for a reconfigurable design are compiled into a netlist of basic gates. The netlist is then mapped into LUTs. For an FPGA logic block that contains multiple LUTs, the mapped LUTs are clustered

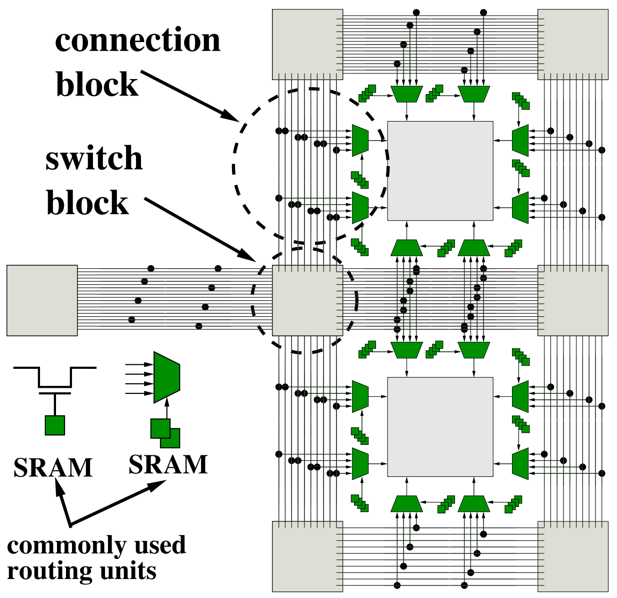


Figure 2.4: The routing infrastructure of an FPGA.

and partitioned into smaller LUT groups, with each LUT group mapped into a CLB. A netlist of logic blocks is formed. Finally, logic blocks and interconnections in the logic block netlist are placed into the available resources of an FPGA, and routed through the interconnect network shown in Figure 2.4. A generated configuration file for the synthesis procedure determines the SRAM values for switch matrices, LUTs, CBs, and SBs. For conventional design approaches, the FPGA configuration data for an application remain the same once initialised.

2.2.2 Runtime Reconfiguration

The configuration infrastructure of an FPGA consists of a configuration memory system, configuration interfaces, and configuration storage. Figure 2.5 shows the split view of an FPGA. The logic plane contains the reconfigurable elements that construct customised designs, and the configuration plane contains a configuration memory system and configuration interfaces. The configuration memory system organises the SRAM cells for logic elements and interconnect network to keep them addressable, and configuration interfaces download compiled configuration files into the memory system.

The basic unit of a configuration memory is an SRAM cell. In a reconfigurable device, its SRAM

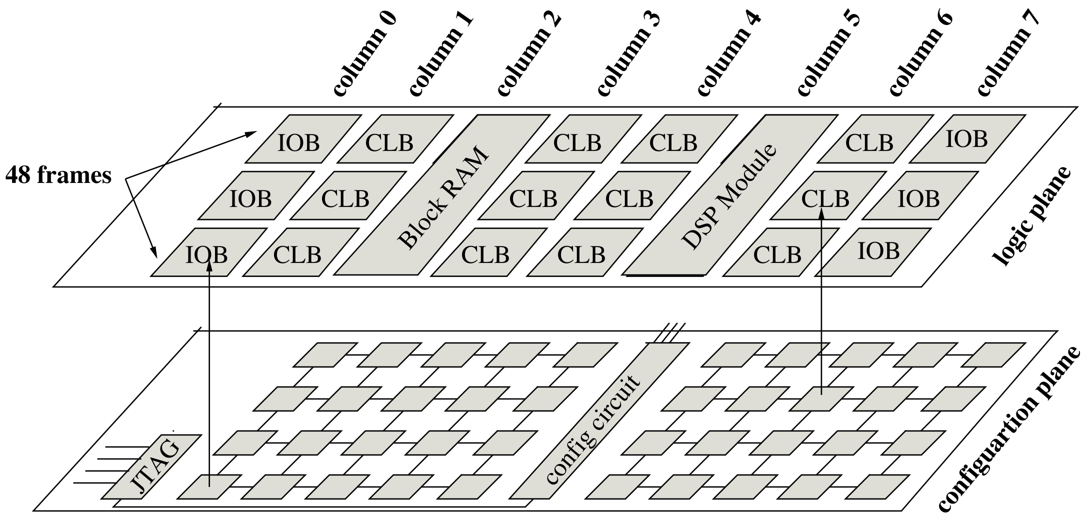


Figure 2.5: A split-level view of an FPGA.

cells are organised in columns, as shown in Figure 2.5. A configuration frame is the smallest addressable configuration unit. A configuration column consists of multiple configuration frames. In the latest Xilinx 7 series FPGAs [Xila], a configuration column contains up to 64 frame, and each frame has 3032 SRAM bits. To update SRAM values, a configuration memory system first specifies a column, and then selects a frame within the column. Configuration frames are downloaded into the SRAM arrays via configuration interfaces. The configuration interfaces shown in Figure 2.5 include Joint Test Action Group (JTAG) and other general interfaces.

Full Reconfiguration

Full Reconfiguration (FR) refers to swapping the whole configuration file for an FPGA, and therefore is compatible with conventional design approaches. To fully reconfigure an FPGA, the configuration circuits shown in Figure 2.5 are first initialised. The initialised circuits take configuration options such as configuration file size, and incrementally download configuration frames into the SRAM arrays. After a configuration file is downloaded and verified, the configuration circuits generate a done signals, and switch the FPGA from initialisation mode to user mode.

To support FR, multiple configuration files need to be prepared during the development process of a reconfigurable design. As shown in Figure 2.6(a), the prepared configuration files are stored in hard disks. During runtime, the prepared configurations are downloaded into FPGAs the same way FPGAs are initialised, and therefore FR does not need additional design steps or architecture support. As a consequence, FR introduces comparatively large configuration time.

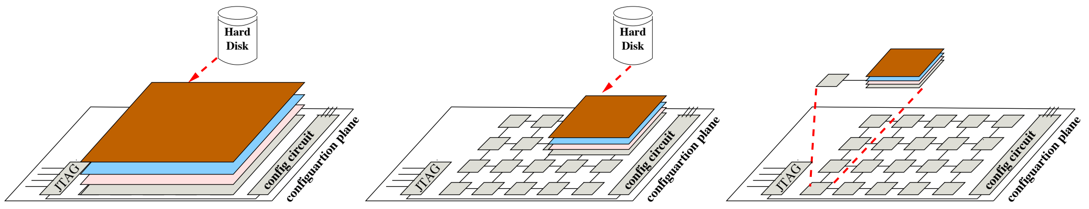


Figure 2.6: Runtime reconfiguration approaches: (a) full reconfiguration, (b) partial reconfiguration, and (c) multi-context reconfiguration.

Partial Reconfiguration

In order to reduce the reconfiguration time, Partial Reconfiguration (PR) is proposed to only update the configurations for a fraction of an FPGA, which is named as a PR region. For multiple configurations to be reconfigured during run time, a partial reconfiguration design flow extracts the differences between the configurations [Xilb]. The common parts of the configurations are used as a base configuration, with the different parts mapped as partial configuration files. During runtime, partial configuration files are updated into corresponding SRAM cells via Internal Configuration Access Port (ICAP) [Xilc], or by means of dedicated modes.

Partially reconfiguring circuits requires defining PR regions in FPGAs. As shown in Figure 2.6(b), multiple PR modules are mapped into the same PR region. This leads to three limitations in a PR design: (1) the area of a PR region is determined by the upper area bound of the mapped PR designs, which leads to area overhead when the mapped PR designs have different resource usage; (2) the logic and routing resources in a PR region are exclusive to the mapped PR designs. This increases the placement and routing complexity for other design modules, and reduces the achievable operating clock frequency; (3) hardware programmers need to do manual floorplanning to specify the PR regions. This limits the productivity for PR designs. Moreover, although partially reconfiguring FPGAs reduces reconfiguration time compared with full reconfiguration, the reconfiguration time for PR designs is still unacceptable for designs that need frequent reconfiguration (e.g. every clock cycle). Given the smallest addressable configuration size (3232 bits) and maximum reconfiguration throughput (400 MB/s) in the latest devices [Xila, Xilc], the minimum reconfiguration time is 1.01 μs (151 clock cycles for the 150 MHz operating frequency). If the operations in a reconfigurable design need to be reconfigured every clock cycle, between two consecutive reconfiguration operations, it takes 151 cycles to reconfigure circuits, and takes 1 cycle to process data. The hardware efficiency in this example drops to almost 0.

Multi-context Reconfiguration

To further reduce reconfiguration time, Dynamically Programmable Gate Array (DPGA) [DeH96] and time-multiplexed FPGAs [TCJW97] are proposed. In these architectures, configuration memories for reconfigurable logic are replicated to store multiple configurations on-chip. As shown in Figure 2.6(c), a reconfigurable unit is coupled with multiple configuration memory sets, with each set storing one possible runtime configuration. Therefore, design configurations can be updated within a cycle. During compile time, application operations are decomposed into multiple operations sets, with operations in the same set executed at the same time. Each operation set is synthesised as a configuration file, which is stored in one of the replicated configuration memories before execution. The stored configuration data are selected during runtime, to configure different circuits from time to time.

The use of multi-context reconfiguration technique is limited by its compatibility and area overhead. Since application operations are divided into frequently reconfigured operation sets, most of the previous design techniques and tools cannot be applied to multi-context designs. New development and synthesis tools need to be developed. More importantly, replicating configuration memories on-chip introduces large area overhead, given the fact that configuration memories are already extensively used in conventional FPGAs. The additional memory area is fixed once FPGAs are fabricated. Static designs are implemented with the same area overhead, although only one of the replicated configuration memories is required.

Comparison of Runtime Reconfiguration Techniques

We compare the three reconfiguration approaches in Table 2.1, in terms of compatibility with existing techniques, reconfiguration time, and area overhead.

- Fully reconfigured designs have no area overhead as each configuration file corresponds to a

Table 2.1: Comparison of reconfiguration techniques.

configuration approach	compatibility	reconfiguration time	area overhead [1]
full	high	high (0.8 s [2])	none
partial	medium	medium (50 ms $\sim$ 1.01 μs [3])	medium
multi-context	low	low (1 clock cycle)	high

[1] area overhead refers to the increase in the area usage of a reconfigurable design.
[2] measured on a design consuming 71% of a Virtex-6 SX475T FPGA.
[3] 1.01 μs calculated with peak reconfiguration throughput and minimum configuration frame size, and 50 ms measured for configuring a clock region of a Virtex-6 SX475T FPGA.

separated hardware design, which is developed and synthesised independently. However, swapping the whole configuration file during runtime introduces large reconfiguration time. For a large scale FPGA Virtex-6 SX475T, a full reconfiguration operation takes around 0.8 second to finish.

- For PR regions, only updating the configurations for PR modules reduces reconfiguration time (50 ms if a clock region is to be reconfigured, and 1.01 μ if a frame is to be reconfigured). However, a PR design contains PR modules and static design modules in the same hardware design. As discussed earlier in this section, this leads to compromised compatibility and increased area overhead.

- Multi-context designs push reconfiguration time into 1 clock cycle by storing possible configuration files on-chip. As a consequence, new design approaches need to be developed, and the replicated configuration memories introduce large area overhead. In this thesis, we focus on FR techniques to keep the proposed approaches applicable to existing tools and systems. The use of PR and multi-context techniques are discussed in Chapter 6.

2.2.3 Data-Flow Programming Model

Programming Model

Traditionally, computer programs are modelled as a series of instructions executed in a specific order. Following the von Neumann (control-flow) programming model [vN93], a hardware architecture only executes instructions when its program counter reaches these instructions.

In contrast, computer programs that follow the data-flow programming model execute (or fire [AC86]) program instructions when the input data of an instruction become available. Compared to a control-flow program, a data-flow program can execute multiple instructions at the same time, possibly out of order. Figure 2.7 demonstrates the execution flow for a simple program in its control-flow and data-flow equivalents. The control-flow program takes three cycles to finish, assuming each operation takes one clock cycle. For the data-flow program, the input data for both the adder and the divider are ready at the first cycle, and therefore the two operations are executed in parallel, finishing the program in two clock cycles. It is clear that a data-flow program can better exploit the instruction-level parallelism than its control-flow counterpart. In addition, if the same instructions are executed repetitively over multiple data sets (e.g. a for loop), the processing of the second data set can start before the first data set is finished. In Figure 2.7(c), the adder and the divider start processing the second input data set at the second clock cycle, before the fist output data set is generated. This is known as *pipelined data-flow* [GP89], and a data-flow program that generates one set of output data for each input data set is said to be *well-behaved* [DM74].

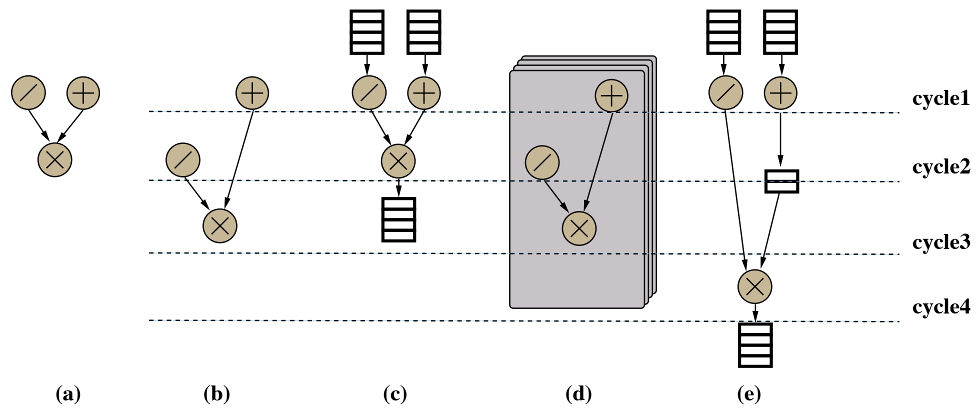


Figure 2.7: An example data-flow graph with three arithmetic nodes, following the (a) control-flow model, (b) data flow model and (c) pipelined data-flow model. (d) A vectorised hardware architecture that processes multiple data concurrently. (e) A hardware implementation of a well-behaved data-path, which generates one set of output data per clock cycle.

Vector Computer Architecture

In addition to instruction-level parallelism exploited by data-flow programming models, Single instruction, Multiple Data streams (SIMD) and Multiple Instruction, Multiple Data streams

(MIMD) architectures were proposed [Fly72] to exploit data-level parallelism, as shown in Figure 2.7(d). Instruction set extensions — such as MMX [PWW97] and Streaming SIMD Extensions (SSE) [SB01] from Intel — provide SIMD parallelism to execute the same instruction on multiple data concurrently. Recently, vector processing architectures [YSR08, NFMM13] have been proposed as soft processors implemented in FPGAs, to accelerate computationally intensive applications. With the massive data-level parallelism in graphics applications, Graphics Processing Units (GPUs) contain a large number of streaming cores (e.g., Tesla C2070 from Nvidia contains 448 streaming cores) to provide high application throughput. As more and more applications, such as matrix processing and deep learning algorithms, expose data-level parallelism, it is becoming increasingly common to use General Purpose Graphics Processing Units (GPGPUs) as hardware accelerators. Compared with CPU designs, large improvements in performance and power efficiency have been shown for matrix processing [YPS11] and seismic imaging [PF10a]. While well suited for applications with heavy data-level parallelism, vector architectures achieve relatively low performance for flow-control-heavy tasks. As an example, for parallel threads in GPUs that apply the same instruction on different data, computational efficiency reduces each time the instructions hit branch operations: even if there is only one thread that enters a certain branch, the remaining threads need to wait for this one thread to finish its branch to continue.

Data-Flow Hardware Architecture

While effective in theory, the data-flow programming model is difficult to implement in practice mainly because two assumptions: (1) the model assumes unlimited data storage between two graph nodes (unlimited First In, First Out (FIFO) queues), and (2) the model assumes unlimited hardware resources to executed any number of instructions in parallel. Static [DM74] and dynamic [AN87] data-flow architectures are proposed to execute instructions out of order. However, due to limited hardware resources and complex data dependencies, the proposed architectures sacrifice design performance to ensure architecture generality.

Reconfigurable devices provide another direction to implement data-flow programming models. Instead of building general-purpose architectures that execute instructions out of order, customised data-paths are developed application-by-application, and are mapped into reconfigurable fabrics. Figure 2.7(e) presents a customised data-path for the example program. Each node in the program data-flow graph is implemented as a customised arithmetic unit, and arcs between the nodes are implemented as wire connections and FIFOs. Instruction-level parallelism is exploited since all nodes that are fireable at the same time are executed in parallel, with the readiness of input data handled by FIFOs. As an example, if we assume an addition and an division respectively take 1 and 3 cycles, a FIFO with 2-data depth is inserted between the adder output and the multiplier input to ensure input data are ready at the same time. In addition, by streaming input data cycle by cycle, *pipelined data-flow* is supported, i.e., the data-path starts processing the second input data sets before the first data set is finished, generating one output set per clock cycle per data-path (*well-behaved*). There are two levels of parallelism in the customised data-paths. A *well-behaved* data-path fully exploits the instruction-level parallelism within a program: all implemented arithmetic nodes are busy during runtime. For the example program in Figure 2.7(a), at each clock cycle, all three arithmetic operators are processing data in parallel. On top of the instruction-level parallelism, multiple data-paths can be implemented in an FPGA, generating multiple output sets per clock cycle.

2.3 Related Work

2.3.1 Runtime Reconfigurable Applications

In the last decade, both industry and research communities have been seeking applications that can benefit from runtime reconfiguration. Recent progress enables runtime reconfigurable designs to be applied to the field of networking, control, Software Defined Radio (SDR), signal processing, and data management.

Runtime reconfiguration opportunities for networking applications stem from complex runtime conditions. An I/O crossbar is a commonly used switching module in networking applications. All to all connections are provided between the inputs and the outputs of a crossbar, with the active connections defined by runtime data values. In [Y^+03], routing multiplexers are used to compose a crossbar. When runtime conditions change, a new partial configuration file is downloaded into the configuration memory for the routing multiplexers, to update the crossbar connections. Compared with static designs that map the whole crossbar into reconfigurable elements, the runtime reconfigurable design achieves higher flexibility. For coarse-grained modules, networking modules with various functionality are dynamically configured into FPGAs to share the silicon area [LNTT01a].

In the fields of control, SDR, and data management, runtime reconfiguration is introduced to map various design modules into the same reconfigurable region. The control functions of automotive applications can be mapped into partially reconfigurable regions [HBB04]. Using an ICAP interface with 66 Mbyte/s, a control function can be reconfigured within 2 ms. In an SDR platform implemented in a Xilinx Spartan3-200 FPGA [SFG06], both 802.11 and Zigbee receivers reside in the same FPGA to support different communication protocols. It is proposed that runtime reconfiguration can be used to map the two receivers into the same reconfigurable region. For large-scale data sorting, a reconfigurable design [KT11] divides its sorting and merging parses into two configurations, and uses full reconfiguration to switch between the two design phases.

For signal processing applications, besides using runtime reconfiguration to swap possible modules [BSPM09], runtime reconfigurable designs can exploit the constant coefficients within signal processing components. In [BS08], tunable LUTs are proposed to generate customised LUT-based operators for arithmetic operations with slowly varying coefficients. The arithmetic operators are implemented as parametrisable designs. In a parametrisable design, a basic static design is placed and routed in FPGAs, and variations in coefficients are supported by changing LUT configurations. Compared with static designs that implement the arithmetic operations as general operators, the design based on tunable LUTs is 2.3 times smaller for an Finite Impulse Response (FIR) filter implementation.

Although many applications have exploited runtime reconfiguration, the improvements in system performance are limited by reconfiguration overhead and design generality. The reconfiguration overhead refers to reductions in design performance, such as the reduction in clock frequency due to floorplanning constraints, and the increase in execution time due to reconfiguration operations. Design generality indicates the range of applications to which the proposed approaches can be applied. For the previous applications, the techniques to apply runtime reconfiguration are often application-specific. In this thesis, we focus on general design approaches to exploit runtime reconfiguration techniques, with the reconfiguration overhead properly handled.

2.3.2 Optimisation Opportunities to Apply Runtime Reconfiguration

Runtime reconfigurable designs can be categorised into optimisation opportunities where specific runtime reconfiguration techniques can be applied. We summarise three existing optimisation opportunities that runtime reconfiguration can be beneficial.

Reconfigurable module opportunities refer to the use of runtime reconfiguration to swap design modules during runtime. Partial reconfiguration is typically used in this case. For a reconfigurable module, the candidate functions are compiled into partial configuration files. One of the configuration files is initialised into the reconfigurable module before execution. During runtime, the remaining files are reloaded into the module based on runtime conditions. Therefore, the same reconfigurable area can be reused to provide various functionality during run time. However, partially reconfigurable modules suffer the limitations discussed in Section 2.2.2, which include reduced clock frequency, area overhead in PR regions, and reduced productivity. For real-world applications, reconfigurable modules are developed to support multiple communication or control protocols in the same design, with the candidate protocols multiplexed during runtime. Previous designs with reconfigurable modules include a packet processing platform with various processing modules [LNTT01b], an SDR design with multiple waveform modules, and automotive and robotic designs with different control modules.

Design tuning opportunities refer to the use of runtime reconfiguration in semi-dynamic applications, where design properties are occasionally updated. Constant coefficients in FIR filters [BS08] and option pricing [BJLW11, JBLT12a] are utilised to construct constant-specific operators. When coefficients are updated during execution, variations in customised operators are updated with runtime reconfiguration. The customised operators consume fewer resources and operate at higher frequency, compared with general-purpose operators. Resource usage for FIR filters and finite-difference computational kernels is reduced respectively by 36% and 22%.

Runtime reconfiguration is required when a target application would not fit into available resources all at once. The application is partitioned into subprograms, which are sequentially reconfigured into the available resources. In temporal partitioning [PB99], target applications are partitioned into multiple configurations. The configurations are swapped in and out of reconfigurable fabrics in a specific sequence to implement the application functionality. Application tasks are represented using Data Flow Graphs (DFGs), and partitioned under resource constraints. The problem is formulated as an Integer Non-linear Programming (INLP) model [KV98] to minimise communication operations between partitioned segments. Spatial partitioning is covered in [HLH$^+$98] to support multiple devices. The temporal and spatial partitioning approaches are applicable to applications which cannot be accommodated by available resources. As Moore's Law continues, logic capacity in recent FPGAs has increased to a level where lots of applications can be accommodated without being dynamically reconfigured. Area constraints in the temporal and spatial partitioning methods will still be satisfied even when all operations are partitioned into the same configuration. However, as discussed later in this thesis, even when there are sufficient resources to implement an application, grouping all application functions into the same configuration does not necessarily provide the optimal solution.

2.3.3 Design Approaches for Runtime Reconfiguration

Previous design approaches and tools improve the performance of runtime reconfigurable systems in terms of reconfiguration time and execution time. Reconfiguration time refers to the time consumed to download a new configuration file, either partial or full, into the target FPGA. Therefore, the reconfiguration time depends on the size of the downloaded configuration file and the throughput of the used reconfiguration interface. For FR designs, the reconfiguration time also includes the time to preserve application context data when on-chip memory controllers are reconfigured. The performance of a runtime reconfigurable design is severely limited by the reconfiguration time. As shown in Figure 2.5, configuration SRAM cells are organised in columns, where each column contains multiple configuration frames. In [HK12], the partial configuration file for a crossbar is compressed by only updating specific frames in a configuration column. Moreover, the proposed approach in [HK12] improves configuration throughput by placing reconfigurable blocks into the least number of columns. The reconfiguration flow in [FBS13] only updates the differences in routing configurations. A 2 times reductions in reconfiguration time is achieved. A different direction to reduce the reconfiguration time is to reduce the number of reconfiguration operations. A partition approach is proposed in [HMZB12], where functions activated at different time intervals are combined into the same reconfigurable module. Under the same resource constraints, grouping functions activated at different time intervals reduces the number of reconfiguration operations, thus reducing the overall reconfiguration time. The proposed approach saves up to 70% of the overall reconfiguration time.

The execution time of a runtime reconfigurable design is determined by its reconfiguration time, as well as whether the optimal configuration is generated and selected. Various scheduling approaches have been proposed to schedule the prepared configurations during runtime. The SCORE project [CDW01] abstracts reconfigurable programmes as fixed-size compute pages, which are swapped into reconfigurable resources during runtime. Page schedulers are developed to make reconfiguration decisions that minimise execution time or data buffers. In multi-thread system, multiple reconfiguration candidates exist at each reconfiguration interval. A knapsack-based scheduler is proposed in [FC05] to select the configurations (design kernels) with maximum speedup. The scheduler is further improved in [FC08] by adaptively adjusting reconfiguration intervals, which reduces the overall scheduling overhead by 85%.

2.3.4 Tools for Runtime Reconfiguration

Development tools and runtime support for runtime reconfiguration aim to reduce the effort for designing and managing runtime reconfigurable designs. If reconfigurable designs can be developed and executed in conventional design environment, reconfigurable devices can be used in mainstream systems. The work to support such integration can be divided into three categories: operating system support, design automation, and runtime scheduling.

Operating system support relies on operating systems to instantiate and reconfigure design modules. Egret [WB04] is proposed as a modular platform based on uCLinux. Egret maps System-on-Chip (SoC) modules onto reconfigurable devices with runtime reconfiguration, through ICAP via standard UNIX commands. The Linux system presents devices as files under the /dev directory. Therefore it is easy to send configuration files (bitstreams) to reconfigurable devices connected to the operating system, by using commands to write to /dev files. From an operating system perspective, the configuration files can be stored in local storage, generated by other programs (configuration file compression, runtime configuration modification) or stored in remote devices (configuration server). BORPH [SB08] is an extended Linux operating system that manages and executes hardware designs as normal UNIX processes, which gain access to Operating System (OS) services. Various OS extensions such as Virtual Hardware Operating System [Bre96], ReConfigME [WKW02], ReconOS [SWP04], and Hthreads [PAA+06] have been proposed for reconfigurable systems.

Design automation tools provide Graphical User Interfaces (GUIs) or automatic support for users to develop runtime reconfigurable designs with improved productivity. Xilinx PlanAhead [Xild] integrates partial reconfiguration support with additional design steps to define runtime reconfigurable modules and runtime reconfigurable regions. GoAhead [KTB+12] provides a GUI and command script interface for floorplanning and macro placement of runtime reconfigurable regions. In order to further reduce the design effort to place and floorplan runtime reconfigurable regions, automatic placement and floorplanning approaches have been proposed [HLM+13].

Runtime scheduling tools aim to select suitable reconfigurable designs during runtime, and to reduce reconfiguration overhead. [FC05] evaluates three scheduling algorithms to allocate portions of the reconfigurable hardware at runtime, in order to select the configuration with maximum speedup. On the other hand, reconfiguration time can be reduced with various scheduling approaches. [BBD05] exploits configuration prefetching to reduce reconfiguration overhead. Since configuration information is downloaded into reconfigurable devices column by column, similarities in configurations for used LUTs [HK12] and routing [FBS13] are explored to reduce the number of columns of reconfigurable devices that need to be reconfigured, i.e., reduce the size of configuration files.

2.4 Comparison to the Related Work

This thesis aims at providing a systematic approach to exploit runtime reconfiguration to improve the performance of reconfigurable designs, in terms of throughput and power efficiency. Compared with previous work, this thesis introduces new optimisation opportunities, proposes new design techniques, and benefits various application domains.

2.4.1 Optimisation Opportunities

The new optimisation opportunities are motivated by idle resource units in reconfigurable hardware designs. The idle resource units refer to reconfigurable units that do not always contribute to valid result generation during runtime. At different levels of a reconfigurable design, the idle resource units are introduced by different design limitations, which are categorised into different optimisation opportunities to apply runtime reconfiguration.

At the circuit level, arithmetic operations with constant inputs are mapped as general arithmetic operators, to cope with various constant inputs. A general arithmetic operator is developed to handle all input value combinations. In the case that one of operator inputs is constant during runtime, parts of the arithmetic logic gates become redundant (idle). Previous work [BJLW11] focuses on how to generate constant operators for specific constant operations. We identify new optimisation opportunities to tune algorithms such that the tuned algorithm makes use of constant coefficients that are preferable to hardware implementations.

At the function level, application functions become idle during runtime, due to data dependencies and resource constraints. Instead of partitioning applications when the target applications do not fit into FPGAs, the function-level approach extracts functions active at the same time, groups the functions into a configuration, and replicates the grouped functions to utilise the resources previously consumed by idle functions. There are new optimisation opportunities for generating runtime reconfigurable designs that design functions are only implemented when they are active.

At the system level, an FPGA device is considered as a computing node. During runtime, the computing nodes in a reconfigurable system are shared by various applications, and the availability of a computing node depends on indeterministic user behaviours. When executed in such complex runtime scenarios, static designs lead to idle computing nodes. The use of runtime reconfiguration enables dynamic designs to use additional computing nodes that are initially unavailable to the dynamic design but become available during design execution. The optimisation opportunities at the system level enable dynamically reconfiguring hardware designs to adapt to resource availability variations.

2.4.2 Design Objective and Design Optimisation Techniques

Runtime reconfiguration improves the performance of reconfigurable designs as it creates wider design space to explore. Associated with the new optimisation opportunities, new design spaces are created at the three design levels. Throughout this thesis, the design objective is to minimise the execution time of reconfigurable designs by eliminating idle resource units at each design level.

At the circuit level, in each runtime scenario, the resource usage of a reconfigurable design depends on the value of constant coefficients. In order to further customise the constant arithmetic operators, we propose design approaches to build an algorithm design space where each constant set in the design space is mathematically equivalent, i.e. each constant coefficient set can be implemented as customised operators that generate correct results for the algorithm. We develop design models to capture the constant coefficient set that leads to minimal resource usage. Compared with previous work, the design optimisation process further reduces resource usage of the constant operations.

At the function level, an application function becomes idle when its input data has not been generated, or the input data do not arrive in time due to bandwidth constraints. In previous partitioning work, the design approaches group functions into a configuration as long as there are enough resources in an FPGA. We develop an application analysis approach to identify application functions that can work at the same time, and introduce a development flow to generate runtime reconfigurable designs with maximum design throughput.

At the system level, the proposed approach aims at exploiting the FPGA nodes that become available during runtime. For a conventional multi-FPGA design, the design configuration cannot be changed during execution. The number of used FPGAs is fixed at compile time. For a dynamic design, the design challenges include how to ensure correct functionality and how to improve performance, when new FPGA nodes are reconfigured into the current application. We propose a two-step design approach at the system level. (1) Compile-time optimisation involves customising a reconfigurable design for heterogeneous FPGAs in a reconfigurable system, and (2) a runtime scaling process schedules reconfiguration and communication operations, when a runtime reconfigurable design scales over new FPGA nodes.

2.4.3 Application Domains

Focusing on idle resources at each design level, the design approaches invented in this thesis are not application-specific. A reconfigurable design can benefit from runtime reconfiguration as long as there are idle resources in the design. At each design level, we evaluate the proposed approaches with various applications from one or multiple application domains. At the circuit level, the proposed approaches are applicable to applications with tunable constant coefficients.

Finite-difference applications are supported, and two benchmark applications from financial pricing and seismic imaging are evaluated. At the function level, the proposed approach can be applied as long as there are idle functions during runtime. Applications from the fields of control, finance and geophysics are evaluated. The system-level approach is applicable to all multi-FPGA designs, when they are implemented in a reconfigurable system with unknown resource availability. To demonstrate the generality of the system-level approach, applications with no communication operations and with intensive communication operations are evaluated. The previous work for the related application domains is introduced and compared in the following chapters.

2.4.4 Novel Aspects Compared with Related Work

To summarise, while runtime reconfiguration techniques show the potential to improve the performance of reconfigurable designs, current techniques have two major limitations as follows.

1 Existing work involving runtime reconfiguration is often driven by application requirements. For example, for applications such as FIR filtering [BS08], SDR [HBB04], sorting [KT11], the proposed designs approaches are specific to the target applications, and are therefore difficult to be generalised for a wider range of applications.

2 Current runtime reconfiguration techniques often do not lead to direct performance improvements for the target applications. Dynamically reconfiguring design modules bring more flexible integration of designs modules [WB04] and more functionality supported by an application [BSPM09]. However, there is a large design space to explore for runtime reconfigurable designs. The lack of proper design models leads to inefficient designs (i.e. configuration files). The design efficiency achieved by various scheduling approaches [FC05, FC08, HMZB12] is often limited. For example, in [HMZB12], grouping functions active at different time reduce the number of required reconfiguration operations. However, instantiating idle functions in configurations reduces the configuration performance (i.e. longer execution time). As application data size increases, the increase

in configuration execution time will outweigh the reduction in reconfiguration time. The runtime solution proposed in [HMZB12] then becomes inefficient.

This work aims to propose general design models, approaches, and tools that can be applicable to as many applications as possible that can benefit from runtime reconfiguration, and to bridge the performance gap between ASIC and reconfigurable designs. Compared with the state of the art in runtime reconfiguration, the novel aspects of this work include:

1 Rather than focusing on application-specific requirements, we generalise the opportunities in applications to remove idle resource units in reconfigurable designs. Based on this idea, the opportunities to apply runtime reconfiguration can be divided into three categories: at circuit level, where an idle resource unit refers to a redundant logic gate; at function level, where an idle resource unit refers to a inactive function module; and at system level, where an idle resource unit refers to an unused reconfigurable device.

2 Starting from the idle resource units, the design models and design approaches aim at exploiting the additional resources from eliminating the idle resource units to improve application performance. We extract the general properties of reconfigurable designs and runtime reconfiguration operations and capture them in our performance models and design models, and extend the model to exploit domain-specific optimisation opportunities. This enables the proposed approaches to explore the large design space of a reconfigurable design, after idle resource units are eliminated by runtime reconfiguration.

3 At each design level, various applications from different application domains are evaluated to verify the generality of the proposed approaches. Automatic designs tools are developed, with tool front-end compiling high-level descriptions into hardware designs, and tool back-end integrating the design approaches proposed in this thesis (the circuit-level approach is currently not integrated due to language incompatibility: the generated arithmetic operators are described in VHDL, while current tool front-end only takes MaxCompiler Java descriptions [Tec]).

2.5 Summary

This chapter introduces the background information for runtime reconfiguration, and discusses the related work in the field of runtime reconfiguration. We compare our work with the related work, and summarise the novel aspects of this thesis.

Chapter 3

Circuit-Level Optimisation for Runtime Reconfigurable Designs

3.1 Introduction

This chapter discusses the use of runtime reconfiguration at the circuit level. We define constant operators as the arithmetic operators with some of their input bits being constant during certain periods of runtime (runtime scenarios). For the runtime scenarios with constant operators, customised operators are developed, and dynamically reconfigured when the constant operators are no longer needed. This has been studied by various researchers [BJLW11, JBLT12b, BS08]. In this chapter, we take the circuit-level optimisation one step further. For the same algorithm, there are many constant coefficients which can satisfy the algorithm requirements. In other words, while having different values, these constant coefficients are mathematically equivalent from the algorithm perspective. Therefore, even for a deeply optimised arithmetic operator, there can still be redundant logic gates since certain bits in the coefficients may have negligible contributions to the algorithm output. We study the impacts of runtime reconfiguration for this case.

Outline. Section 3.2 provides an overview of the circuit-level optimisation approach. Section 3.3 defines the idle resource units at the circuit level, and builds the design space for constant operators. Section 3.4 presents the optimisation techniques for constant operators, and Section 3.5 proposes the design models to integrate the optimised constant operators into reconfigurable designs. A runtime evaluator is developed in Section 3.6 to ensure the design configuration with minimum execution time is executed. Section 3.7 introduces two benchmark applications, and Section 3.8 evaluates the efficiency of the proposed approach with the benchmark applications. Section 3.10 discusses the limitations and the potential improvements for the current work, and Section 3.11 concludes this chapter.

3.2 Approach Overview

This section demonstrates the basic idea of this chapter with a motivating example, presents the overall design flow of the proposed method, and briefly introduces finite-difference algorithms and corresponding hardware implementations,

3.2.1 Motivating Example

In a data-path following the data-flow programming model, as discussed in Section 2.2.3, arithmetic operations are implemented as independent operators that work concurrently. A constant operator can be mapped into FPGAs as a general operator or as a customised operator. A general operator refers to an arithmetic operator, such as an adder, with all input bits being runtime variables. Figure 3.1(a) shows a general multiplier with two 32-bit unsigned inputs. A customised operator refers to an operator without redundant logic gates introduced by constant inputs. A customised operator can be implemented as a static operator or a dynamic operator. For a static operator, padded '0's in the constant input, along with the coupled logic gates, are removed from the general operator, as shown in Figure 3.1(b). Since the padded '0's do not contribute to computational results, the static operator has the same precision as a general operator. For a dynamic operator, as shown in Figure 3.1(c), logic gates are reorganised for a specific constant value to minimise operator resource usage. The same computational precision is preserved for the specific constant value.

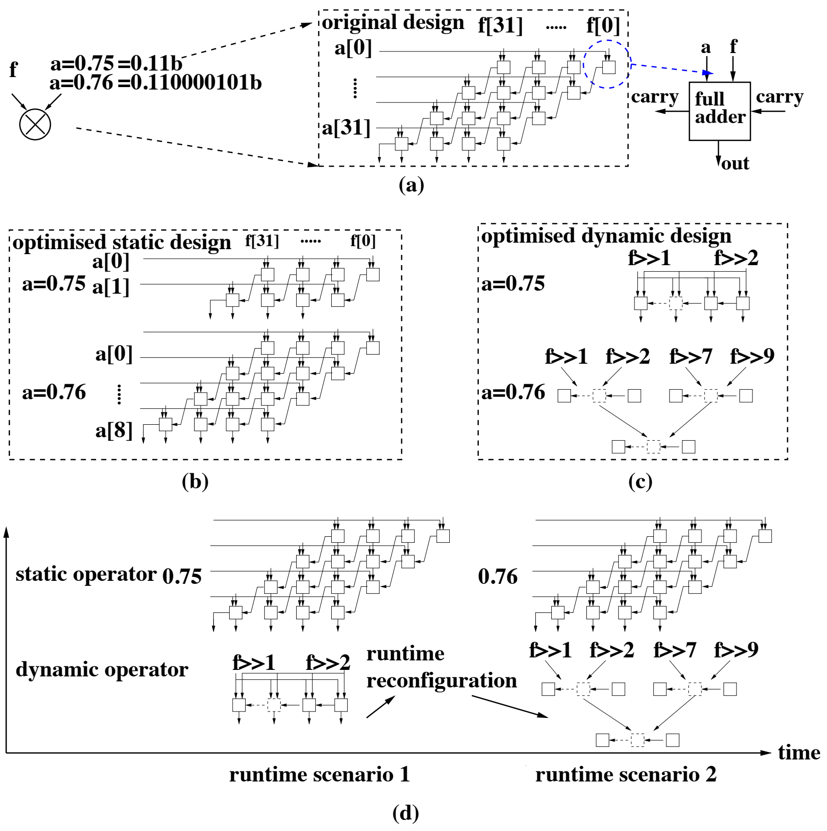


Figure 3.1: Hardware implementations for a constant operator. (a) A multiplier with constant coefficient is implemented as a general adder matrix. (b) A static operator following general multiplier design covers the upper bound of constant coefficient width to ensure that the operation does not need to be reconfigured during runtime. In this example constant coefficients 0.75 and 0.76 are respectively represented with 2 and 9 bits, therefore we use a 9-bit*32-bit multiplier, assuming the other input is 32-bit. (c) A dynamic operator is deeply optimised for a target constant, therefore a operator must be reconfigured to support a different constant. (d) The execution process of a static operator and a dynamic operator during runtime. To switch between two scenarios with two different constant coefficients, the static operator only needs to update the coefficient stored in registers, while the dynamic operator needs to be reconfigured.

The resource usage of customised operators depends on constant values. In this example, we show a multiplier a constant coefficient 0.75. As shown in Figure 3.1(b), 0.75 is represented with 2 bits in fixed-point format. The following 30 '0' bits are redundant as any input data multiplied with them will generate the same result. For the static operator, the redundant logic gates connected to the 30 bits are removed. For a dynamic operator, $0.75 \cdot f$ can be implemented as $0.5 \cdot f + 0.25 \cdot f$, where $0.5 \cdot f$ and $0.25 \cdot f$ are implemented as a 2-bit right shifter and a 1-bit right shifter, respectively. Therefore, it takes the dynamic operator two shifter and one adder to implement the same operation. While a dynamic operator consumes less resources, the customised operators require runtime reconfiguration to support different constant coefficients.

When constant coefficients change during runtime, the execution process for a static operator and a dynamic operator is illustrated in Figure 3.1(d). A static operator handles all coefficients with the same circuits, while a dynamic operator implements the customised circuits in each runtime scenario. As an example, we assume the constant coefficient in Figure 3.1(d) changes from 0.75 to 0.76 in the second runtime scenario, where 0.76 are represented with a 9-bit datum in fixed-point format. In order to support the coefficients during runtime, the static operator implements a $32 \cdot 9$ multiplier, with register values updated from 0.75 to 0.76 in the second runtime scenario. The implemented dynamic operator is customised for 0.75 in the first runtime scenario, and adapts to 0.76 in the second runtime scenario. Runtime reconfiguration operation is introduced before the second runtime scenario. While consuming less resources, a dynamic operator suffers runtime reconfiguration overhead to achieve the same design generality as a static operator.

Since the resource usage of constant operators depends on constant values, we can tune an algorithm to use hardware-preferable constant coefficients to reduce design resource usage. In this example, if in the second runtime scenario, 0.76 and 0.75 are mathematically equivalent for the target algorithm, a $32 \cdot 2$ multiplier can be used for the static operator, and the resource usage of the dynamic operator in the second runtime scenario can be reduced. For a reconfigurable design with various constant operators and multiple coefficients sets during runtime, this tuning process can significantly improve design performance. In this chapter, we define algorithm instances, algorithm design spaces and constant coefficient sets as follows.

- Algorithm instance: an instance of an algorithm, with initial algorithm parameters.
- Algorithm design space: for an algorithm instance, the range of algorithm parameters where these parameters (and thus constant coefficients) can vary without compromising algorithm mathematical correctness.
- Constant coefficient set: a point in an algorithm design space that specifies constant

values used in constant operators.

These three terms are hierarchical: an algorithm can contain multiple algorithm instance, each algorithm instance has a design space, and a design space contains various constant coefficient sets. As an example, an algorithm can be executed over different data sets, and parameter values vary with data sets. An algorithm instance refers to an algorithm with a specific data set (i.e. initial parameter values). We implement a reconfigurable design for each algorithm instance. During runtime, we map the reconfigurable designs into FPGAs, and dynamically reconfigure designs when supported data sets (i.e. algorithm parameters) change.

Challenges remain for how to combine circuit optimisation, runtime reconfiguration, and algorithm tuning. First, we need to build algorithm design space where constant coefficients can be tuned without compromising mathematical correctness of the target algorithm. Second, we need to model the mapping process from constant coefficients to customised operators, so that each point in the algorithm design space can be evaluated in terms of operator resource usage. Third, our design model needs to integrate the customised operators such that the optimisation process can be aware of the reduced operator resource usage and the introduced runtime reconfiguration operations.

3.2.2 Design Flow

The proposed approach includes front-end code generation (see Figure 3.2(a)) and back-end design optimisation (see Figure 3.2(b)). The front-end tool first builds high-level descriptions for the original algorithm. Tuned design details, such as data width and constant values, are fed into the descriptions. We use FloPoco libraries for fixed point arithmetic [dDP11] to generate VHDL code based on the descriptions. The generated designs are synthesised with vendor tools into hardware executables.

In the back-end, the design space of each algorithm instance is created, where each point in the design space corresponds to a valid constant coefficient set. A circuit model is developed for static and dynamic operators to capture the design properties of the algorithm points

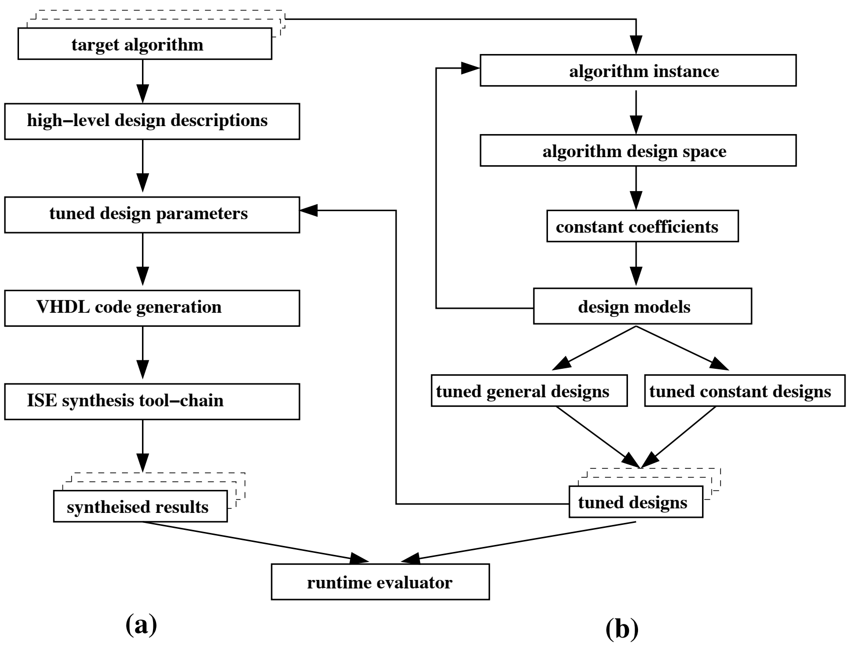


Figure 3.2: Design flow of the (a) design generation and (b) design optimisation.

in the design space. The tuning process explores the design space, and selects the constant coefficient set that will lead to minimal resource usage. These constant coefficients are used to build customised constant operators. A design model is built on top of the constant operators to optimise the runtime reconfigurable designs. In each runtime scenario, a tuned algorithm instance is generated. The optimised designs for various runtime scenarios are fed into code generation and synthesis tools to generate FPGA configuration files. During runtime, when a specific coefficient set needs to be used by the algorithm, the corresponding configuration file is downloaded into FPGAs.

Eventually, an evaluator is introduced to estimate design execution time based on design details, runtime parameters, and runtime reconfiguration operations. The evaluator selects the reconfigurable design (either static or dynamic) with minimal execution time, to avoid the case where using runtime reconfigurable designs reduces design performance.

Our current circuit-level approach uses FloPoco libraries [dDP11] to generate VHDL code for data-paths with customised constant operators. Given a target algorithm, a description of algorithm data-path is developed by designers using FloPoco libraries, in C++. For the back-end tool, designers specify the algorithm parameter range that is acceptable for all algorithm constant operators (e.g., given a finite-difference algorithm that is discussed in the following section, designers specify the range of step size in time and space dimensions). Starting from the specified parameter ranges, the back-end design models explore the design space of the constant operators, to select the algorithm parameters (i.e. constant coefficient values) that lead to minimum design resource usage. The selected constant coefficient values are written to an intermediate file, which is read by the algorithm description as program input. The algorithm description is then executed to generate data-paths with customised constant operators, with constant coefficients specified by the intermediate file. The generated data-paths (in VHDL) are then integrated into reconfigurable designs, and are synthesised into configuration files. When multiple algorithm instances need to be generated for different runtime scenarios, a script goes through the design flow multiple times to generate an optimised design (i.e. configuration file) for each runtime scenario.

3.2.3 Finite-Difference Algorithms

Algorithms that can benefit from the tuning process need to meet two criteria: (1) the algorithm involves constant operators, and (2) the constant coefficients depends on algorithm parameters. In this chapter, we use finite-difference algorithms to study the effectiveness of the proposed approach. Other algorithm domains that potentially can benefit from the proposed approach include signal filtering [BS08] and neural network training [CSL12].

The finite difference numerical method approximates solutions to differential equations. Derivatives are expressed with a finite difference between consecutive points in target dimensions. There are three main finite-difference methods in common use: implicit, explicit and Crank-Nicolson, corresponding to three different ways of expressing derivatives with neighbouring points. The proposed approach aims to construct a design space to optimise finite-difference algorithms, and is applicable to all three finite-difference methods.

To capture dynamic properties within target systems, a Partial Differential Equation (PDE) can be formulated as follows,

$$A\frac{\partial^2 f}{\partial t^2} = B\frac{\partial^2 f}{\partial s^2} + C\frac{\partial f}{\partial s} \tag{3.1}$$

where A, B and C are PDE parameters. Two finite-difference applications, option pricing and Reverse-Time Migration (RTM) are used as benchmark applications in this chapter. A financial option is a contract which allows its owner to sell assets at specific price in the future. Pricing options usually involves solving Black Scholes PDEs [Hul05], where $f_{t,s}$ denotes the option price for asset with price s at time t. A, B and C are determined by risk-free interest rate and volatility of the underlying assets. RTM is a seismic imaging technique that generates terrain images based on Earth's response to injected waves. Wave propagation is modelled with isotropic acoustic wave equation [AP+11], where $f_{t,s}$ is the injected wave at position s at time t. A, B and C are calculated with the sound speed and pressure in target terrains. Algorithm and application details for benchmark applications are presented in Section 3.7. While PDE variable t is in one dimension for all PDEs, variable s, known as stencil in finite-difference algorithms, can span multiple dimensions, as shown in Figure 3.3.(a). For option pricing, the number of dimensions in s is determined by how many assets are involved in the pricing process. For RTM, as the detected terrains are usually in 3-D, s covers three dimensions. By replacing the derivatives with finite difference expressions, Eq.3.1 can be mapped into discrete computational grids to solve the corresponding PDE. Eq.3.2 is expanded with one-dimension stencil in space. For applications with higher dimensions, dimension variable s is replaced with $(x, y, z...)$.

$$f_{t+1,s} = \alpha \cdot f_{t,s+1} + \beta \cdot f_{t,s} + \gamma \cdot f_{t,s-1} + \lambda \cdot f_{t-1,s} \tag{3.2}$$

$$\alpha = 2 - \frac{2B\Delta t^2}{A\Delta s^2} - \frac{2C\Delta t^2}{A\Delta s} \quad \beta = \frac{B\Delta t^2}{A\Delta s^2} - \frac{C\Delta t^2}{2A\Delta s} \quad \gamma = \frac{B\Delta t^2}{A\Delta s^2} - \frac{3C\Delta t^2}{2A\Delta s} \quad \lambda = -1 \tag{3.3}$$

$f_{t+1,s}$ indicates system status at the $t+1$ point in time dimension and the s point in space dimension, as shown in Figure 3.3(b). The corresponding hardware implementation is shown in Figure 3.3(c). If required input data $f_{t,s+1}$, $f_{t,s}$, $f_{t,s-1}$ and $f_{t-1,s}$ are available, the hardware module generates one result per clock cycle. The system status is propagated forward in the time dimension, with the step size Δt.

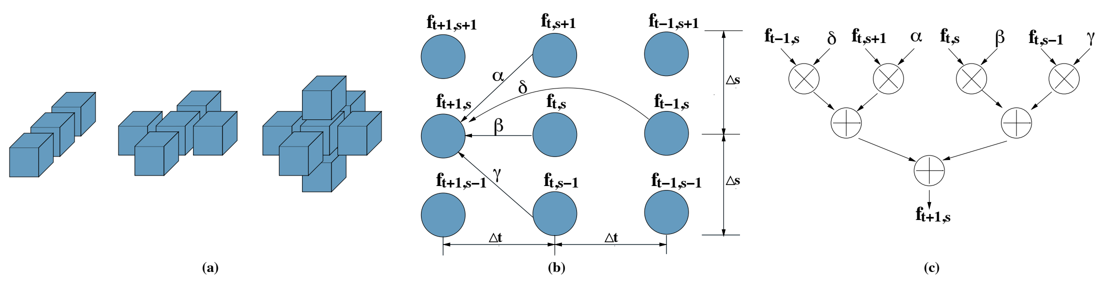


Figure 3.3: (a) Finite-difference stencil in 1-D, 2-D and 3-D space. (b) 1-D finite-difference computation for Eq.3.2 in time (t) and space (s) dimensions. (c) Hardware architecture for Eq.3.2.

3.3 Runtime Reconfiguration Opportunities at a Circuit Level

3.3.1 Idle Resource Units

At the circuit level, idle resource units refer to the logic gates in an arithmetic operator that are no longer required in a runtime scenario, which can be divided into two categories. First, for an arithmetic operator with constant coefficient, the logic gates coupled with the constant coefficients become redundant (i.e., idle resource units), as demonstrated in Figure 3.1(c). Second, for a tunable algorithm, there are various points in an algorithm design space, where all points are valid for the algorithm, and each point corresponds to a constant coefficient set. Therefore, there will be one point in the design space that consumes the minimal resources. We define the additional resources beyond the minimal resource usage as idle resource units, since results with same numerical accuracy are generated with additional logic gates.

We illustrate the tunable design space and idle resource units in Figure 3.4. We assume the three coefficients — 1.749511719, 1.750876563, and 1.751953125 — produce results that are within the acceptable error bound. Therefore for an algorithm that uses the relevant constant coefficient, all the above three coefficients will meet the required numerical accuracy. In this example, the coefficients are represented in a 10-bit fixed-point format. For static operators, the fewer bits that can be used to represent the coefficients, the fewer resources will be used. As shown in Figure 3.4, the second coefficient can be represented with three bits, since the remaining bits

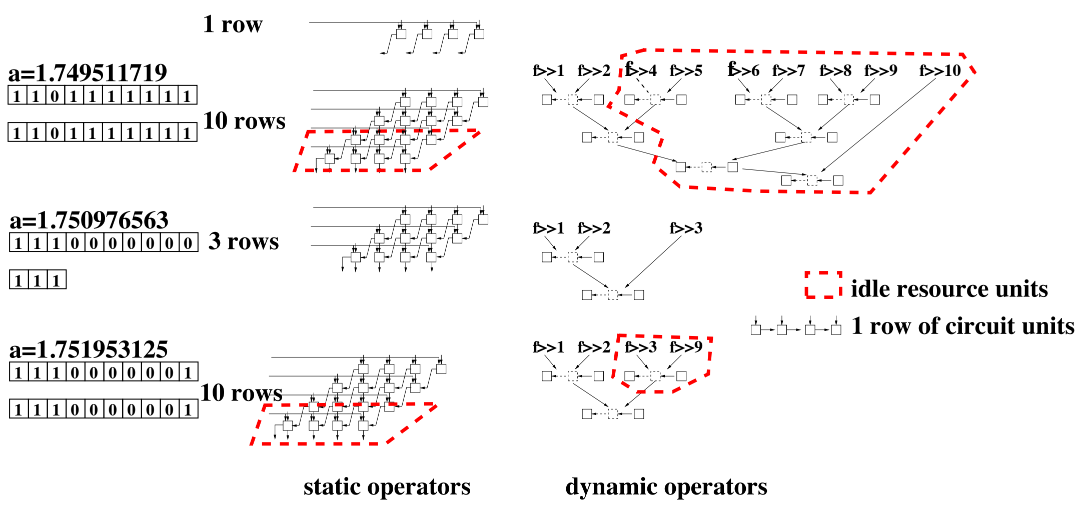


Figure 3.4: Customised operators and idle resource units for three coefficients that all produce results within error bound.

are all '0'. As a consequence, the implemented static operator achieves the minimum resource usage. The redundant logic gates for the other two static operators thus refer to the remaining 7 rows, which are labelled as idle resources. For dynamic operators, the resource usage depends on the non-zero bits in constant coefficients. As shown in Figure 3.4, the 3-bit coefficient leads to the operator with minimum resource usage. Similarly, the additional logic gates in the other two operators are labelled as idle resources. This example demonstrates two important aspects that drive algorithm design space building: (1) the resource usage of customised operators significantly depends on coefficient values, and (2) even a deeply customised operator can have idle resources, as the underlying coefficient can be undesirable for hardware implementation.

3.3.2 Constructing the Algorithm Design Space

An algorithm design space refers to the range of algorithm parameters where algorithm constant coefficients can vary without compromising mathematical correctness. For a tunable application domain, in order to construct an algorithm design space, we need to first find the algorithm parameters that define the coefficient values, and then explore the valid parameter range to collect valid coefficient sets.

For finite-difference algorithms, a valid design space refers to the range of step size that ensures both computation accuracy and PDE stability. Computation accuracy is specified by users, and is expressed as the number of bits B involved in computation. Increasing B results in a larger design space, since the number of constant coefficient sets increases with B. The specified accuracy is ensured during algorithm tuning. The stability condition of PDEs requires the local error in finite-difference algorithms to be reduced in subsequent computations. The local error is defined as the difference between the actual value $a_{(t,s)}$ in a PED and the discretised value $f_{(t,s)}$ in the corresponding finite-difference algorithm. Based on Von Neumann stability analysis [CvN50], the stability condition for a finite-difference equation can be expressed as follows. The $|g|$ is bounded to be less than 0.5 instead of 1 to ensure fast convergence.

$$\epsilon_{(t,s)} = a_{(t,s)} - f_{(t,s)} \tag{3.4}$$

$$\epsilon_{(t+1,s)} = g \cdot \epsilon_{(t,s)} \quad |g| \leq 0.5 \tag{3.5}$$

$$\epsilon_{(t+1,s+1)} = e^{c(t+\Delta t)} e^{ik_m(s+\Delta s)} \quad k_m = \frac{\pi m}{L} \quad m \in (1, 2, ... \frac{L}{\Delta s}) \tag{3.6}$$

Among the stable finite-difference algorithms, users specify one initial parameter set $(\Delta t, \Delta s)$, based on available computational resources and stability requirements. A small perturbation ζ is added into the specified step size, to provide a tunable design space which satisfies both stability conditions and user requirements, as shown in Eq. 3.7. Within the derived valid design space, finite-difference algorithms can be tuned with design models for both static and dynamic operators. The design spaces of option pricing and RTM for the derived ζ is shown in Figure 3.5 and 3.6, where step size Δt and Δs vary from $(0.975\Delta t, 0.975\Delta s)$ to $(1.025\Delta t, 1.025\Delta s)$. For the same finite-difference algorithm, the resource usage is almost doubled from the optimal case to the worst case. In other words, if a given finite-difference algorithm can be properly tuned, resource usage of its hardware implementations can be halved.

$$\overline{\Delta t} = (1+\zeta)\Delta t \quad \overline{\Delta s} = (1+\zeta)\Delta s \quad |\zeta| \leq 0.025 \tag{3.7}$$

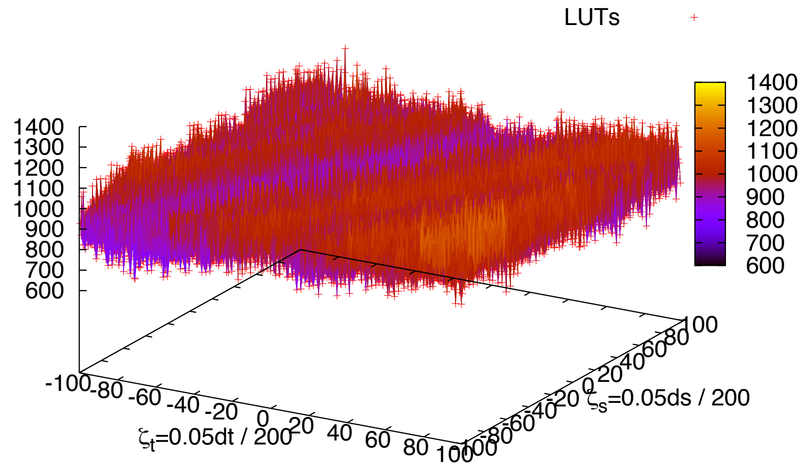


Figure 3.5: Design space of option pricing for different computational grids (dt, ds)

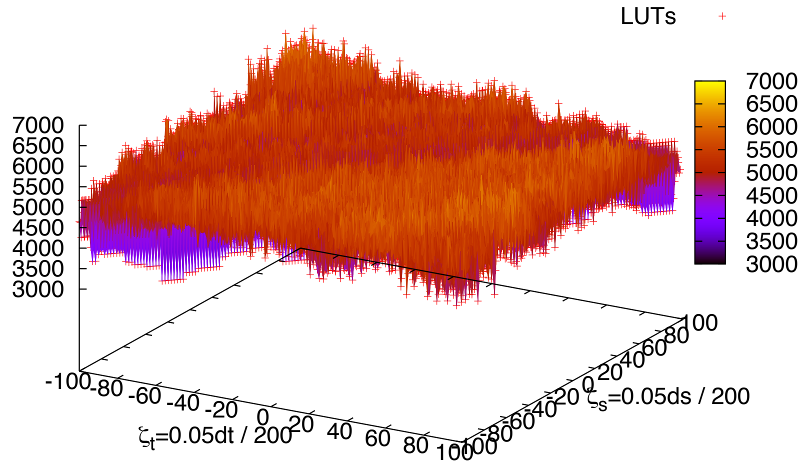


Figure 3.6: Design space of RTM for different computational grids (dt, ds)

3.4 Arithmetic Operator Optimisation

Arithmetic operator optimisation process refers to exploring possible coefficients in an algorithm design space to minimise the hardware resource usage. In order to properly select the operator to use, the optimisation tool needs to be aware of the corresponding resource usage for each point in the design space. Previous work [JBLT12b] used synthesised results to evaluate the tuned designs, which is time-consuming and not sustainable for large-scale designs. We develop circuit design models for both static and dynamic operators to rapidly evaluate the coefficients in an algorithm design space.

3.4.1 Optimising Static Operator

For static operators, we develop a design model to compress the data width for constant coefficients. As shown in Figure 3.4, while constant α with value 1.749511, 1.75097 and 1.75195 can all be mapped into proper computation grids, the constant 1.750976 outperforms other neighbouring constant coefficients in terms of resource usage, as it can be represented with fewer bits without precision reduction. Either for arithmetic operators based on DSP blocks [dDP09, BdDPT10] or for arithmetic operators mapped into LUTs [TW04], reducing input data width directly reduces resource usage.

For each point in a valid design space, the generated constant coefficients are represented with two's complement $Twos$. The design model traverses from the least significant bit to the most significant bit of $Twos$, until the first "1" bit is found. The number of data bits between the most significant bit and the first "1" bit is updated as the data width W_{c_i} of the explored constant, where c_i indicates the constant. The data width is propagated through connected operators in an algorithm instance, under the following rules.

$$\forall c = a \pm b \quad W_c = max(W_a, W_b) \qquad\qquad \forall c = a \cdot b \quad W_c = W_a + W_b \tag{3.8}$$

For adders and subtracters, the output data width is the same as the maximum input data width, while for multipliers, the output data width is the sum of the two input data width. Adders and subtracters are directly mapped into LUTs, with each LUT accommodating a 1-bit adder/subtracter. The carrier bits are fed forward along with output bits. Therefore, the resource usage is the same as the output data width of mapped adders/subtracters. For multipliers, an adder matrix is used to accumulate the multiplication results. The resource

Algorithm 3 Design model for dynamic operators.

Input: Constant coefficients expressed with CSD coding *csd*.
Output: Resource consumption R_{dyn}

```
 1: for i = 0 → B do
 2:     if $csd_i$ == "+" or $csd_i$ == "-" then
 3:         $N_{csd}$ += 1
 4:     end if
 5: end for
 6: while $N_{csd}/2$ do
 7:     op = $N_{csd}$ / 2
 8:     mod= $N_{csd}$ % 2
 9:     B++
10:     $R_{dyn}$ += B · op
11:     $N_{csd}$ = op + mod
12: end while
```

usage R_{sta} for a multiplier can be estimated with:

$$\forall c = a \cdot b \quad R_{sta} = \sum_{i=W_a}^{W_b} i \quad W_a \leq W_b \quad i \in (W_a, W_a + 1, W_a + 2 ..., W_b) \tag{3.9}$$

3.4.2 Optimising Dynamic Operator

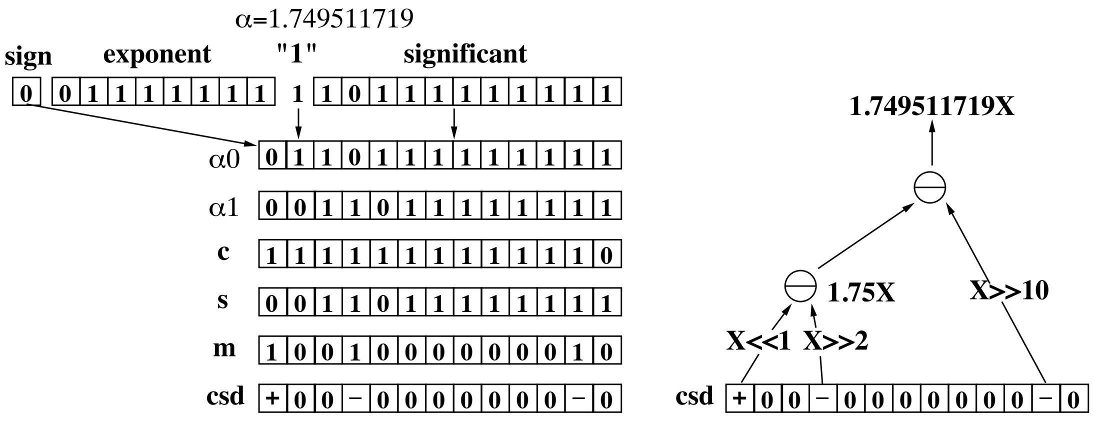
In this work, dynamic operators are implemented based on the Canonical-Signed-Digit (CSD) coding [Rei60]. Constant coefficients can be converted into CSD, to construct a multiplier with addition, subtraction and shifting operations. Since a dynamic operators is customised for a specific constant, dynamic operators can only be implemented with fine-grained logic units (LUTs and FFs) of FPGAs. Figure 3.7 shows the steps to convert a floating point datum to CSD. First, a floating point datum is converted to its fixed-point equivalent. We merge the significant bits of floating point data, the hidden 1 in floating point data and the sign bit together. The fixed-point representation is labelled as $\alpha 0$. Second, $\alpha 0$ is left shifted by 1 bit to form $\alpha 1$. Carrier bits c are calculated based on the original data and shifted data, as shown in Eq.3.10. Third, sign s and magnitude m of CSD data are calculated, based on original data, shifted data and carrier bits. Finally, the CSD bits *csd* are calculated with sign s and magnitude m as follows.

$$c_{i+1} = \alpha0_i \cdot \alpha1_i + c_i \cdot \alpha0_i + c_i \cdot \alpha1i \quad c_0 = 0 \tag{3.10}$$

$$s_i = \alpha1_i \tag{3.11}$$

$$m_i = \overline{\alpha0_i} \cdot c_i + \alpha0_i \cdot \overline{c_i} = \alpha0_i \oplus c_i \tag{3.12}$$

$$csd_i = \begin{cases} + & : m_i = 1 \quad s_i = 1 \\ - & : m_i = 1 \quad s_i = 0 \\ 0 & : m_i = 0 \end{cases} \tag{3.13}$$



α=1.749511719
sign
exponent
"1"
significant
0
0 1 1 1 1 1 1 1 1
1
1 0 1 1 1 1 1 1 1 1 1
α0 0 1 1 0 1 1 1 1 1 1 1 1 1
α1 0 0 1 1 0 1 1 1 1 1 1 1 1
c 1 1 1 1 1 1 1 1 1 1 1 1 0
s 0 0 1 1 0 1 1 1 1 1 1 1 1
m 1 0 0 1 0 0 0 0 0 0 0 1 0
csd + 0 0 − 0 0 0 0 0 0 0 − 0
1.749511719X
1.75X
X>>10
X<<1
X>>2
csd + 0 0 − 0 0 0 0 0 0 0 − 0

Figure 3.7: A dynamic operator implementation with CSD coding.

A design model is developed to estimate the resource usage by simulating the building process for dynamic operators. As shown in Algorithm 3, the number of non-zero bits N_{csd} indicates the number of partial results that need to be summed. The accumulation process of partial results is divided into several stages. As one adder/subtracter can process two partial results, $N_{csd}/2$ adders/subtracters are required for the first stage, generating $N_{csd}/2$ partial results for the second stage (line 7). If N_{csd} is not an even number, the remainders of the first stage $N_{csd}\%2$ are added into the following stage (line 8). N_{csd} is updated for the second stage, as $N_{csd}/2 + N_{csd}\%2$. Additional stages are introduced until the final result is generated, i.e., $N_{csd}/2 = 0$ (line 6). Correspondingly, the resource usage for a stage can be estimated with the number of adders / subtracters in that stage. For example, the resource usage for the first stage is $(N_{csd}/2) \cdot (B + 1)$ (line 10), where $B + 1$ covers the width of the adders/subtracters,

Table 3.1: Variables and parameters in the circuit-level design model.

variables		**parameters**	
		indices	
o	operator type	s	on-chip resources
sta/dyn	static or dynamic operator	bw	memory bandwidth
a	array indices	d	data-path indices
c	clock cycle	bit	data bit-width
fix	fixed or floating point	D	dimension index
		design model	
P	design parallelism	A	available resources
L_s	logic resource usage	BW	available bandwidth
M_s	memory resource usage	$N_{ari,o}$	number of operator o
M_{bw}	memory bandwidth usage	$R_{s,o}$	resource usage of an operator o in type s
N_{arr}	number of arrays	$mem_{a,d}$	on-chip data of array a in data-path d
		domain-specific aspects	
dm_{ds}	impacts on data size	n_D	dimension i size
dm_{Ms}	impacts on memory usage	w_D	finite-difference order in dimension i
sk	spatial blocking ratio	tk	temporal blocking ratio
		performance model	
T	overall execution time	N_r	number of reconfiguration units
RT_{bne}	runtime benefits	R_U	configuration unit size
O_{rf}	reconfiguration time	R_{dp}	data-path size
ϕ	(re)configuration throughput	θ	data transfer throughput
		γ	configuration file size per reconfiguration unit

and the additional 1 bit is used to prevent overflow.

3.5 Runtime Reconfigurable Design Optimisation

Once the customised operators are selected, the next step is to use them in a reconfigurable design. Using the customised operators reduces the resource usage of a `well-behaved` data-path (discussed in Chapter 2.2.3), and thus increases the number of replicated data-paths under the same resource constrains. Given a tunable algorithm with customised operators, we develop a design model to ensure the generated reconfigurable design can fully exploit available resources.

3.5.1 Design Model

The design model captures reconfigurable design properties in three aspects: computational resource usage, memory resource usage, and off-chip memory bandwidth usage. Table 3.1 lists the design parameters used in this model. At the circuit level, the System Resource Abstraction (SRA) file contains the available resources for an FPGA, along with the prepared resource usage of arithmetic operators. Bounded by the available resources in SRA, the objective of the model is to maximise design throughput. The resource usage refers to the number of used on-chip FPGA resources (LUTs, FFs, DSPs and BRAMs), and the required off-chip memory bandwidth to ensure all implemented data-paths can work in parallel.

In a DFG extracted from high-level descriptions, the nodes are implemented as a pipelined data-path, as shown in Figure 3.8(a). With the arithmetic operator resource usage information stored in RSA, the resources consumed by P replicated data-paths can be estimated as:

$$L_s = P \cdot \sum_{o \in \odot} N_{\mathrm{ari,o}} \cdot R_{\mathrm{s,o}} \cdot B_{s,o,bit,fix} \quad \odot = \{+, -, *, \div, sta, dyn\} \quad s \in \{LUT, FF, DSP\} \tag{3.14}$$

where L_s accounts for logic resource usage, $N_{ari,\mathrm{o}}$ indicates the number of operators for arithmetic operation type o (an operator o can be an adder, a subtracter, a multiplier, a divider, a static multiplier, or a dynamic multiplier), $R_{\mathrm{s,o}}$ indicates the number of on-chip logic resource s consumed by one arithmetic operator o, and $B_{s,o,bit,fix}$ accounts for the impacts of bit-bit data presentation fix. $fix = 0$ indicates floating-point data, while $fix = 1$ indicates fixed-point data are used. Given enough input data, the P data-paths can generate P results per clock cycle. The resource type s includes LUT, FF and DSP. Therefore, the resource usage of LUT, FF, and DSP is estimated with Eq.3.14 by specifying proper $R_{\mathrm{s,o}}$.

In the extracted DFG, the edges represent the communication operations in an algorithm. The edges between arithmetic operators are implemented as wire connections, while the edges connected to sink nodes or source nodes are labelled as data access edges. Our design model explores on-chip data reuse by grouping the data access edges. As shown in Figure 3.8(b), when

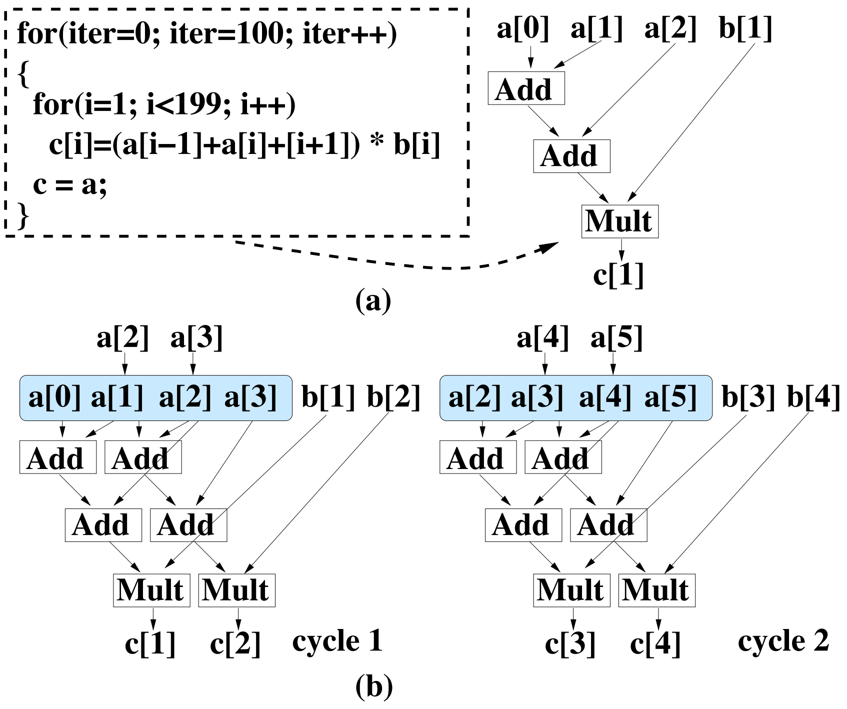


Figure 3.8: An example computational kernel described in C, when implemented with (a) a single data-path and (b) two replicated data-paths.

multiple data-paths are replicated on-chip, the data access edges overlap with each other. As an example, both data-paths in Figure 3.8(b) access $a[1]$ and $a[2]$ at cycle 1. We group the data access edges for an array as mem_a, where $mem_{a,max}$ and $mem_{a,min}$ respectively indicate the maximum and the minimum offset values in mem_a. For the example in Figure 3.8, $mem_{a,max} = 1$ and $mem_{a,min} = -1$. The mem is implemented as an on-chip memory architecture to buffer the accessed data. In this example, buffering the accessed data a reduces the number of accessed data in each cycle from 6 to 4. The on-chip memory resource usage can be estimated as the number of memory blocks consumed by the grouped mem,

$$M_s = \sum_{a=1}^{N_{arr}} \sum_{d=1}^{P} (mem_{a,d,max} - mem_{a,d,min} + 1) \cdot bit \tag{3.15}$$

where N_{arr} indicates the number of arrays in the design, d indicates a replicated data-path, and bit indicates the number of bits for each data.

With data accesses shared on-chip, the off-chip bandwidth requirements can be calculated as the new data in *mem*, compared with the *mem* in the previous clock cycle.

$$M_{bw} = \sum_{a=1}^{N_{arr}} \sum_{d=1}^{P} (mem_{a,d,c+1} \cup mem_{a,d,c} - mem_{a,d,c}) \cdot bit \tag{3.16}$$

where c is the clock cycle, and $mem_{a,d,c+1} \cup mem_{a,d,c} - mem_{a,d,c}$ indicates the new data in cycle $c + 1$. For the example in Figure 3.8(b), $(mem_{a,d,c+1} \cup mem_{a,d,c} = (a[0] \sim a[5])$, and $mem_{a,d,c} = a[0] \sim a[3]$. Therefore, two data items are loaded from off-chip memory per clock cycle.

An optimisation model is developed to determine the number of replicated data-paths P (parallelism) to achieve minimum execution time for each configuration, i.e., the ratio between overall data size ds and computational capacity $P \cdot f_{dp}$.

$$\textbf{minimise: } \frac{ds \cdot dm_{ds}}{P \cdot f_{dp}} \tag{3.17}$$

subject to:

$$L_{LUT/FF/DSP} \cdot P \cdot dm_{LUT/FF/DSP} + I_{LUT/FF/DSP} \leq A_{LUT/FF/DSP} \tag{3.18}$$

$$M_s \cdot dm_{MS} + I_{MS} \leq A_{MS} \tag{3.19}$$

$$M_{bw} \cdot dm_{BW} \leq BW \tag{3.20}$$

In this model, we use the available on-chip resources and off-chip memory bandwidth in SRA as constraints, and divide the circuit properties into general aspects and domain-specific aspects. The resource constraints contain the available LUTs (A_{LT}), FFs (A_{FF}), BRAMs (A_{MS}) and DSPS (A_{DP}) on-chip, and the off-chip memory bandwidth BW. In correspondence, we use $I_{LUT/FF/DSP/MS}$ to indicate the infrastructure resource usage for LUTs, FFs, DSPs and BRAMs. In SRA, the infrastructures refer to communication infrastructures such as memory controllers and PCIe controllers. In general, the data-path resource usage grows linearly with P, and the memory usage is analysed by Eq.3.15 and Eq.3.16. In practice, domain-specific optimisation techniques can be applied to further customise the implemented circuits, and the domain-specific aspects are labelled as dm in the optimisation model.

3.5.2 Domain-Specific Aspects

Domain-specific aspects for finite-difference applications include spatial blocking and temporal blocking. Figure 3.9(a) presents an example finite-difference architecture, where a data-path is connected to an on-chip memory architecture. Three-dimensional (3D) data are used in this example, and three slices of data in the slowest dimensions are buffered. As demonstrated in Figure 3.9(b), when four data-paths are replicated, the memory usage stays the same, and the bandwidth requirements increase linearly. As dimension size nx and ny increase, the memory usage can easily exceed on-chip memory capacity for large-scale finite-difference algorithms. Moreover, in order to support `well-behaved` data-paths, the off-chip memory channels need to accommodate the parallel data accesses from the replicated data-paths. To address these issues, spatial blocking reorganises computation order to reduce on-chip memory usage, and temporal blocking supports processing multiple time steps on the same memory pass to save bandwidth.

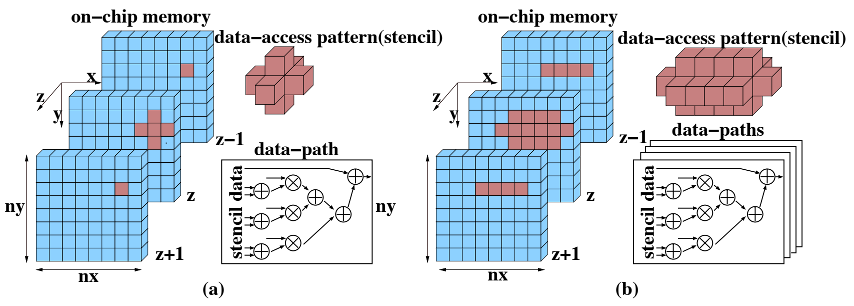


Figure 3.9: Data access patterns, memory architectures and data-paths in streaming architectures for a finite-difference algorithm with (a) a single data-path ($P = 1$) and (b) four replicated data-paths ($P = 4$). 3-D data structures are used.

Halo data in finite-difference algorithms refer to the data that are involved in computation, but are not updated during runtime. Algorithm 4 shows the computational kernel of a first-order finite-difference problem. The computation of one data point requires its neighbouring data $x \pm 1$, $y \pm 1$ and $z \pm 1$. Therefore the outside data layer of the 3D data, as shown in Figure 3.10(a), is not updated during runtime. After temporal and spatial blocking, additional halo layers are introduced. The halo data in one block are kernel data in another data block, and therefore are updated by saving all the data block results in one shared off-chip memory (see Figure 3.10(b)).

Algorithm 4 A first-order finite-difference algorithm with three dimensions x, y and z.

```
for $t = 0 \leftarrow$ nt-1 do
  for $z = 0 \leftarrow$ nx-1 do
    for $y = 0 \leftarrow$ ny-1 do
      for $x = 0 \leftarrow$ nz-1 do
        p(t,x,y,z) =dvv *(
        c0 * p(t,x,y,z) +
        c11* (p(t,x-1,y,z) + p(t,x+1,y,z)))
        c21* (p(t,x,y-1,z) + p(t,x,y+1,z)))
        c31* (p(t,x,y,z-1) + p(t,x,y,z+1)))
        d0 * p(t,x,y,z) + d1 * p(t-1,x,y,z-1) + f(t,x,y,z);
      end for
    end for
  end for
end for
```

Spatial blocking reduces memory resource usage. While the number of buffered data slices is algorithm-specific, the slice size depends on the size of the corresponding dimensions (nx and ny in Figure 3.9 and Figure 3.10(a)). When the number of dimensions increases, memory resource usage can easily exceed resource constraints. Blocking dimensions in memory slices regroups streaming patterns in the blocked dimensions, which effectively reduces the slice size and memory resource usage. As an example, in Figure 3.10(b), halving nx and ny reduces memory usage to 1/4, and one more layer of halo data are introduced for each of the four data blocks. In a dimension D with n_D kernel data (as shown in Figure 3.10(a)) and blocking ratio sk_D, the size of blocked dimension can be expressed as $\frac{n_D}{sk_D} + 2 \cdot w_D$. Since halo data are distributed to each data block, spatial blocking increases the overall data size compared with unblocked designs.

Spatial blocking affects the overall data size ds and on-chip memory resource usage M_s. For a finite-difference application with N_D dimensions, the domain-specific parameter for data size can be expressed as:

$$dm_{ds} = \prod_{D=1}^{N_D-1} \frac{n_D + 2 \cdot w_D \cdot sk_D}{n_D} \tag{3.21}$$

where $2 \cdot w_D \cdot sk_D$ indicates the additional halo data size in dimension i, and n_D is the unblocked dimension size. Since the blocked dimension size drops from n_D to $\frac{n_D}{sk_D} + 2 \cdot w_D$, the reduction in memory usage can be expressed as:

$$dm_{MS} = \prod_{D=1}^{N_D-1} \frac{\frac{n_D}{sk_D} + 2 \cdot w_D}{n_D} \tag{3.22}$$

where $\prod_{D=1}^{N_D-1} \frac{n_D}{sk_D} + 2 \cdot w_D$ estimates the size of one buffered data slice after spatial blocking.

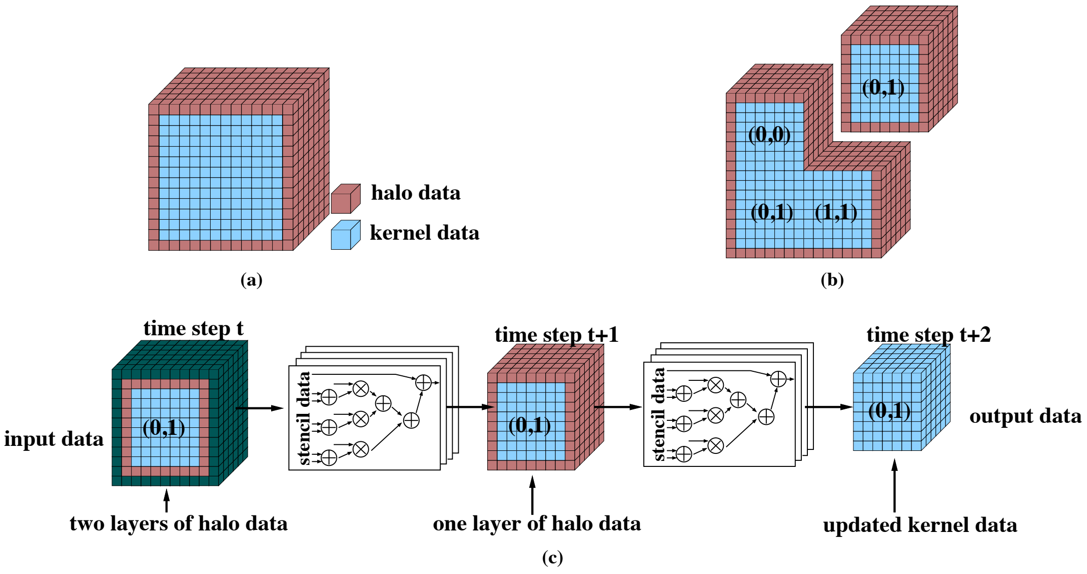


Figure 3.10: Data cube to process in finite-difference applications for (a) original data, (b) after spatial blocking, and (c) after spatial and temporal blocking.

Temporal blocking is applied to reduce memory bandwidth requirements. For a given memory bandwidth, there will be a point where the memory system cannot afford to load and to write P data units per clock cycle. As shown in Figure 3.10(c), when memory channels are saturated, output data of the current time step can be stored as intermediate data accessed as input data for the next step, accomplishing multiple time steps in one memory pass. The memory architecture is replicated to accommodate the intermediate data, and the attached data-paths are also replicated to process the intermediate data in parallel. Meanwhile, for the spatially blocked data, accomplishing one more time step on-chip introduces one more layer of halo data for data blocks, to ensure the halo data of intermediate results can be properly updated without synchronising with neighbouring blocks (see Figure 3.10(c)). Therefore, the size of blocked dimension D with spatial blocking ratio sk_D and temporal blocking factor tk can be expressed as $\frac{n_D}{sk_D} + 2 \cdot w_D \cdot tk$, where tk layers of halo data are represented as $2 \cdot w_D \cdot tk$. The size of one data slice after spatial and temporal blocking is:

$$sl = \prod_{i=1}^{D-1} \left(\frac{n_D}{sk_D} + 2 \cdot w_D \cdot tk \right) \tag{3.23}$$

Temporal blocking affects the consumed memory bandwidth M_{bw}, the overall parallelism P, the overall data size ds, and therefore the memory resource usage M_s. As the temporal blocking ratio tk increases, more data-paths are replicated without loading and writing data from off-chip memory, as the data are directly transferred into following data-paths, as shown in Figure 3.10(c). In Eq.3.16, the data accesses of par data-paths are combined to calculate bandwidth requirement M_{bw}. After temporal blocking, only $\frac{1}{tk}$ data-paths consume the off-chip bandwidth. To cooperate the impact of temporal blocking, we use par to indicate the number of initial data-paths. Therefore, the domain-specific parameter in Eq.3.20 dm_{BW} is 1, and the overall parallelism in Eq.3.18 is expressed as $par \cdot tk$. Similar to spatial blocking, temporal blocking brings overhead as the overall data to process is increased. After spatial blocking and temporal blocking, dm_{ds} and dm_{MS} can be expressed as

$$dm_{BW} = 1 \tag{3.24}$$

$$P = par \cdot tk \tag{3.25}$$

$$dm_{ds} = \prod_{D=1}^{N_D-1} \frac{n_D + 2 \cdot w_D \cdot sk_D \cdot tk}{n_D} \tag{3.26}$$

$$dm_{MS} = \left(\prod_{D=1}^{N_D-1} \frac{n_D}{\frac{n_D}{sk_D} + 2 \cdot w_D \cdot tk} \right) \cdot tk \tag{3.27}$$

where $2 \cdot w_D \cdot sk \cdot tk$ indicates the additional data for a blocked data dimension, and the multiplied parameter $\cdot tk$ in Eq.3.27 indicates tk on-chip memory architectures are implemented.

3.6 Runtime Evaluation

We use a runtime evaluator to schedule the optimised static and dynamic designs, and to ensure high performance during runtime. We develop a performance model to estimate execution time

T with runtime data size ds.

$$T = \sum_{rf=1}^{R} \frac{ds_{rf} \cdot dm_{ds}}{par_{rf} \cdot tk_{rf} \cdot f_{dp,rf}} + O_{rf} \tag{3.28}$$

where f_{dp} is the operating frequency, $par \cdot tk$ indicates the design parallelism, $ds \cdot dm_{ds}$ indicates the overall data to process, and O_{rf} is the reconfiguration time to switch from configuration rf to configuration $rf + 1$. At the circuit level, each algorithm instance has its configuration file, and therefore various configuration files need to be switched to support finite-difference applications with different parameters, i.e., different constant coefficient sets. Another approach to support various algorithms is to implement reconfigurable design with static operators, which ensures all constant coefficient sets to be accommodated into the same operator set. Two execution strategies thus can be applied: (1) implementing dynamic operators supported with runtime reconfiguration, or (2) implementing optimised static operators to support all runtime scenarios with one configuration. Given a reconfigurable designs with R different constant sets to support, the execution time and reconfiguration time is accumulated. While the dynamic designs achieve higher parallelism, the static designs do not suffer reconfiguration overhead. The performance of these designs are evaluated, and the runtime evaluator selects the design with minimum overall execution time to execute.

Reconfiguration overhead includes configuration time and the time to preserve application context data. The configuration time can be calculated as the ratio between configuration file size and the configuration interface throughput θ, and the data transfer time can be estimated based on data size and data interface throughput ϕ.

$$O_{rf} = \frac{N_r \cdot \gamma}{\theta} + \frac{2 \cdot ds_{rf} \cdot dm_{ds}}{\phi} \tag{3.29}$$

$$N_r = \frac{I + R_{dp} \cdot P}{R_U} \tag{3.30}$$

where N_r is the number of reconfiguration units used, and γ accounts for the configuration file size for each consumed resource unit. For FR designs, a reconfiguration unit refers a full configuration file, while a reconfiguration unit in PR designs can refer to a clock region or a configuration frame. PR designs do not need to transfer intermediate results, since memory controllers are still alive during reconfiguration. R_U indicates the resource usage for one reconfiguration unit, R_{dp} accounts the resource usage for one data-path, and P is the number of implemented data-paths. We use FR designs in the current circuit-level approach, as (1) reconfiguring all replicated data-paths takes a long time for both FR and PR designs, the reduction in reconfiguration time is small, and (2) the performance of PR designs is often limited by their reduced clock frequencies. We discuss the use of PR designs in Chapter 6.

The runtime benefits RT_{bne} determines which design to execute. During runtime, an application executes its static design if $RT_{bne} < 0$, while $RT_{bne} > 0$ means the dynamic design provides better performance. We estimate the runtime benefits as follows,

$$\begin{aligned} RT_{bne} &= T_{st} - T_{dy} && (3.31)\\ &= \frac{\sum_{rf=1}^{R} ds \cdot dm_{ds}}{par \cdot tk \cdot f_{dp}} - (\sum_{rf=1}^{R} \frac{ds_{rf} \cdot dm_{ds}}{par_{rf} \cdot tk_{rf} \cdot f_{dp,rf}} + O_{rf}) && (3.32) \end{aligned}$$

where T_{st} refers to the overall execution time for static designs, and T_{dy} refers to the overall execution time for runtime reconfigurable designs. The dynamic and the static designs possess different data-path resource usage R_{dp}, which leads to different par, tk, dm_{ds} and O_{rf} for each configuration.

3.7 Benchmark Applications

3.7.1 Option Pricing

An option is a financial instrument which provides its owner the right to buy or to sell an asset at a fixed price in the future. A *call option* allows owners to buy asset, while a *put option* allows owners to sell asset. Options are popular in the financial industry and pricing options usually involves solving PDEs, especially the Black Scholes PDE [Hul05]. The Black Scholes PDE with one variable (asset) following geometric Brownian motion is described as

Eq.3.33, where $f_{(t,s)}$ denotes the price of the option, s denotes the value of the underlying asset, t denotes a particular time, τ is the risk-free interest rate, σ is the volatility of the underlying asset. Using explicit finite-difference expressions to replace the derivatives, the asset value $f_{(t,s)}$ can be calculated as in Eq.3.34, where α, β and γ are the constants determined by σ, τ and computational grid step size.

$$\frac{\partial f_{(t,s)}}{\partial t} + \tau s \frac{\partial f_{(t,s)}}{\partial s} + \frac{1}{2}\sigma^2 \frac{\partial^2 f_{(t,s)}}{\partial s^2} = \tau f_{(t,s)} \tag{3.33}$$

$$f_{(t,s)} = \alpha f_{(t-1,s+1)} + \beta f_{(t-1,s)} + \gamma f_{(t-1,s-1)} \tag{3.34}$$

3.7.2 Reverse Time Migration

Reverse Time Migration (RTM) is an advanced seismic imaging technique to detect terrain images of geological structures, based on the Earth's response to injected acoustic waves. The wave propagation within the tested media is simulated forward, and calculated backward, forming a closed loop to correct the terrain image. The propagation of injected waves is modelled with the isotropic acoustic wave equation:

$$\frac{d^2 p_{(t,s)}}{dt^2} + {dvv_{(s)}}^2 \bigtriangledown^2 p_{(t,s)} = f_{(t,s)} \tag{3.35}$$

where $dvv_{(s)}$ is the sound speed at terrain point s, $p_{(t,s)}$ is the pressure value, and $f_{(t,s)}$ is the input wave. Three dimensions are covered in the finite-difference space, i.e., $s = (x, y, z)$. The propagation in space is replaced with fifth-order finite-difference expressions, and first-order approximation is used for propagation in time. With derivatives replaced with finite-difference expressions, the dynamic model can be mapped into computational grids as follows.

$$p_{(t,s)} = dvv_{(x,y,z)}(\sum_{i=x}^{z}\sum_{j=1}^{5} c_{ij} \cdot (p_{(t,si-j)} + p_{(t,si+j)}) + c \cdot p_{(t,s)}) + p_{(t,s)} - 2 \cdot p_{(t-1,s)} \tag{3.36}$$

where $p_{(t,si-j)}$ refers to $p_{(t,x-j,y,z)}$ when $i = x$. c_{ij}, and c_{ij} and c are constant coefficients tuned in the proposed approach.

3.8 Results

The effectiveness of the proposed approach is evaluated in three aspects: model accuracy, resource usage of optimised designs and runtime performance. We collect the resource usage of static and dynamic designs from Xilinx ISE 13.3 post-synthesis results. A reconfigurable design can map its arithmetic operators into either LUTs or DSP blocks. If DSP blocks are used, the resource reductions due to the proposed approach is limited by resource granularity, since the minimal input width for Xilinx DSP blocks is 18-bit. As an example, while LUT usage reduces linearly as operator data width decreases, both an 8-bit multiplier and an 18-bit multiplier consume a DSP block. In addition, since previous work use LUT usage to evaluate the approach efficiency, we map arithmetic operators into LUTs to provide fair comparison. We set the precision requirement B in the experiments to be 24 bits, based on previous experiment results for precision optimisation [NJL+12b].

Current designs target at a Xilinx Virtex-6 SX475T FPGA hosted by a MAX3424A card from Maxeler Technologies, with memory bandwidth of 38.4 GB/s. Our current circuit-level approach uses FloPoco libraries [dDP11] to generate VHDL codes for data-paths with customised constant operators. The current system (MAX3424A card from Maxeler Technologies) we use to test designs only captures designs with MaxCompiler, which is not compatible with VHDL codes. Therefore, we simulate the runtime performance of optimised designs in two steps. (1) We develop the benchmark applications with MaxCompiler, and measure their runtime performance. (2) We calculate the designs parallelism P by comparing the original resource usage and the resource usage after optimisation, and estimate the runtime performance based on Eq.3.28. Results from previous work for optimising constant operators in finite-difference algorithms and accelerating finite-difference algorithms are compared with results for the proposed approach.

3.8.1 Model Accuracy

The proposed approach uses design models to capture optimal designs without synthesising each possible coefficient sets in algorithm design space. Therefore, proper design space exploration at the circuit level calls for high model accuracy. We define the model accuracy as the ratio between estimated resource usage and measured post-synthesis resource usage.

In order to test the generality of the tuning process, we randomly generate 100 algorithm instances for each benchmark application in this experiment, and each algorithm instance has randomly initialised parameters. Bounded by the stability requirements in Eq.3.6, the proposed approach construct a design space for each algorithm instance. In an algorithm design space, the operator optimisation process approach evaluates resource usage of points in the design space, and picks the point (constant coefficient set) with minimal resource usage. We refer the reader to the Glossary and Section 3.2.1 which define the terms algorithm instances, algorithm design space and constant coefficient set. The estimated and the synthesised resource usage are shown in Figure 3.11 and 3.12. In the worst case, the model accuracy for the dynamic designs of the option pricing application is around 80%, as there are only three constant operators involved in the designs (as shown in Eq.3.34). The small resource usage amplifies the error ratios. In the other three cases, the model accuracy is around 90%. More importantly, despite the difference between estimated values and actual resource usage, the design models capture the general trend of design properties, as shown in Figure 3.11 and 3.12. With the high-level design models, design space in finite-difference algorithms can be explored promptly and properly.

3.8.2 Resource Usage

We compare the resource usage of application data-paths before and after optimisation in Figure 3.13 and Figure 3.14, with resource usage expressed with the number of consumed LUTs. We define improvement ratio as the reduction in resource usage after optimisation, for both static and dynamic designs. An original static design refers to a data-path using general operators with full input bit-width, and an original dynamic design refers to a data-path customised for the initial constant coefficients. For the original static designs, 3042 LUTs are consumed for the option pricing application, and 15964 LUTs are consumed for the RTM. For the 100 algorithm instances, the resource usage of original dynamic designs depends on the

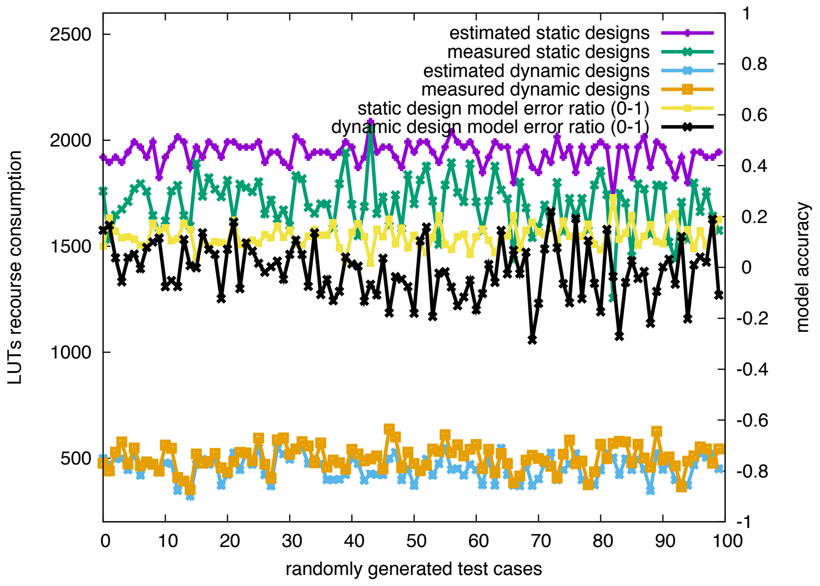


Figure 3.11: Model accuracy of optimised static and dynamic designs for Option Pricing (OP).

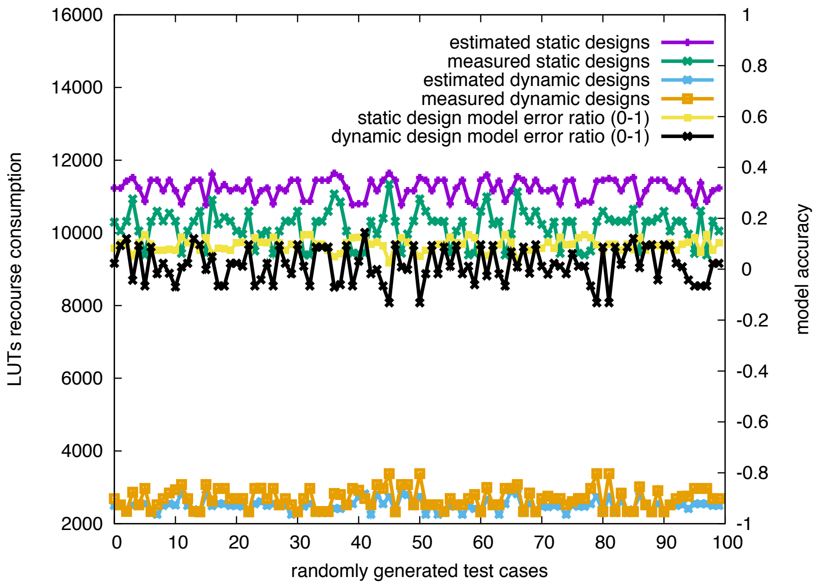


Figure 3.12: Model accuracy of optimised static and dynamic designs for RTM.

initial algorithm parameters. As shown in Figure 3.13 and 3.14, for both static and dynamic designs, the improvement ratio is around 50%. In other words, the circuit-level design approach halves the resource usage of both static and dynamic designs. Compared with the original static designs, the optimised dynamic designs reduce resource usage by up to 6.1 times.

In previous work to optimise finite-difference applications, [BJLW11] applied fixed-point representation for constant operators, and reduced the resource usage for option pricing from 13759

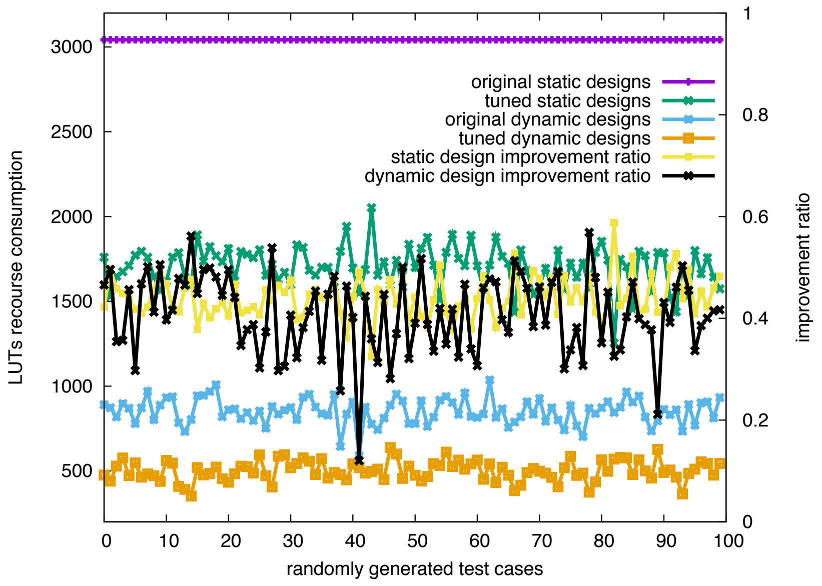


Figure 3.13: Resource reduction of optimised static and dynamic designs for OP.

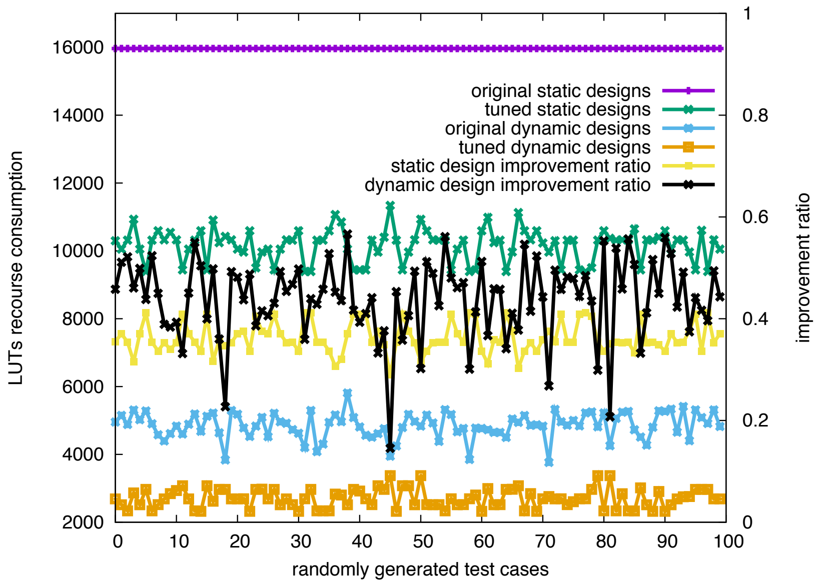


Figure 3.14: Resource reduction of optimised static and dynamic designs RTM.

LUTs for implementations using double-precision operators to 2977 LUTs. The constant coefficients were tuned in [JBLT12b], guided by moment-matching algorithms and synthesised results, the resource usage was further reduced to 710 LUTs. In our work, by selecting constant coefficients preferable to hardware implementations, the resource usage for option pricing is further reduced to 501 LUTs. More importantly, the proposed method enables evaluation of design spaces without going through time-consuming synthesis procedures, which makes it applicable to large-scale designs such as RTM.

3.8.3 Runtime Performance

The runtime performance of the optimised designs is evaluated in two scenarios: (1) the pure throughput when only one algorithm instance is required during runtime, and (2) the overall throughput when an application needs to support multiple algorithm instances, and therefore the dynamic designs use runtime reconfigurations to switch between different algorithm instances. For a single implementation, the runtime performance of original designs is measured from target MAX3424A card. In current implementations, 1000 time steps are propagated for each application, and dimension size is set to be 1024. The runtime performance of optimised static designs and dynamic designs is simulated based on results measured from the target card, as generated VHDL codes are not computable with the compiler of available system (as discussed at the beginning of this section). Both execution time and reconfiguration overhead are included in the runtime performance. The compilation time of static and dynamic designs, on the other hand, does not contribute to the runtime performance, since the tuning and compilation processes are finished before execution. Meanwhile, the increased compilation time for dynamic designs can be reduced by synthesising various algorithm instances in parallel. As an example, the synthesis process for a single RTM kernel takes 21 s to finish, and synthesising 100 dynamic instances in parallel on a 12-core Dell PowerEdge R610 machine takes less than 5 minutes. Since the I/O interfaces of data-paths for original, static and dynamic designs are identical to each other, resources consumed by communication infrastructures R_I (PCI-E drivers and memory controllers) are assumed to be the same.

We summarise the implementation results for OP and RTM in Table 3.2. Each application contains three hardware implementations: original, static and dynamic, and the resource constrains A, infrastructure resource I and data-paths resource usage R_{dp} for each implementation are listed. In addition, the available off-chip memory bandwidth in this reconfigurable system is 38.4 GB/s. For the option pricing application, one datum is read from and written into off-chip memory per cycle per data-path ($M_{bw} = par \cdot bit$). For the RTM, this number

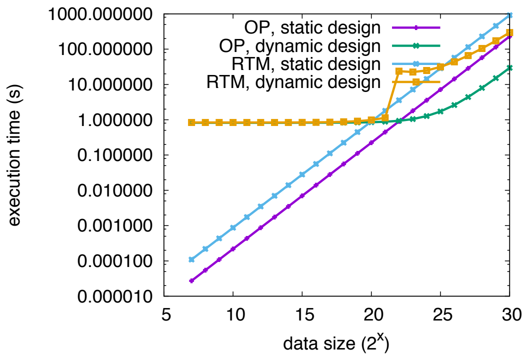
Table 3.2: FPGA implementation results.

	OP			RTM		
	original	static	dynamic	original	static	dynamic
f_{dp} (MHz)	100	100	100	100	100	100
I (LUTs)	29926	29926	29926	34665	34655	34655
A (LUTs)	238080	238080	238080	238080	238080	238080
R_{dp} (LUTs)	3042	2098	501	15964	10926	2702
pd	48	48	48	12	12	12
sd	1	2	8	1	1	6
output data per second (10^9)	4.8	9.6	38.4	1.13	1.13	3.5

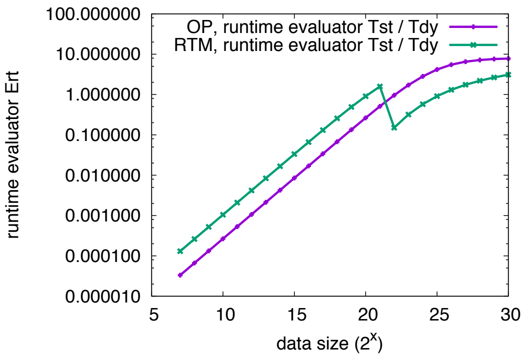
increases to 4 ($M_{bw} = 4par \cdot bit$). Our design model optimise design parallelism $par \cdot tk$ for each implementation. The temporal blocking ratio tk is determined by available resources and data-path resource usage. tk of optimised dynamic option pricing is increased to 8, and tk of optimised dynamic RTM is increased to 6. Table 3.2 presents design throughput in terms of the number of output data items generated per second. For the dynamic design of option pricing, the temporal blocking ratio is 8, with 48 data-paths replicated in each temporal block. At each clock cycle, the dynamic option pricing design generates 384 results. Due to the large number of arithmetic operations in a data-path (i.e. increased data-path resource usage), up to 72 data-paths are implemented for RTM. Moreover, since RTM use three-dimension data structures, increasing temporal blocking ratio increases the overall data size to process, as indicated in Eq.3.26. Therefore, bounded by dm_{ds}, the dynamic RTM design in average generates 29.7 results per clock cycle. Based on the experiment results, the dynamic designs achieve up to 8 times improvement in pure throughput, compared with the original static designs.

In previous work on accelerating RTM, one Blue Gene/Q processes 54 M results per second,[LM13], an optimised CUDA design running on an NVIDIA Tesla C2070 GPU achieves 1.07 G results per second [PF10a, NCJ$^+$13b], and the highest performance number for RTM is 1.62 G results per second on a Virtex-6 SX475T FPGA [NCJ$^+$13b]. Note that both the Tesla C2070 GPU and the Virtex-6 SX475T FPGA are based on 40-nm silicon technology, but the algorithms running on them may not be the same. Without sacrificing any computational precision, the RTM design optimised with the proposed approach is expected to achieve 2.97 G, which is 1.828 times faster than the best published results. It is worth mentioning that R_{knl} of implemented original designs are placed & routed results, while the R_{knl} of optimised static and dynamic designs is measured from post-synthesis results, which means the actual resource usage of optimised designs can be further reduced.

The overall performance of dynamic designs depends on the pure throughput of each configuration, as well as the reconfiguration overhead. Figure 3.15 presents the runtime evaluation results, as problem data size increases. The execution time of static designs increases linearly with data size, since tk is limited to 1 for static designs, and reconfiguration overhead is 0 (see Figure 3.15(a)). For dynamic designs, when data size is small, reconfiguration overhead dominates overall design execution time. When data size is large enough, i.e. beyond 2^{27} for one-dimension option pricing and 512 for three-dimension RTM, dynamic designs start to outperform their static counterparts. Figure 3.15(b) shows the runtime evaluator T_{st}/T_{dy} results. Based on the evaluation results, large speedup can be achieved for using static designs for finite-difference applications with small data size. On the other hand, the dynamic designs improve performance of large-scale applications by 7.8 and 3.01 times for option pricing and RTM, respectively. The fluctuations in T_{st}/T_{dy} are due to the spatial and temporal blocking overhead in data size dm_{ds}.



(a)



(b)

Figure 3.15: (a) Execution time of static and dynamic designs. (b) Run-time evaluation results. For RTM designs, when not all data can be stored on-chip, the design models introduce spatial blocking to split data into smaller blocks. This brings overhead as additional data need to be processed for the blocking. In the current experiments, the data size is 2^{22}.

3.9 Related work

3.9.1 Accelerating Finite-Difference Applications

Driven by the high-performance requirements of finite-difference algorithms, various researchers have worked on accelerating the computation process. One straightforward solution is to distribute workloads into parallel CPU cores. However, data dependencies between distributed workloads, i.e., boundary conditions in finite-difference algorithms, limits the scalability of the parallelised CPU designs. Optimised communication patterns between CPU cores were proposed for Blue Gene/P [PLL^{+}12] and Blue Gene/Q [LM13], achieving 2.99 TFLOPS for Reverse Time Migration (RTM) with a Blue Gene/P rack with 1024 4-core CPUs.

GPUs are widely used to accelerate finite-difference algorithms, as the high on-chip hardware concurrency and memory bandwidth can satisfy the high-performance requirements of finite-difference algorithms. An NVIDIA Tesla C2070 GPU has 448 CUDA cores running at 1.15 GHz, which provides 1.03 TFLOPS peak performance. The challenges for GPU designs are how to efficiently load data from global memory, and how to share loaded among parallel cores. Blocked data access patterns were proposed [Mic09, PF10a] to share accessed data among threads in the same Streaming Multiprocessor (SM). The blocking technique reduces data access redundancy to support high parallelism in GPUs. On the NVIDIA Tesla C2070, optimised GPU designs achieve up to 100.7 GFLOPS for financial pricing and 84.3 GFLOPS for seismic imaging.

FPGAs provide a platform to implement customised memory architectures and data-paths can be implemented. A customised memory architecture which supports two compute units is proposed in [AP^{+}11]. Interconnected soft-processors are mapped into FPGAs to process application workloads in parallel [S^{+}11]. The scalability of the proposed architecture is limited by processed data size: it only works when the accessed data in one cycle are small enough to fit into on-chip memory. A scalable memory architecture was proposed in [FC11a] to support on-chip data access from pipelined data-paths, and an analytical model was proposed in [NJL^{+}12c] to automatically optimise the hardware design. To reduce the resource usage of data-paths, arithmetic operations in finite-difference algorithms are represented with fixed-point format in [BJLW11]. Constant coefficients in the algorithms are used in [JBLT12b] to generate operators customised for specific constant.

Compared with previous work, as discussed in Section 3.8, the simulated performance of optimised finite-difference applications is expected to be up to 1.8 times better than the best published results, including customised designs running in Blue Gene/Q [LM13], NVIDIA C2070 [PF10a, NCJ+13b] and Virtex-6 SX475T FPGA [NCJ+13b].

3.9.2 Circuit-Level Runtime Reconfiguration

Previous work on circuit-level runtime reconfiguration focuses on constant operators and reconfiguration overhead modelling. In [BAS09, BJLW11, JBLT12b], constant operators are mapped into reconfigurable devices for applications with constant operators, and the constant operators are reconfigured when supported constant values change during runtime. The relationship between reconfiguration overhead and execution time is analysed in [EAGEG09], and a reconfiguration overhead model is built in [DML12]. Compared with previous work, we propose a systematic design flow to explore algorithm design space, integrate optimised operators into reconfigurable designs, and to dynamically evaluate runtime performance. Compared with previous work [BJLW11, JBLT12b], the operator resource usage in this work is 1.17 to 2 times smaller. With a performance model that combines optimised design parameters, the proposed evaluator is capable of estimating reconfiguration overhead and execution time during runtime, to execute the reconfigurable design with minimised overall execution time.

3.10 Limitations and Future Work

The proposed circuit-level approach exploits the redundant logic resources in arithmetic operators by customising constant operators as well as tuning algorithm parameters. Results show that customised dynamic designs for Option Pricing and Reverse Time Migration achieve up to 6.1 times reduction in resource usage and 7.8 times improvements in overall design throughput.

The current approach is limited by the fact that the supported applications must have tunable algorithm parameters.

While the constant operator design approach is applicable to all applications with constant operations, the tuning process requires the algorithm constant values can be tuned by varying the algorithm parameters. This leads to two requirements for the supported algorithms. First, the constant values are determined by algorithm parameters. Second, the algorithm parameters have a design space where the parameters can be changed without affecting the algorithm functionality. Besides finite-difference algorithms, various approaches in numerical analysis — such as the Runge-Kutta methods — and signal filtering have tunable parameters. Therefore, the future work will focus on extending our approach to cover other algorithms. The design flow and designs models have been generalised such that algorithm-specific techniques are covered in the domain-specific aspects. In order to extend the current approach, the design space of other algorithms needs to be built to reflect the relationship between algorithm tunable parameters and constant operator properties, and more domain-specific aspects need to be extracted.

3.11 Summary

This chapter explores the runtime reconfiguration opportunities at the circuit level. Arithmetic operators are developed to handle all possible input combinations, which is an overkill for constant operations. Developing customised constant operators will significantly reduce resource usage. However, the constant operators are not supported in either GPPs or ASICs, since even ASIC designs cannot afford only supporting an application using a specific constant set. The use of runtime reconfiguration enables reconfigurable devices to apply the constant operators without losing design generality, by dynamically switching between implemented operators during runtime. Moreover, the constant operators can be further customised by exploring the design space of target algorithms, and selecting the optimal constant set for each design configuration. Experiment results show this approach provides large improvements in design performance compared with conventional reconfigurable designs.

Chapter 4

Function-Level Optimisation for Runtime Reconfigurable Designs

4.1 Introduction

An application often contains more than one function. Inter-function dependencies constrain the execution order of application functions. In order to support an application with multiple functions, a reconfigurable design needs to implement all functions as hardware modules, and activate the hardware modules when the dependent data are ready. More often than not, the dependent data of application functions cannot be ready at the same time, which introduces idle resource units at the function level. In this chapter, idle resource units refer to application functions that become idle during runtime.

Resource sharing and allocation for multicore and manycore processors are usually achieved through thread management at runtime [CGH09]. Such runtime thread management is general purpose, but does not support reorganisation and customisation of computational resources to meet application-specific requirements. Reconfigurable computing supports design customisation at compile time and at runtime. However, such customisation often restricts resource sharing at the function level, since a static design customised to support one function often cannot support a different function.

At the function level, we use runtime reconfiguration to separate functions that are active at different runtime scenarios into various design configurations. For each configuration, as the idle function units are removed, the active functions gain more resources to further improve configuration performance. The major challenges to achieve this design objective include (1) identifying idle application functions, (2) grouping these idle functions based on function idle time and optimising the generated configurations, and (3) linking the optimised configurations as a valid reconfigurable design. The function-level approach addresses these challenges in three steps: function analysis, configuration organisation and partition generation.

Outline. Section 4.2 provides an overview of the function-level approach. Section 4.3, 4.4, and 4.5 respectively present the details of the function analysis, configuration organisation and partition generation design steps. These steps are based on Reconfiguration Data Flow Graph (RDFG), a hierarchical graph structure for analysing and optimising designs. Novel algorithms such as As Timely As Possible (ATAP) assignment method and ending-edge search are proposed to support these design steps. Section 4.6 presents the evaluation approach at the function level to dynamically select the partitions with maximum performance. Section 4.7 discusses the benchmark applications used in this chapter, and Section 4.8 shows the experiment results. Finally, Section 4.9 compares the related work, Section 4.10 discusses the approach limitations, and Section 4.11 summarises this chapter.

4.2 Approach Overview

In order to address the design challenges for exploring runtime reconfiguration at the function level, Reconfiguration Data Flow Graph (RDFG), a new hierarchical design representation, is proposed. We represent application functions as graph nodes, capture I/O operations of connected functions with graph edges, and store algorithm-level details in each function-level graph nodes. In this section, we show the basic idea of this chapter with a motivating example, and then present the design flow of the proposed approach. Finally, we introduce an example application, which is used in the following sections to explain the proposed algorithms.

4.2.1 Motivating Example

In a static design, all functions are mapped into reconfigurable fabrics and the mapped functions are replicated as much as possible to optimise concurrency. However, limited by data dependency and mapping strategies, some computational resources can be left idle from time to time. This situation is shown in Figure 4.1(b): there are four function units, each implementing respectively the function A, B, C and D in the dataflow graph in Figure 1(a). Given that each function takes n cycles, the entire computation would take 4n cycles. It is assumed that the application RDFG indicates each function consumes 1 resource unit, and computation within functions starts once the last output datum of the leading functions becomes available. For t=0..4n-1, several function units would become idle. How could runtime reconfiguration be used to reduce the number of cycles required for this computation?

One possibility involves reconfiguration of the idle function units to perform useful work. Let us assume that there is sufficient data independence in each function to enable linear speedup with additional function units: for k function units, the function takes n/k cycles to complete. So for k=1, it takes n cycles to complete the function as described before, and if k=n, it could potentially only take one cycle, although in practice, k is likely to be smaller than n.

With this assumption, Figure 1(c) shows a design which speeds up computing the functions A and B in the second level of the data flow graph in Figure 1(a) by reconfiguring the two idle function units C and D to A and B. This increase in parallelism means that these functions can be completed in n/2 cycles, during t=n..3n/2-1. For the functions in the third level of the data flow graph, B and C are reconfigured as A and D, finishing computation in A and D in n/2 cycles, during t=3n/2..2n-1. Then the same can be done in computing the last function C in the dataflow graph: this time all four function units are configured to compute C so that it can be completed in n/4 cycles, during t=2n..9n/4-1. The total number of cycles is thus 9n/4, reduced from the 4n cycles for the static design in Figure 1(b). The speedup stems from reconfiguring the resource occupied by the idle functions to generate multiple replications of the active functions, leading to increased parallelism.

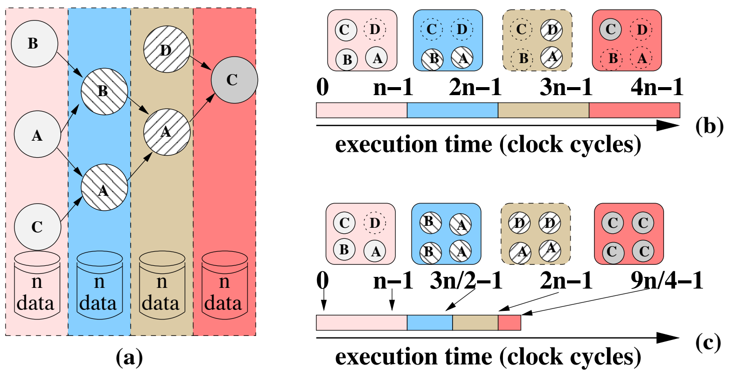


Figure 4.1: Motivating example. The idle function nodes during run time are shaded. (a) Application data flow graph with 4 functions (A, B, C and D), and 8 function instances. Each function has n data items to process. (b) Static implementation, showing which function units are inactive (with dotted boundaries) during t=0 to 4n cycles. The same configuration is executed, consuming n cycles for each frame. (c) Dynamic implementation. An executed configuration only contains functions active in a particular frame. Execution time for a time frame depend on configuration parallelism. As an example, in the second time frame, configuration parallelism is 2 (2 copies of function A and B are implemented), reducing the execution time to $\frac{n}{2}$.

One can observe that in the reconfigurable design above, limited by the reconfiguration granularity, function unit D is inactive from t=0..n-1. If target platforms support finer reconfiguration granularity, the one resource unit can be evenly split between A, B and C; this increase in parallelism would reduce the number of cycles of the first frame from n to 3n/4, so that the total number of cycles for computing the dataflow graph in Figure 1(a) would become 2n.

Of course, the scenario for the motivating example is not realistic; many real-world issues, such as the time required in reconfiguring the function units, are not considered. In the following, we introduce an approach that supports the performance improvement illustrated by this example, while taking into account practical issues in reconfigurable design.

4.2.2 Design Flow

The design flow of the proposed approach is demonstrated in Figure 4.2. The approach starts from an application represented with a hierarchical data-flow graph:

$$A = (G, E_G) \quad G = (V, E) \tag{4.1}$$

where A indicates a function-level graph, and G indicates an algorithm-level graph. G and E_G respectively represent application functions and function I/O operations. Within a function node G, V indicates the arithmetic operations of this function, and E indicates the interconnections between the arithmetic operations.

In order to group and optimise application functions into runtime reconfigurable designs step by step, we build a hierarchy in this work. From bottom to top, a function-level RDFG is divided into segments, configurations, and partitions.

Segments: Function nodes that can be executed without stalling are combined into a segment $S = (G_1, G_2 ...)$. Segments are the basic elements that respect data dependency and expose speedup potential of applications.

Configurations: A configuration $C = (S_1, S_2 ...)$ contains one or multiple segments. A configuration can be synthesised and executed in hardware.

Partitions: A valid partition $P = (C_1, C_2 ...)$ is a combination of configurations that is capable of properly accomplishing the application functionality. The generated partitions for an application are compiled with a host program. In this work, we consider a valid partition as a runtime reconfigurable design.

The proposed approach starts from an application represented as an RDFG, following the design flow in Figure 4.2. The approach contains three compile-time steps and one runtime step. The compile-time steps generate various reconfigurable designs for the target applications. Each reconfigurable design is associated with a specific runtime reconfiguration strategy. The runtime step evaluates the generated reconfigurable designs, to select the design with maximum throughput.

The first step, function analysis, estimates function properties and groups function nodes into segments based on function idle cycles. The second step, configuration organisation, combines segments into configurations, which are optimised to achieve maximum parallelism under available resources. The third step, partition generation, schedules and links the optimised configurations as valid partitions. Basic hardware modules are developed for application functions. We feed the design parameters of the generated partitions (the amount of parallelism, configu-

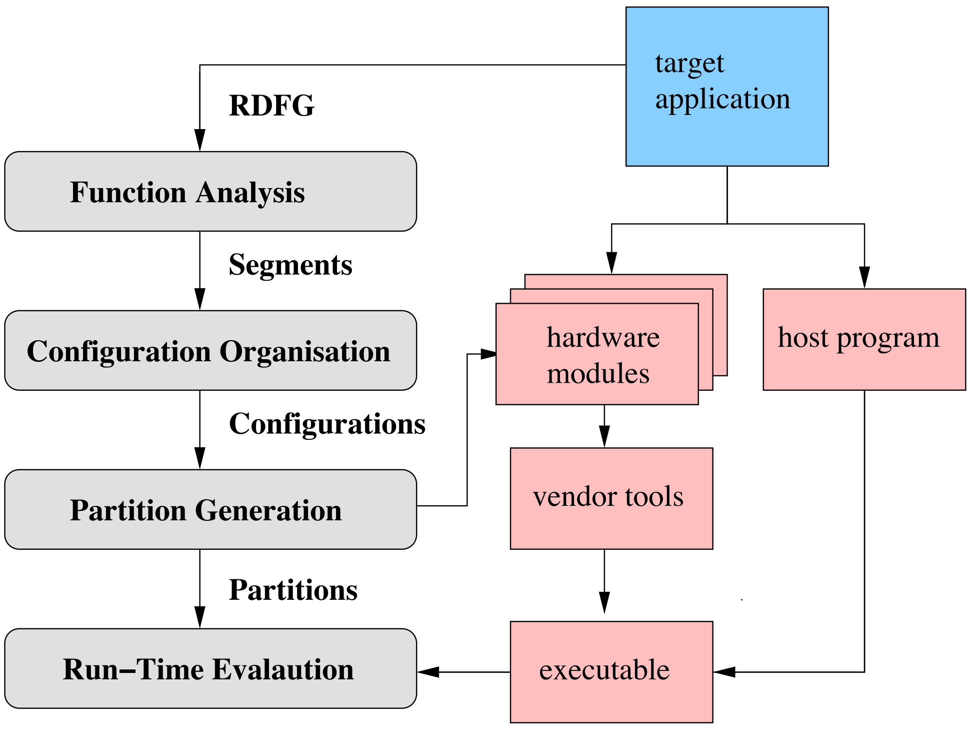


Figure 4.2: Design flow of the proposed approach.

ration organisation, etc.) into the hardware modules. The design parameters of the hardware modules are updated correspondingly. The updated hardware descriptions go through vendor tool chains to generate configuration files, which are compiled with the host program. The fourth step, runtime evaluation, uses a runtime performance model to predict the overall execution time of generated partitions. During run time, the host program selects the partitions with the minimum execution time to download into FPGAs, based on the predicted results. In current approach, the compiler front-end handles RDFG graph extraction and hardware description generation, and the back-end automates function analysis, configuration organisation and partition generation. The host program, on the other hand, is manually developed.

4.2.3 Example Application

Throughout this chapter, we use an example application to demonstrate how an application RDFG is processed step by step to generate reconfigurable designs. Figure 4.3(a) shows the function-level graph of the example application, along with the algorithm-level graph of function

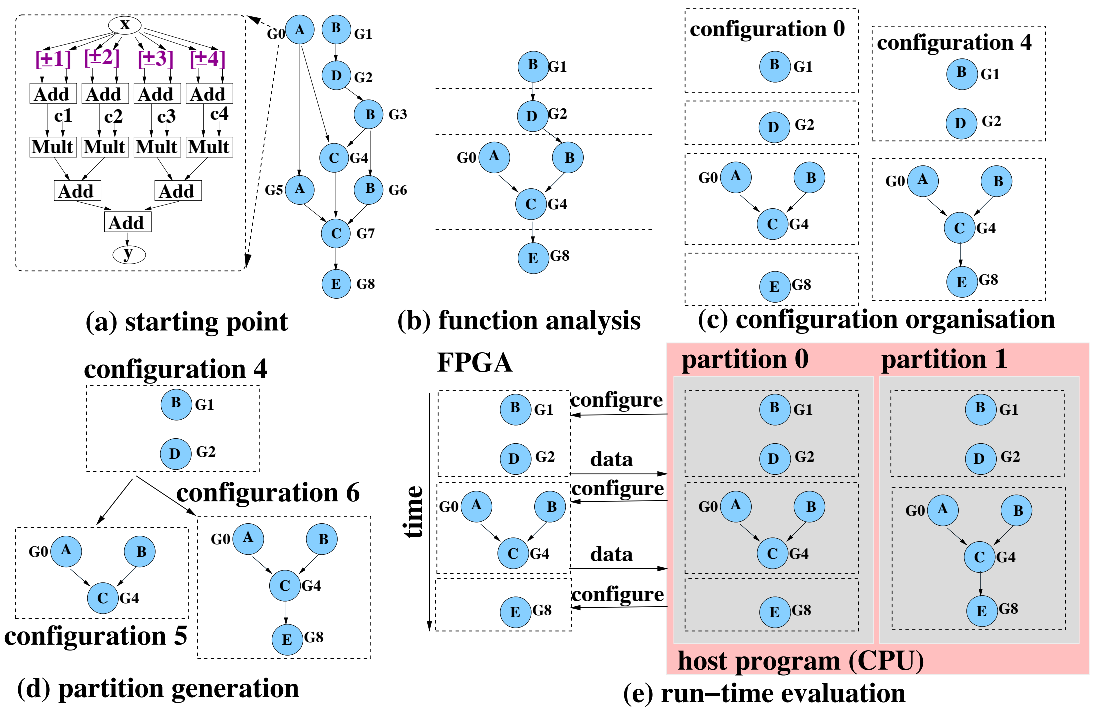


Figure 4.3: An example for the proposed design flow. The example RDFG is shown in (a). Output graphs for function analysis, configuration organisation and partition generation are shown in (b), (c) and (d) respectively. (e) shows the execution of generated partitions (partition 0 is selected in this example). The duplicated segments are removed from the segments, as shown in (b), which is explained in Section 4.4.

node G_0. The processing steps of the example RDFG are summarized as follows. Table 4.1 lists parameters and notations used in these steps.

Function analysis takes the algorithm-level graph of a function node, and estimates resource consumption and idle cycles for the function. Based on analysed idle cycles, we group the functions active at the same time into the same segment. Algorithm details of function node G_0 are shown in Algorithm 5, where x and y are respectively input and output data arrays, and c_j are multiplication coefficients. As shown in Figure 4.3(a), arithmetic operators are mapped as arithmetic nodes, and indices of accessed data are mapped to offset edges. Functions in the same segments can be executed at the same time without stalling, as shown in Figure 4.3(b) with node G_5 merged with G_0 for A, G_6 merged with G_3 for B, and G_7 merged with G_4 for C.

Configuration organisation refers to the combination of function segments and the optimisation of the associated functions. As shown in Figure 4.3(c), a configuration can contain

Algorithm 5 Algorithm detail for function G_0.

```
function $G_0$(float* x, float* y){
for i ∈ (4,n-4) do
    float a1 = x[i-1] + x[i+1];
    float a2 = x[i-2] + x[i+2];
    float a3 = x[i-3] + x[i+3];
    float a4 = x[i-4] + x[i+4];
    y[i] = a1 * c1 + a2 * c2 + a3 * c3 + a4 * c4;
end for
}
```

only one segment, such as `configuration 0`, or it can include multiple segments, such as `configuration 4`. `configuration 0` may achieve higher design parallelism than `configuration 4`, as it requires less hardware resources. On the other hand, the first configuration needs to be reconfigured to execute G_2, which introduces additional reconfiguration overhead compared with `configuration 4`. The objective of configuration organisation is to generate all possible segment combinations, and optimise each of the generated configurations to achieve maximum parallelism.

Partition generation refers to linking optimised configurations as a complete reconfigurable design. As demonstrated in Figure 4.3(d), if `configuration 4` is included in the current partition, to ensure the partition can be executed during run time, the next configuration must include a segment with functions A, B and C. Given this constraint and available configurations, either `configuration 5` or `configuration 6` can be combined into current partition. A searching algorithm is required to select proper configurations to finish the remaining tasks. To reduce the search space, invalid and inefficient configuration combinations are eliminated.

Runtime evaluation refers to the selection of generated partitions during run time. The execution time of a partition depends on configuration properties, reconfiguration time, and runtime data size. While configuration properties and reconfiguration time are known once a partition is generated, data size of the target application remains unknown in compile time. As shown in Figure 4.3(e), `partition 0` achieves higher parallelism since functions in the first and the second time frames are divided into two configurations. As a consequence, more reconfiguration operations are introduced to switch between the configurations. In the current approach, to preserve the data stored in FPGA off-chip memories, the memory data are first transferred back into host memories before a reconfiguration operation. After FPGAs are reconfigured, the stored data are transferred back into FPGA memories, as shown in Figure 4.3(e). For a given data size, if the reduction in execution time outweighs the increase in reconfiguration time, then `partition 0` is selected. A performance model is built to dynamically evaluate design performance when the data size is available. The partition with the minimum execution time is selected and executed.

Table 4.1: Variables and parameters in the function-level approach.

function analysis			
G_i function node i			
$S_{<i,j>}$ function segment at the ALAP level i and the ATAP level j			
L_s	logic resource usage	M_s	memory resource usage
$N_{ari,o}$	number of operator type o in a function	$R_{s,o}$	resource type s consumed for a operator o
mem_{max}	maximum offset value in a function	mem_{min}	minimum offset value in a function
$N_{id,int}$	number of function internal idle cycle	$N_{id,ext}$	number of function external idle cycles
bit	memory bits for one datum	N_{arr}	number of data arrays in a function
configuration organisation			
$C_{<i,j>}$ a configuration that contains $j-i+1$ segments, starting from segment i, ending with segment j			
P	parallelism (number of data-paths)	A	available resources
I	infrastructure resource usage	BW	available bandwidth
N_{in}	number of input edges of a configurations	N_{out}	number of output edges of a configuration
M_{bw}	bandwidth requirement of a configuration		
domain-specific aspects			
Monte-Carlo simulations			
$R_{s,rng}$	resource usage of a RNG in type s		
stencil computations			
sk	spatial blocking ratio	tk	temporal blocking ratio
dm_{ds}	impacts on data size	dm_{Ms}	impacts on memory usage
partition generation			
P_i partition i			
runtime evaluation			
T_i	execution time for segment S_i / partition P_i	O_j	time to reconfigure configuration j
ds	data size	ϕ	throughput of data transfer interface
γ	configuration file size for 1% chip usage	θ	throughput of reconfiguration interface

4.3 Function Analysis

In order to separate functions active at different time intervals, and to duplicate function when there are available resources, we analyse the algorithm details inside a function node to estimate the amount of idle cycles and resource usage. After extracting function details, we schedule function nodes based on (a) interactions between them, as well as (b) function internal idle cycles. A segment S contains function nodes scheduled in the same time frame.

4.3.1 Function Property Extraction

The properties of a function include its resource consumption, its associated data access patterns, and its number of idle cycles. The algorithm-level graph within a function node G_i provides implementation details for the specific function. Fully pipelined data-paths and on-chip memory architectures are constructed to support full resource utilisation of consumed resources, i.e., as long as G_i is active, one data-path for G_i generates one result per clock cycle.

Arithmetic operations within a function are implemented as a pipelined data-path. Within a function node, the resources consumed by arithmetic operations can be estimated as:

$$L_s = P \cdot \sum_{o \in \odot} N_{\mathrm{ari,o}} \cdot R_{\mathrm{s,o}} \cdot B_{s,bit,fix} \quad \odot = \{+, -, *, \div, sta, dyn\} \quad s \in \{LUT, FF, DSP\} \tag{4.2}$$

where L_s stands for the logic resource usage, $R_{\mathrm{s,o}}$ indicates the resource usage of arithmetic operators, including constant operators, and $B_{s,bit,fix}$ accounts the impacts of different data widths. L_s is estimated the same way in the circuit-level model. The resource type s includes LUT, FF and DSP. In other words, the resource usage of LUT, FF and DSP can be estimated with equation 4.2 by specifying proper $R_{\mathrm{s,o}}$.

A function is active once its arithmetic operators start processing data. The number of idle cycles before a function becoming active depends on the number of cycles it takes to get the first input data (i.e., external idle cycle $N_{id,ext}$), and the number of cycles it takes to start processing, once the first input data are available (i.e., internal idle cycle $N_{id,int}$). As an example, for function node G_0, as shown in Algorithm 5, processing of $y[i]$ requires $x[i-4] \sim x[i+4]$. If we assume input data item $x[0]$ in function G_0 is available at cycle n, and the function streams one data item each cycle, the arithmetic operations in the function thus start at cycle $n+9$.

Inside a function node, we analyse memory usage and internal idle cycles based on data offset values. The on-chip memory resources are used to buffer input data, when not all accessed data are available. In Algorithm 5, to calculate $y[i]$, data items before $x[i+4]$ need to be buffered before $x[i+4]$ arrives. The offset edges in Figure 4.3(a) are thus mapped into memory buffers, with the relative position between the maximum and the minimum offsets indicating the buffer size. For a function node G_i, its input nodes are traversed and the offset values are combined into G_i.mem. In mem, mem_{max} and mem_{min} respectively indicate the maximum and the minimum offset values. As an example, there are 8 offset edges for Algorithm 5, as shown in Figure 4.3(a). We thus group the offset edges into mem of G_0 as $[-4, 4]$, where $\text{mem}_{\text{max}} = 4$ and $\text{mem}_{\text{min}} = -4$. A memory architecture buffering 9 consecutive data is generated. The buffered data to calculate $y[4]$ are shown in Figure 4.4(a). In the next cycle, $x[9]$ is streamed into the memory architecture to update buffered data. A data-path connected to the memory architecture can run without stalling. On-chip memory resource used by a function node can be calculated with the relative position as follows. *bit* is the number of bits of one datum, and the resource usage of all accessed arrays in a function N_{arr} is accumulated.

$$M_s = \sum_{a=1}^{N_{arr}} (\text{mem}_{\text{a,max}} - \text{mem}_{\text{a,min}} + 1) \cdot bit \tag{4.3}$$

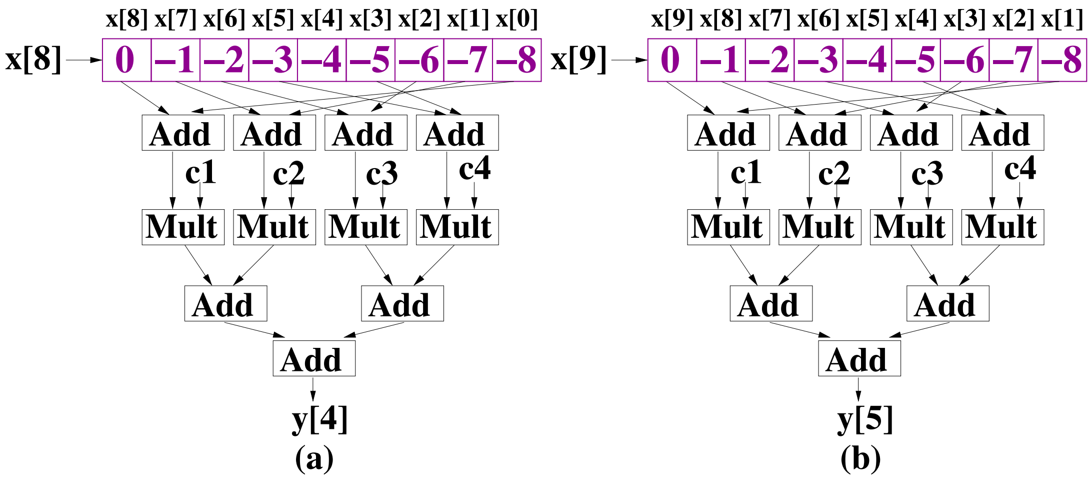


Figure 4.4: Data buffering for offset edges in Algorithm 5, to calculate (a) $y[4]$ and (b) $y[5]$.

$N_{id,int}$ indicates the number of cycles that arithmetic operators in a function have to wait after the first input datum is available. This normally happens when the arithmetic operations in a function depend on more than one datum. For the example in Figure 4.4, the computation depends on 9 data, and cannot start when x[0] is available. The number of cycles a function needs to wait depends on the distance between the required data, i.e., the distance between

$\mathrm{mem}_{\mathrm{min}}$ and $\mathrm{mem}_{\mathrm{max}}$. Since $\mathrm{mem}_{\mathrm{min}}$ and $\mathrm{mem}_{\mathrm{max}}$ can be either positive or negative, $N_{id,int}$ can be expressed as:

$$N_{id,int} = \max\left(\mathrm{mem}_{\mathrm{a,max}} + \frac{|\mathrm{mem}_{\mathrm{a,min}}| - \mathrm{mem}_{\mathrm{a,min}}}{2} + 1\right) \forall a \in \{1...N_{arr}\} \tag{4.4}$$

When $\mathrm{mem}_{\mathrm{min}}$ is less than 0, such as -4 in Algorithm 5, the minimum offset edge points at the first input datum. $N_{id,int}$ is the number of cycles to buffer data in mem, i.e., $\mathrm{mem}_{\mathrm{max}} - \mathrm{mem}_{\mathrm{min}} + 1$. When $\mathrm{mem}_{\mathrm{min}}$ is above 0, for example, if we add 100 to all data indices in Algorithm 5, the minimum offset edge (96) points to the 96th data after the first input datum. In other words, the computation in this function starts until the 96th cycles. The $\mathrm{mem}_{\mathrm{min}}$ is added into $N_{id,int}$ to take the initial delay into account. $N_{id,int}$ therefore is expressed as $\mathrm{mem}_{\mathrm{max}} + 1$. The idle cycles introduced by each array a is compared, and the idle cycle of a function is defined as the maximum of the idle cycles.

4.3.2 Segment Generation

A segment S_i includes function nodes that are active at the same time. We use external idle cycles and internal idle cycles to classify application functions: functions with the same $N_{id,ext}$ and $N_{id,int}$ are grouped into the same segment, indicating these functions can be activated at the same time. $N_{id,ext}$ of a function depends on the execution status of its predecessor functions, which can only be properly estimated once complete reconfigurable designs (partitions) are generated. In this stage, the design objective is to differentiate functions active at different time intervals. As-Late-As-Possible (ALAP) levels are assigned based on function-level edges E_G. Functions that depend on the same input data would have the same external idle cycles, i.e., the same ALAP levels. Within the same ALAP level, internal idle cycle count $N_{id,int}$ is used to further separate functions with different offset values. As an example, if another function node G_x starts its computation once $x[0]$ is available, G_x and G_0 are active at different cycles while they share the same $N_{id,ext}$. To demonstrate the segment generation process, we use the RDFG in Figure 4.5 as an example. $N_{id,int} = N$ for functions A and B, and $N_{id,int} = M$ for

Algorithm 6 As Timely As Possible Assignment. The algorithm merges functions that start at the same time into one segment.

```
input: $G$, function nodes assigned with ALAP levels
output: $S$, generated segments
  for $G_i \in G$ do
    $G_i$.atap $\leftarrow$ $G_i.N_{id,int}$
    for $G_j \in G_i$.outputs do
      if $G_j$.alap = $G_i$.alap + 1 then
        if !$G_j.N_{id,int}$ then
          $G_j$.alap $\leftarrow$ $G_j$.alap - 1
          $G_j$.atap $\leftarrow$ $G_i$.atap
        end if
      end if
    end for
    $S_{<G_i.alap,G_i.atap>}$.add($G_i$)
  end for
```

function D. Arithmetic operations in function C are:

$$\forall i \in (0, n) \quad z[i] = x[i] * y[i] \tag{4.5}$$

where computation starts as soon as input data are ready, i.e., $N_{id,int} = 0$.

In order to simplify context saving and recovery operations, we assign ALAP levels [GDWL92] to function nodes. Various scheduling algorithms have been proposed to ensure correct execution of nodes in a graph [PB99, HLH+98]. As full-reconfiguration is used in the present method, the communication between consecutive configurations in a reconfigurable design is not affected by reconfiguration: output data of the current configuration are transferred from local memories into host memories before reconfiguration takes place, and from host memories to local memories after reconfiguration, as shown in Figure 4.3(e). For the example in Figure 4.5(a), if scheduled As-Soon-As-Possible, function node G_0 will be executed once the application starts. The output data of G_0, on the other hand, are only used when G_4 is executed. Complex memory control is required to store and transfer the output data of G_0 properly. By assigning ALAP levels, we ensure only output data of the previous configuration need to be transferred, as shown in Figure 4.5(b).

Inside an ALAP level, function nodes with different $N_{id,int}$ are further separated into different levels, named As-Timely-As-Possible (ATAP) levels. The ATAP level of a function node is

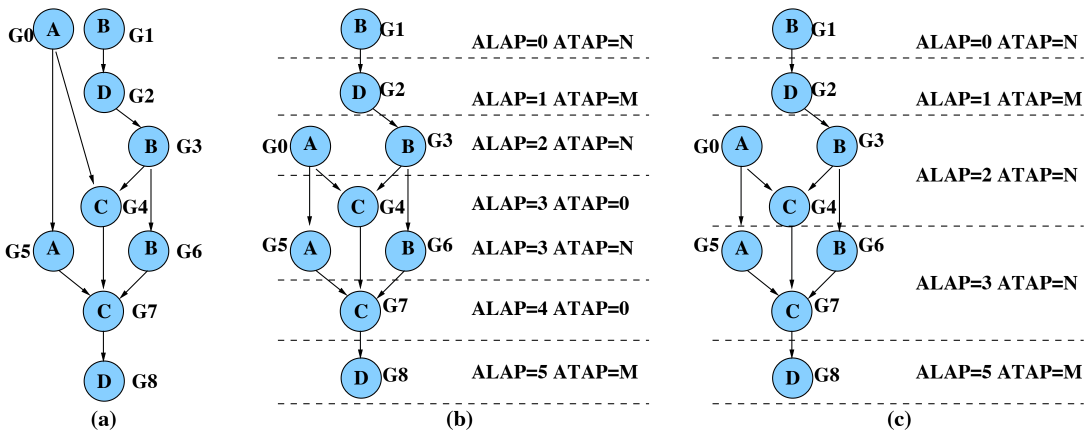


Figure 4.5: (a) RDFG of the example application. (b) RDFG after assigning the initial ALAP and ATAP levels. (c) Generated segments based on assigned ALAP and ATAP levels.

assigned with respect to its $N_{id,int}$ (line 2 of Algorithm 6). After assigning the ATAP levels, there are three scenarios to consider. (1) Inside the same ALAP level, function nodes with the same ATAP levels can run in parallel at the same time. Therefore these nodes are assigned to the same segment. (2) A function node with $N_{id,int} = 0$ indicates its arithmetic operations can start as soon as input data from previous ALAP level are ready (line 5). Implemented in hardware, such a function node can be pipelined with functions in its previous ALAP levels. As shown in Figure 4.5(b), G_4 is dependent on G_0 and G_3. In hardware, G_4 is implemented as a multiplier, with its offset edges mapped as on-chip wires. The multiplier can be merged into the data-paths of G_0 and G_3. Therefore, G_4 can be executed at the same time as G_0 and G_3. In the scheduling algorithm, ALAP level of G_4 is reduced by 1, and its ATAP level is assigned as N, indicating it starts once G_0 and G_3 start (line 6∼7). (3) Function nodes with different ALAP levels or ATAP levels are assigned to different segments (line 11), since these functions will be active in different time intervals.

4.4 Configuration Organisation

After function-level RDFG is divided into segments, operations at the configuration level include distributing segments into different configurations and optimising each configuration to fully utilise available resources. A configuration is expressed as $C_{<i,j>}$, where i indicate the starting

Algorithm 7 Configuration generation. The algorithm enumerates all legal sequences of segments over the sets of compressed segments.

Input: compressed segments S=(S_0, S_1...)
Output: all valid configurations C=(C_0, C_1...)

```
1: for i = 0 → S.size do
2:     $C_{buf} \leftarrow \emptyset$
3:     for j = i → S.size do
4:         $C_{buf}$.add($S_j$)
5:         $C_{<i,j>} \leftarrow C_{buf}$
6:     end for
7: end for
```

segment, and j implies the ending segment. Therefore, $C_{<i,j>}$ contains $j - i + 1$ segments.

4.4.1 Configuration Generation

Ideally, every segment can be considered as a configuration, and design inefficiency can be eliminated by dynamically reconfiguring segments. For the generated segments in Figure 4.6(a), the 5 segments can be mapped into 6 separated configurations, which are configured and executed as scheduled. Theoretically, optimal performance is achieved as no idle cycle is introduced. In practice, such a configuration generation scheme introduces two problems. First, there are configurations with the same function nodes. As shown in Figure 4.6(a), $S_{<2,N>}$ and $S_{<3,N>}$ share the same functions. One configuration is capable of accomplishing the functions of the two segments. Separating them into two configurations introduces reconfiguration overhead. Second, large reconfiguration overhead makes this scheme impractical. In this approach, we use full reconfiguration to switch between different configurations; the reconfiguration overhead includes the time to configure the FPGA and the time to preserve computational context. If we generate one configuration for each segment, we introduce frequent reconfigurations. When the number of eliminated idle cycles is less than the reconfiguration overhead, overall performance is reduced. In order to generate reconfigurable designs with the minimum overall execution time (including the execution time and the reconfiguration time of configurations), we generate all valid configurations from segments. During run time, the configurations are selected based on data size and reconfiguration overhead.

Design rules for configuration are introduced to reduce complexity for generating valid configurations. Combining segments into configurations is a combinatorial problem where all subsets of of segments $(S_1, S_2, S_3 ...)$ are generated. However, the number of combinations can easily become too large to process when the graph size increases. We introduce two design rules, to remove redundant and invalid segment combinations.

Rule 1: Consecutive segments with the same functions are defined as duplicated segments. The duplicated segments are removed to leave just one such segment. In hardware, the duplicated segments can be executed with the same hardware modules. Distributing these segments into different configurations cannot provide better runtime performance. For example, as shown in Figure 4.6(a), $S_{<2,N>}$ and $S_{<3,N>}$ share the same functions, and can be assigned to different configurations. If we assign $S_{<1,N>}$, $S_{<2,N>}$ and $S_{<3,N>}$ to the same configuration, when the configuration is executing the functions in $S_{<2,N>}$, only $S_{<2,N>}$ in this configuration is active. The hardware modules (A, B, C) in $S_{<3,N>}$ remain idle since these modules depend on the output of $S_{<2,N>}$, although these two segments share the same hardware functions. By removing the duplicated segments, we eliminate such inefficient configurations. Moreover, we reduce the search space to generate configurations. For large-scale applications, the same functions can be iteratively called thousands of times. The removal of duplicated segments can significantly reduce the complexity of generating configurations.

Rule 2: As function segments are arranged according to data dependency levels, only configurations with consecutive segments are considered as valid. In Figure 4.6(b), a configuration that contains S_0 and S_3 is considered as an invalid configuration. If such configuration is downloaded into an FPGA, either S_0 or S_3 would stall: when S_0 is executed, S_3 remains idle as it needs output data from S_2; when S_3 is executed, the function in S_0 has been accomplished. For a configuration with consecutive segments, it respect data dependencies between involved segments.

While *Rule 1* reduces the number of segments, *Rule 2* defines which segments can be combined into one configuration. Algorithm 7 (line 1 to 3) searches segments in a consecutive manner, from source nodes to segments assigned the maximum levels, and each valid combination is stored as a configuration (line 4 and 5). As shown in Figure 4.6(c), configuration $C_{<0,0>}$ indicates that a configuration starts from segment 0, and contains 1 segment. Similarly, $C_{<0,3>}$

contains all 4 segments, starting from segment 0. After generating all configurations that start from the first segment, the algorithm restarts the process from the second segment (line 1).

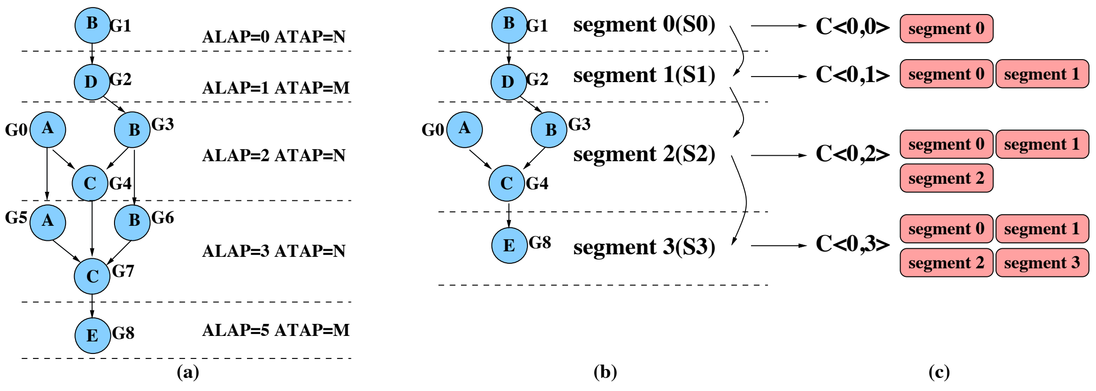


Figure 4.6: (a) Segments generated from function analysis. (b) Compressed segments based on Rule 1. (c) Generated configurations that start from the first segment S_0, following the combination order defined by Rule 2.

4.4.2 Configuration Optimisation

With functions active at different time intervals distributed into different configurations, hardware resources occupied by the idle functions are freed. The freed resources are utilised by optimising each configuration. Required resources are first extracted from the segments in a configuration, and relevant functions are replicated to fully utilise available resources.

The required resources include hardware resources and bandwidth requirements. As all arithmetic operators in data-paths run concurrently, consumed resources cannot be shared. Therefore, in a configuration, resource consumed on data-paths can be directly accumulated as follows, where C is the target configuration, S and G are respectively all segments and function nodes included in C, and $N_{\mathrm{G,o}}$ is the number of operations of type o in function node G, the LUT resource usage Ls is given by:

$$L_s = P \cdot \sum_{S \in C} \sum_{G \in S} \sum_{o \in \odot} (N_{\mathrm{ari,o}} \cdot R_{\mathrm{s,o}} \cdot B_{s,o,bit,fix}) \quad \odot = \{+, -, *, \div, sta, dyn\} \quad s \in \{LUT, FF, DSP\} \tag{4.6}$$

On-chip memories, on the other hand, can be shared by replicated functions. As an example, for the function node G_0 in Figure 4.3, if two data-paths are implemented, $\mathrm{mem_i} \cup \mathrm{mem_{i+1}}$ only increases from [1,9] to [1,10]. Instead of doubling the memory resource usage, implementing one more data-path only requires one more datum to be buffered. Figure 4.7 demonstrates the additional memory usage, when the arithmetic operators are duplicated. For a function with parallelism P, its memory space can be updated as the union of buffered data mem $= \bigcup_{i=1}^{P} \mathrm{mem_i}$. Besides additional memory storage resources, more memory I/O ports are required to run the duplicated arithmetic operators in parallel. We use $\mathrm{mem_{edge}}$ to indicate the number of edges in a segment, and $\mathrm{N_{port}}$ to indicate the number of I/O ports for a BRAM. Therefore the memory resource usage for a segment is determined by the maximum value of I/O bounded BRAM usage ($\frac{\mathrm{P \cdot mem_{edge} + P}}{\mathrm{N_{port}}}$) and storage bounded BRAM usage ($(\mathrm{mem_{max}} - \mathrm{mem_{min}} + 1) \cdot R_{\mathrm{M}}$). The memory resource usage for a configuration C can then be accumulated as:

$$M_s = \sum_{S \in C} \sum_{G \in S} \max\left(\frac{\mathrm{P \cdot mem_{edge} + P}}{\mathrm{N_{port}}}, \sum_{\mathrm{a=1}}^{\mathrm{N_{arr}}} \sum_{\mathrm{d=1}}^{\mathrm{P}} (\mathrm{mem_{a,d,max}} - \mathrm{mem_{a,d,min}} + 1) \cdot \mathrm{bit}\right) \tag{4.7}$$

Besides resources consumed by data-paths and memory architectures, communication infrastructures consume resources for connecting on-chip memory architectures to off-chip data ports. The consumed LUTs, FFS, DSPs and BRAMs are respectively labelled as I_{LUT}, I_{FF}, I_{DSP} and I_{Ms}, and considered as constant parameters for each configuration.

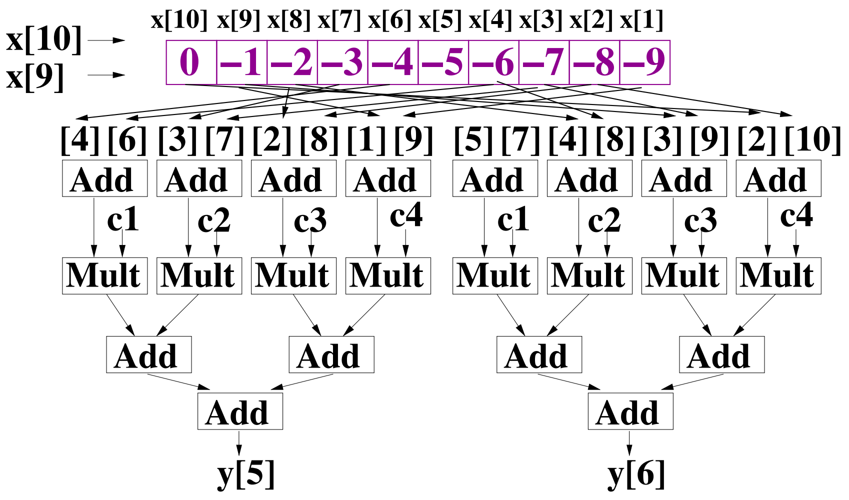


Figure 4.7: Data buffering for offset edges in Algorithm 5, to calculate two results per cycle ($y[5]$ and $y[6]$).

The bandwidth requirement M_{bw} depends on the number of input/output edges of a configuration. The number of input edges N_{in} and output edges N_{out} of a configuration can be updated by searching all edges in the configuration. As only edges not connected to internal function nodes would involve memory access, an input edge is considered as an input edge of a configuration if its input node is not included in the configuration. Similarly, if an output edge is pointing at function nodes outside its configuration, it is included in the configuration output edges. M_{bw} can then be expressed as:

$$M_{bw} = P \cdot (N_{in} + N_{out}) \cdot f_{dp} \cdot dw \tag{4.8}$$

where f_{dp} is the data-path operating frequency, and dw is the width of represented data.

After collecting configuration properties, a similar optimisation model as the circuit-level model is applied to maximum the design parallelism. Since functions active at different time are separated into different configurations, the optimisation model reuse the resources previously consumed by idle functions to replicate more active functions.

$$\textbf{minimise: } \frac{ds \cdot dm_{ds}}{P \cdot f_{dp}} \tag{4.9}$$

subject to:

$$L_{LUT/FF/DSP} \cdot par \cdot dm_{LUT/FF/DSP} + I_{LUT/FF/DSP} \leq A_{LUT/FF/DSP} \tag{4.10}$$

$$M_s \cdot dm_{Ms} + I_{Ms} \leq A_{MS} \tag{4.11}$$

$$M_{bw} \cdot dm_{BW} \leq BW \tag{4.12}$$

where L_s is the configuration logic resource usage, M_s is the memory resource usage, M_{bw} is the bandwidth requirements, and I is the constant resource usage for communication infrastructures. Bounded by the available resources A, the optimisation objective is to achieve the maximum throughput for each configuration. To preserve generality of this approach, the optimisation problem is simplified. Application-specific optimisation techniques can be applied to further improve configuration performance.

4.4.3 Domain-Specific Aspects

In general, the function-level approach is applicable to applications with idle functions. The more complex applications are (more function nodes, complex inter-function dependencies), the more design space the approach gains. In the this chapter, we explore two application domains: Monte-Carlo simulations and stencil computations. The finite-difference applications in Chapter 3 can be considered as one type of stencil computations.

Monte-Carlo Simulations

Monte-Carlo simulations are a class of algorithms based on randomisation. Given an unknown probabilistic data characteristics, the same simulation is ran many times with random variables to capture the result data distribution. The computation depends on parallel random trials to statically coverage the results. Monte-Carlo simulations are widely used in scientific and financial modelling, and are often computationally expensive to implement. To produce accurate simulation results, thousands to millions of simulation runs need to be executed, where each simulation run goes through the same arithmetic operations with different random variable values. Monte-Carlo simulations are inherently parallel since there are no data dependencies between the simulation runs. The computationally intensive operations of the Monte-Carlo application involve generating random numbers and executing parallel simulation paths.

Random Number Generators (RNGs) play an important role in Monte-Carlo simulations as running the simulations in parallel requires a large number of random data at each clock cycle. In Monte-Carlo simulations, the data-paths and memory architectures are mapped into FPGAs with the general design models. The RNGs are labelled with pragmas in the high-level descriptions, and are represented as a special node in the intermediate DFGs. In this work, we use the piecewise fixed-point linear generation method [TL06], and directly map the RNG nodes into hardware. The edges between RNG nodes and following arithmetic nodes are implemented as wire connections. In addition, since the RNG only needs to be initialised at the beginning of computations, and does not require input data during runtime, the RNG nodes are not involved in estimating the off-chip memory bandwidth requirements M_{bw}. We collect the resource usage of RNGs from synthesised results, and consider the resource usage as constant when estimating data-paths resource usage L_s.

Stencil Computation

Stencil computation refers to a class of iterative operations to update array data with a fixed pattern, named as a stencil. Stencil computations are commonly used in simulating dynamic systems, such as fluid dynamics and heat diffusion, as well as in solving Partial Differential Equations (PDEs). Algorithm 8 shows an example application with 3-D stencil. The 3-D data structure is shown in Figure 4.8(a). Since neighbouring data are required to support the calculation, boundary data are not updated during computation, named as halo data. The simulated time dimension is discretised into nt time steps. In time step t, the constructed stencil sweeps over kernel data to propagate $f(s,t)$ in time dimension, and the system status in $t+1$ is simulated based on the results in time steps t and $t-1$.

Algorithm 8 An example of a stencil code.

```
1: for $t \in 0 \rightarrow nt$ do
2:   for $z \in 1 \rightarrow nz-1$ do
3:     for $y \in 1 \rightarrow ny-1$ do
4:       for $x \in 1 \rightarrow nx-1$ do
5:         $f_{[t+1][z][y][x]} = (f_{[t][z][y][x-1]} + f_{[t][z][y][x+1]}) * \alpha$
6:                     $+ (f_{[t][z][y-1][x]} + f_{[t][z][y+1][x]}) * \beta$
7:                     $+ (f_{[t][z-1][y][x]} + f_{[t][z+1][y][x]}) * \gamma$
8:                     $- f_{[t-1][z][y][x]}$;
9:       end for
10:    end for
11:  end for
12: end for
```

For stencil computations, similar to the finite-difference algorithms in Section 3.5.2, we support temporal and spatial blocking. Figure 4.8 demonstrates the impacts of the blocking techniques. As shown in Figure 4.8(b), for a 3-D stencil application, blocking the lowest two dimensions (x and y) in half reduces data distance between neighbouring data at the highest dimension (z) by 75%, which allow four parallel cores to process data blocks with improved data locality. When performance of parallelised designs is bounded by memory bandwidth, temporal blocking is used to propagate multiple time steps with one memory pass. As shown in Figure 4.8(c), propagating stencil data for time steps t and t+1 can be accomplished by either executing the unblocked designs twice, or buffering the intermediate results on-chip to eliminate the redundant memory access operations. The domain-specific aspects can be summarised as follows

$$P = par \cdot tk \tag{4.13}$$

$$dm_{MS} = \prod_{D=1}^{N_D - 1} \frac{n_D}{\frac{n_D}{sk_D} + 2 \cdot w_D \cdot tk} \tag{4.14}$$

$$dm_{ds} = \prod_{D=1}^{N_D} \frac{n_D + 2 \cdot w_D \cdot sk_D \cdot tk}{n_D} \tag{4.15}$$

where P indicate the overall parallelism, dm_{MS} accounts for the reduction in memory usage, and dm_{ds} accounts for the increase in the overall data. After spatial and temporal blocking, $2 \cdot w_D$ additional data (one layer of halo data) are introduced in dimension D when the spatial blocking ratio sk increases by one, and as the temporal blocking ratio tk increases by one, one layer of halo data are introduced in all dimensions, and par more data-paths are implemented.

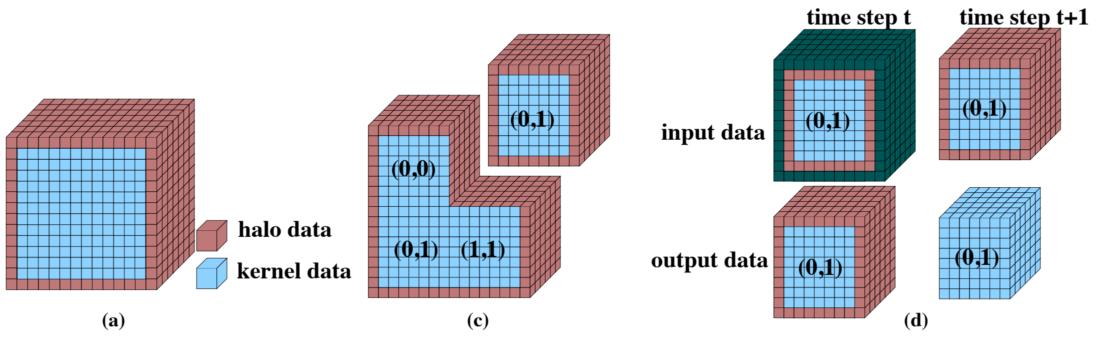


Figure 4.8: Data organisation of (a) an original 3-D stencil problem, (b) after spatial blocking, and (c) after spatial and temporal blocking.

4.5 Partition Generation

A valid partition consists of a combination of configurations that respects data dependencies and does not have redundant functions. Optimised configurations are combined into a partition as a complete reconfigurable design. During run time, an FPGA is dynamically configured following a specific order determined at compile time. As shown in Figure 4.9, partition P_n contains configurations $C_{<0,2>}$ and $C_{<3,3>}$. The search algorithm finds $C_{<0,2>}$ first and then combines $C_{<3,3>}$ into P_n. During run time, a host program downloads configurations based on the order of combination. The host program first configures $C_{<0,2>}$ into the available FPGAs. When $C_{<0,2>}$ finishes its function operations, the host program then reconfigures FPGAs with $C_{<3,3>}$ to finish the remaining functions.

Similar to the configuration generation process, random combinations will generate invalid designs. Several rules for partition generation are applied to construct the search space.

Rule 3: Data dependencies between configurations are implied by the combined segments. Configurations must be included into partitions in a way that ensures segments with lower data dependency level finish first. For configurations generated from segments in Figure 4.9(a), combining configuration $C_{<0,2>}$ as the first configuration in P_n indicates $C_{<0,2>}$ will be executed first. As function node G_1 (segment 0) will be instantiated first in the target application, $C_{<0,2>}$ needs to contain segment 0 to ensure correct execution. In other words, configurations starting from segment 0 ($C_{<0,0>} \sim C_{<0,3>}$) need to be combined into a partition first. This requires the search process to start from configurations including segments with the lowest level.

Rule 4: As a complete reconfigurable design, the generated partitions must be capable of accomplishing the target applications. To finish the example application in Figure 4.3(a), all A, B, C, D and E functions must be contained in a partition. As function nodes are grouped as segments, this requires that a partition contains all function segments. As an example, if all configurations in a partition do not include S_3, the partition cannot finish the application in Figure 4.9(a). This requires all compressed segments must be included in a valid partition.

Rule 5: To ensure hardware efficiency, configurations with overlapped segments cannot be combined into the same partition. Otherwise, the same functions will be implemented multiple times, introducing redundant hardware.

$$\forall (C_i, C_j) \in P_i \quad C_i \cap C_j = \emptyset$$

For the application in Figure 4.9(b), if P_n contains $C_{<0,2>}$ and $C_{<2,3>}$, S_2 is included in both configurations. When S_2 is executed, only one of the configurations is downloaded into FPGAs. The S_2 in the other configuration is never activated, introducing hardware inefficiency.

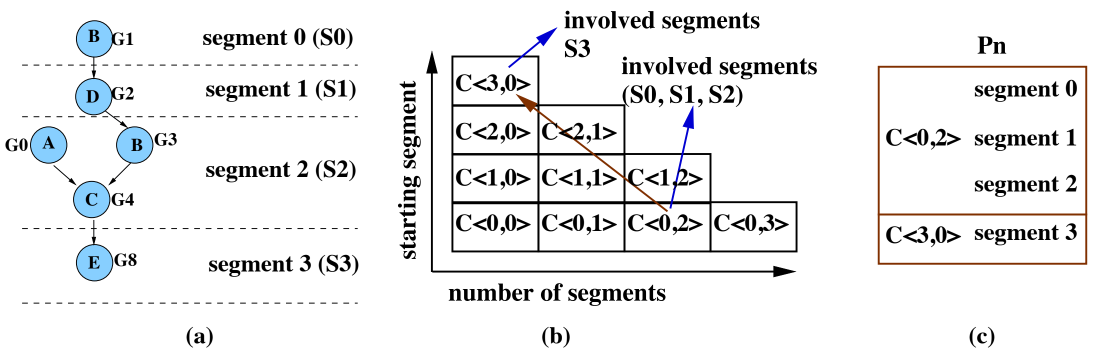


Figure 4.9: (a) Compressed segments from configuration organisation. (b) Generated configurations in a configuration map. The example search operation starts from $C_{<0,2>}$, and looks for the remaining configurations. (c) A valid partition P_6 with configurations $C_{<0,2>}$ and $C_{<3,3>}$. (d) All valid partitions for the example application.

We search valid partitions recursively, as shown in Algorithm 9. The starting point of each search operation is defined in the main function. The present partition and the starting point for the next search operation are passed into the search function `Find_Partition` (line 10 and 21 in Algorithm 9). If the search function finds a valid partition, it returns the partition. Otherwise the search function recursively calls another search function. The rules listed above define the initial starting point in the main function, the starting point for the next search operation, and the ending point for a partition search.

Rule 3 defines the initial starting point of the search operations. We organise the generated configurations in a configuration map, as shown in Figure 4.9(b), where the y axis indicates the starting segment of the configuration (i in $C_{<i,j>}$), and the x axis indicates the number of segments in this configuration (j in $C_{<i,j>}$). The search process begins from the starting point with configurations in the first row in Figure 4.9(b). This is ensured by the first line of Algorithm 9. In this example, we pick $C_{<0,2>}$ as the first configuration. It contains 3 segments (S_0, S_1, S_2).

Rule 5 defines the starting point for the next search operation. Given the current configuration $C_{<0,2>}$ contains segments (S_0, S_1, S_2), the next configuration should start from S_3 to prevent overlapping with segments in existing configurations. Therefore, the next search operation finds configurations in the fourth column of the configuration graph (line 10). Since the starting point of the next search operation depends on the ending segment of the last configuration in the current partition P_{buf}, we name the algorithm as ending-segment search algorithm.

Algorithm 9 Ending-Segment Search Algorithm. The algorithm searches all valid combinations of configurations that are capable of accomplishing application functionality.

```
1: **Find_Partition**($P_{buf}$, $start$) {
2: num_segments : number of compressed segments in $S$.
3: i $\leftarrow$ start
4: for j = i$\rightarrow$ (num_segments-1) do
5:     $P_{buf}$.add($C_{<i,j>}$)
6:     if j == (num_segments -1) then
7:         Partitions.add($P_{buf}$)
8:         return
9:     else
10:        Find_Partition($P_{buf}$, j+1)
11:    end if
12:    $P_{buf}$.pop($C_{<i,j>}$)
13: end for
14: return
15: }
16:
17: main() {
18: $P_{buf}$ $\leftarrow$ $\emptyset$;
19: Partitions $\leftarrow$ $\emptyset$;
20: start $\leftarrow$ 0;
21: Find_Partition($P_{buf}$, start);}
```

Rule 4 defines the ending point of the search operation. As a valid partition contains all segments, once the search algorithm finds out that all segments are included in the current partition, it returns the current partition. After a configuration is found, the search algorithm checks whether all segments have been included, by comparing the number of segments *num_segment* with the ending segment of the last configuration in P_{buf} (line 6). In this example, the search algorithm finds $C_{<3,3>}$ in the fourth column of the configuration map. $j = 3$ indicates that all segments have been included in current partition. Current search operation is terminated, and the partition is saved as a valid partition (line 7∼8).

4.6 Runtime Evaluation

The performance of the generated partitions depends on the application characteristics, design properties and data size. Application characteristics and design properties are available during compile time, and thus their impacts on partition performance can be analysed before execution. The data size of application functions, on the other hand, can either be hard-coded as

Algorithm 10 Partition Scheduling Algorithm. The algorithm estimates the execution time of generated partitions.

Variables: v_i: nodes, p_i: partitions, Cur: current configuration
Functions: Conf(v_i, p_i): find configuration c_i in partition p_i that $v_i \in c_i$

```
for $p_i \in$ Partitions do
    for $v_i \in$ Source Nodes do
        Cur $\leftarrow$ Conf($v_i$, $p_i$)
        while $v_i$.NextNode $\neq \emptyset$ do
            if $v_i \notin$ Cur then
                Cur $\leftarrow$ Conf($v_i$, $p_i$)
                $p_i$.T += Cur.$C_{re}$ + Cur.$C_m$
            end if
            $p_i$.T += Cur.$C_t$
            $v_i \leftarrow v_i$.NextNode
        end while
    end for
end for
```

static constants or be dynamically specified during execution. If data sizes are implemented as compile-time coefficients, performance of each partition can be determined during compile time, and the optimal partition with maximum performance can be selected before execution. However, such a static approach is only applicable to applications with deterministic data sizes. A runtime performance model is introduced in the proposed approach. The execution time and the reconfiguration overhead of partitions are estimated based on data sizes, with constant coefficients indicating application characteristics and design properties.

The constant coefficients for the performance model can be extracted by traversing the uncompressed segments with the generated partitions, as shown in Algorithm 10 (line 4-11). For each segment, the current partition is searched to find a configuration with all segment functions included (line 3 in Algorithm 10). The configuration is named as the current configuration. Since functions in a segment can be executed in parallel, the execution time for a segment S_i can be expressed with segment data size ds_i, configuration parallelism P and data-path frequency f_{dp}.

$$T_i = \frac{ds_i}{P \cdot f_{dp}} \tag{4.16}$$

Design parallelism P and operating frequency f_{dp} are statically configured in each configuration, and are updated when reconfiguration occurs.

A reconfiguration operation is triggered during the graph traversal when a function of the next segment is not included in the current configuration (line 5-7 in Algorithm 10). The current configuration is updated and the reconfiguration overhead O is accumulated. Similar to the reconfiguration overhead model at the circuit level, the reconfiguration overhead includes the time consumed for configuration file downloading and context switching. We estimate chip configuration time as the ratio between configuration file size and configuration interface throughput, and estimate context switching time as the ratio between transferred context data size $2 \cdot ds \cdot dm_{ds}$ and data transfer throughput ϕ. dm_{ds} is the domain-specific aspect. While the configuration file size at the circuit level is mainly determined by LUT usage, the function-level configuration file size depends on the resource usage of all on-chip resource types. The reconfiguration overhead O can thus be expressed as:

$$O_j = \frac{\gamma \cdot \max(\frac{P \cdot L_{\mathrm{s}+I_L}}{A_L}, \frac{P \cdot F_{\mathrm{s}+I_F}}{A_F}, \frac{P \cdot D_{\mathrm{s}}+L_D}{A_D}, \frac{M_{\mathrm{s}}+I_M}{A_M})}{\theta} + \frac{2 \cdot ds \cdot dm_{ds}}{\phi} \tag{4.17}$$

where O_j indicates the reconfiguration overhead when the current configuration is switched to configuration C_j, and γ is the configuration file size for 1% chip usage. We estimate the chip usage with the maximum resource usage in all resource types: $\max(\frac{P \cdot L_{\mathrm{s}+I_L}}{A_L}, \frac{P \cdot F_{\mathrm{s}+I_F}}{A_F}, \frac{P \cdot D_{\mathrm{s}}+L_D}{A_D}, \frac{M_{\mathrm{s}}+I_M}{A_M})$. Theoretically, accumulating the configuration file size for each resource type can provide better estimation. In practice, routing configuration data occupy a large portion of a configuration file file. As FPGA vendors do not provide routing infrastructure details, the routing configuration data size cannot be estimated. In our approach, we use the average chip usage coefficients γ and the maximum resource usage to estimate configuration file size. The additional data transfer time is given by $2 \cdot ds/\phi$.

The overall execution time of a partition P_k can be estimated by accumulating the execution time for each segment and the reconfiguration overhead for each reconfiguration operation. For an application with N uncompressed segments and a partition with R reconfiguration operations, the overall execution time can be expressed as:

$$T_k = \sum_{i=1}^{N} \frac{ds_i \cdot dm_{ds}}{P_i \cdot f_{dp}} + \sum_{j=1}^{R} O_j \tag{4.18}$$

where $\sum_{i=1}^{N} \frac{ds_i \cdot dm_{ds}}{P_i \cdot f_{dp}}$ is design execution time for N segments, and $\sum_{j=1}^{R} O_j$ indicates reconfiguration overhead. For an application, valid partitions include dynamic designs as well as static designs where all functions are grouped into a single configuration file. Therefore, T_k includes static design execution time T_{st} and dynamic design execution time T_{dy}. Given a data set, the overall execution time of all valid partitions are compared, and the partition with the minimum overall execution time T_k is executed.

4.7 Benchmark Applications

4.7.1 Barrier Option Pricing

A vanilla option is a financial instrument which provides the owner the right but not the obligation to buy or sell an asset at a fixed strike price K in the future. Similarly, a multi-variable option is an option with more than one underlying assets. The Multi-variable Barrier option is an exotic type of Multi-variable Option which changes its value if the underlyings reach the predetermined barrier. The rules for the change of value can be simple, for example, an up-and-out barrier option becomes worthless if the underlying asset price moves up across the barrier level. More complex rules can be applied for a Multi-variable Barrier option, Equation 4.19 shows the payoff function of a three-variable Barrier put option, where v_i is the payoff of the option at ith time step; v_i^{EU} is the price of a three-asset European option; lb_i and ub_i are the lower and upper barrier level at time step i; S_1, S_2 and S_3 are the underlying asset prices at time step i. In this case the payoff function contains mutually exclusive operations depending on the underlying asset price and the upper and lower barrier, it is therefore possible to apply our method to this problem.

$$v_i = \begin{cases} v_i^{EU}, & \text{if } lb_i < S_3 < ub_i \\ \max\left(0, K - \sqrt[3]{S_1 S_2 S_3}\right), & \text{if } lb_i \geq S_3 \\ \max\left(0, K - \frac{\sqrt{S_1 S_2} + S_3}{2}\right) & \text{if } ub_i \leq S_3 \end{cases} \tag{4.19}$$

The explicit finite difference (EFD) method is efficient to evaluate the payoff of financial derivatives with up to three underlyings, since it can be applied easily for various types of PDEs and the method is scalable for parallel execution. In this chapter we use EFD to solve the Black Scholes PDE [Hul05] with three underlyings and apply the payoff function as shown in Equation 4.19 to evaluate the payoff the the barrier option. As a result, a nineteen point convolution is shown in Equation 4.20, where j, k and l are indices for underlyings S_1, S_2 and S_3; α is a corresponding coefficient for a particular v.

$$v_{i,j,k,l}^{EU} = \alpha_1 v_{i+1,j,k,l} + \alpha_2 v_{i+1,j+1,k,l} + ... + \alpha_{19} v_{i+1,j,k-1,l-1} \tag{4.20}$$

For financial derivatives with more than three underlyings, the explicit finite difference method is usually considered to be both memory and computationally intensive; and Monte Carlo methods are more favourable. The application RDFG is presented in Figure 4.10(a), with function A and B indicating the payoff functions before and after reaching the barrier.

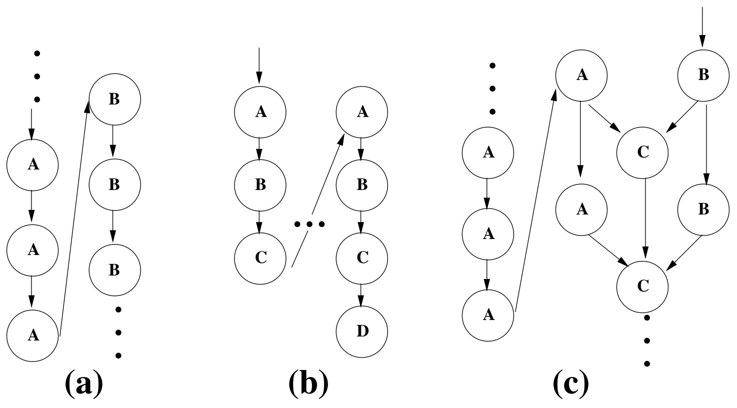


Figure 4.10: Function-level RDFG of (a) BOP, (b) PF and (c) RTM.

4.7.2 Particle Filtering

Particle filter (PF) is a methodology to deal with dynamic system having nonlinear and non-Gaussian properties. PF estimates the state of a system by a sampled-based approximation of the state probability density function. PF has been applied to real-time applications including object tracking [HLP11], robot localisation [M+02], speech recognition [VADG02]. Within the real-time constraint, the PF undergoes three key steps: particle generation, weight updating and resampling. The first two steps is data-independent and can be implemented concurrently by multiple processing elements in the FPGA. However, the resampling step involve communication of data among processing elements and it can only start after the first two steps finish. Therefore, the resampling step is stalled and kept idle. Indicated by the ATAP levels, the stalled functions can be grouped and optimised into different configurations to improve the system efficiency. The grouping strategies depends on the delay between ATAP levels, as well as function characteristics. As shown in Figure 4.10(b), particle generation, weight updating, re-sampling and grouping are represented as function node A, B, C and D, respectively.

4.7.3 Reverse Time Migration

Reverse Time Migration (RTM) is an advanced seismic imaging technique to detect terrain images of geological structures, based on the Earth's response to injected acoustic waves. The wave propagation within the tested media is simulated forward, and calculated backward, forming a closed loop to correct the velocity model, i.e. the terrain image. The propagation of injected waves is modelled with the isotropic acoustic wave equation [AP+11]:

$$\frac{d^2 p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \tag{4.21}$$

The propagation involves stencil computation, as the partial differential equation is approximated with the Taylor expansion. A fifth-order approximation is implemented in our experiment. As demonstrated in Figure 4.10(c), injected waves are first propagated from injected nodes into the detected terrain, labelled as function A. Once the propagation reaches the bottom, a reversed propagation and a backward propagation are instantiated simultaneously, represented as function nodes A and B. The propagated data are convolved in function C to generate the terrain image.

4.8 Results

Benchmark applications are developed with the proposed design flow. The hardware designs are produced by the Maxeler MaxCompiler version 2012.1, implemented on Xilinx Virtex-6 SX475T FPGAs, each hosted by one of the four MAX3424A systems in an MPC-C500 computing node from Maxeler Technologies. CPU designs are compiled with Intel Compiler (ICC) with -O3 flag opened, linked against OpenMP libraries, and executed on a Dell PowerEdge R610 machine, with 24 Intel(R) Xeon(R) X5660 cores running at 2.67GHz. An NVIDIA Tesla C2070 card with 448 CUDA cores is used for GPU designs. GPU implementations are optimised with relevant techniques such as access blocking and data coalescing [PF10a].

4.8.1 Design Flow Output

The RDFGs of benchmark applications are fed into the proposed design flow. Function nodes are assigned ALAP and ATAP levels. Nodes A, B and C for PF (Figure 4.10(b)) are combined into the same segment, as ATAP levels of B and C are 0. Similarly, function C of RTM (Figure 4.10(c)) is moved into the segment containing function nodes A and B. The number of generated segments are listed in Table 4.2, where G, S, C and P stand for the number of function nodes, segments, configurations and partitions generated in the proposed approach. After the ATAP assignment, the number of segments is reduced from 1501 to 501 for PF, and from 3000 to 2000 for RTM. Before generating configurations, the duplicated segments including same functions are eliminatd, leaving 2 segments for each application. Limited by *Rules 1 and 2*, three configurations are generated by Algorithm 7. For two segments, there will only be consecutive segments, i.e., so there will not be inefficient configurations. If the number of segments goes beyond two, for example four, instead of generating all 16 configurations, Algorithm 7 would only generate the 9 valid configurations.

The generated configurations are put into the configuration map shown in Figure 4.9(b). Following *Rules 3, 4 and 5*, the Ending-Segment Search Algorithm generates 2 valid partitions for each application. As listed in Table 4.2, one partition is the static design, where all functions

Table 4.2: Output results of proposed design flow.

application	G	S	C	P	static	dynamic0	dynamic1
BOP	2000	2000	3	2	AB	A	B
PF	1501	501	3	2	ABCD	ABC	D
RTM	4000	2000	3	2	ABC	A	ABC

are included in one configuration, labelled as static. The other partition refers to the design using runtime reconfiguration to eliminate idle functions, with the first and second configurations respectively labelled as `dynamic0` and `dynamic1`. With extracted function properties and reduced search space thanks to the design rules, valid and efficient reconfigurable designs are generated, from large-scale application graphs.

Measured and estimated resource usage are shown in Figure 4.11. We show resource usage of the static designs as a static design contains all application functions. As shown in Figure 4.11, the estimated resource consumption is within 90% of the measured value, which enables the `configuration organisation` step to properly duplicate the relevant functions. The differences between the measured and the estimated resource usage come from the neglected design parameters. One of the neglected design parameters is on-chip memory bandwidth. The current model estimates memory resource usage by accumulating memory bits consumed to store on-chip data. However, memory resource usage also depends on on-chip I/O operations. In Figure 4.4, as there are 8 data buffer elements that read from neighbouring elements and write to data-paths, 8 memory dual-port memory blocks are consumed. For large-scale applications, such as the three benchmark applications, millions of memory bits are used. The optimised designs are bounded by memory capacity instead of memory bandwidth. The model errors due to the neglected parameters are thus small.

4.8.2 Performance of Generated Partitions

The generated reconfigurable designs are evaluated in terms of execution time and resource utilisation ratio. The performance of the reconfigurable designs is measured for the MPC-C500 node. The resource utilisation ratio is calculated as the ratio between theoretical execution time and measured execution time. The theoretical execution time is calculated assuming

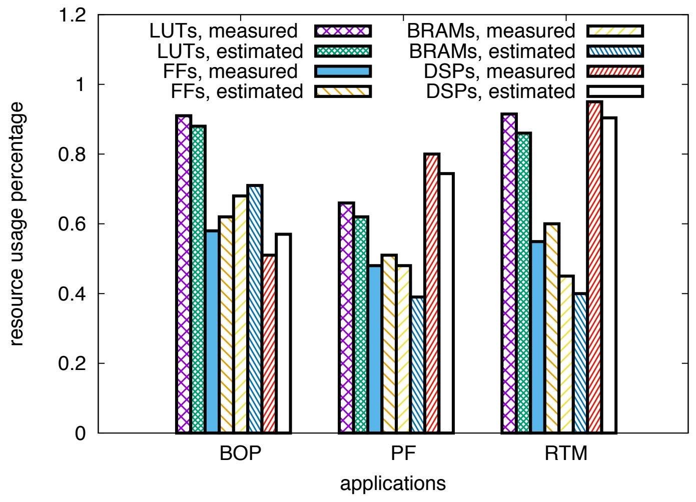


Figure 4.11: Measured and estimated resource usage of static designs for BOP, PF and RTM.

every implemented data-path generates one result per clock cycle. For dynamic designs, the communication between consecutive configurations is through memory transfers: output data of current configuration are transferred back into host memories before reconfiguring FPGAs, and back after the reconfiguration. The reconfiguration overhead O_r includes all configuration time and data transfer time.

Table 4.3: Performance of generated reconfigurable designs.

app	design	P	T(s)	O_r(s)	utilisation	speedup
BOP	static	24	111.84	0.79	0.496	1x
	dynamic0	48	27.94	1.53	**0.97**	**1.95x**
	dynamic1	48	28.2			
PF	static	4	20.9	1.1	0.346	1x
	dynamic0	10	7.41	2.2	**0.76**	**2.19x**
	dynamic1	5	0.39			
RTM	static	6	111.85	1.22	0.73	1x
	dynamic0	12	27.96	2.38	**0.962**	**1.31x**
	dynamic1	6	55.93			

For the static BOP, the mutually exclusive functions determine that only half of the resources can be used to generate useful results. The parallelism P is limited by available on-chip resources. As listed in Table 4.3, the idle functions in static BOP reduce its utilisation ratio to only 0.496. By distributing function A and B into two hardware configurations, P is doubled for both configurations, increasing the resource utilisation ratio to 0.97 and achieving 1.95 times speedup compared with the static design. The left 0.03 inefficiency is introduced by the reconfiguration overhead. For PF, the grouping function D is stalled while particles are updated by function A, B and C. During the grouping stage, function A, B and C are idle. Resources occupied by idle functions are reconfigured to support active functions. The optimised dynamic design for PF runs 2.19 times faster than its static counterpart. For RTM, the static design is bounded by available hardware resources and memory bandwidth. As shown in Figure 4.10, both function A and B require off-chip data. The memory channels connected to function B are idle when only function A is processing data. The generated dynamic design releases the idle resources and the idle memory channels, increasing the design parallelism of the first configuration to 12. The resource utilisation ratio reaches 0.96, and a 1.31 times speedup is achieved for the dynamic design.

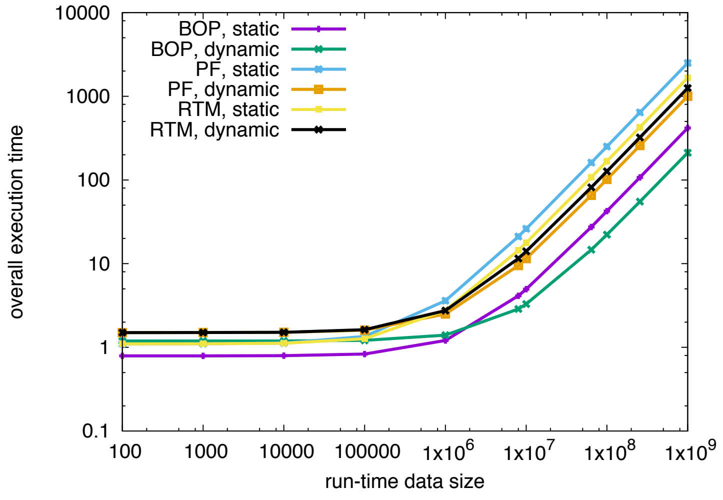


Figure 4.12: Evaluation results from the performance model, for the benchmark application Barrier Option Pricing (BOP), Particle Filter (PF) and Reverse Time Migration (RTM). For various data sizes, the overall execution time for the static and dynamic partitions are compared, and the partition with the minimum execution time is selected.

4.8.3 Runtime Evaluation

Results presented in Section 4.8.2 are for initial data sizes of the benchmark applications. The performance model provides runtime evaluation for the generated partitions, when data size varies. For the three benchmarks, two partitions are generated for each application. Constant coefficients are extracted from the partitions by traversing the application graphs. For static partitions with only one configuration, there is no reconfiguration overhead. For dynamic

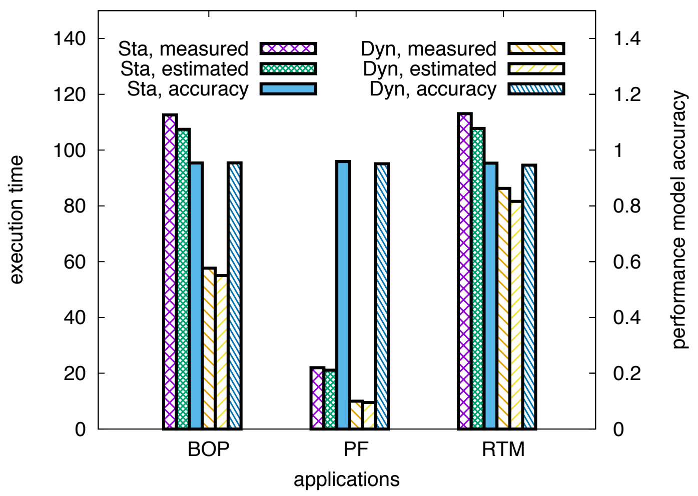


Figure 4.13: Measured and estimated execution time of BOP, PF and RTM. The model accuracy is higher than 95%. Sta indicates the static partitions, and Dyn indicates the dynamic partitions.

partitions, the parallelism in each configuration is increased, while reconfiguration overhead is introduced to eliminate the idle functions in each configuration. The parallelism for configuration in the static and dynamic partitions is presented in Table 4.3. All configurations operate at 100 MHz, and the throughput of PCI-e channels is 1 GB/s. Functions in the same benchmark application process the same data set. Evaluation results from the performance model are presented in Figure 4.12.

Evaluated data size varies from 100 to 10^9 data items for each application function. Reconfiguration overhead dominates the execution time when data size is small, while the impact of eliminating idle functions becomes obvious as data size increases. When there are more than 10^5 data items to process, the dynamic PF and RTM partitions outperform their static counterparts. The dynamic BOP partition runs faster than the static partition when the application data size is beyond $2 \cdot 10^6$. During run time, the performance model provides rapid estimation of execution time of various partitions, by updating the data size variable ds in Eq. 4.18. Figure 4.13 compares the measured execution time and the predicted execution time of the benchmark applications. The measured results align with the estimated values. The accuracy of the runtime performance model is more than 95%. Since the performance model estimates execution time as the ratio between data size and peak performance (with no coefficients tuned for the measured results), this indicates that the benchmark designs achieve 95%

of their theoretical peak performance.

4.8.4 Performance Comparison

The performance of the optimised partitions is compared with CPU and GPU implementations. This verifies whether the method can provide high performance of optimised hardware while achieving high resource utilisation, and evaluates the efficiency of the proposed method in a single-chip environment. To provide a fair comparison, the throughput and efficiency results include reconfiguration overhead O_r and static power consumption.

The performance of the benchmark applications on various platforms are shown in Table 4.4. CPU implementations are used as reference designs, generating 2.18 to 13.29 GFLOPS throughput. With high parallelism in processing units and local memory systems, GPU designs achieve 4 to 18 times speed-up. Based on results from NVIDIA Visual Profiler (NVPP), GPU performance is limited by memory operations to load data from global memory into local memory. The efficiency is limited between 29.5% to 34.3%, i.e., 3 to 4 loading operations are required to load one block of data into local memory. The inefficiency is introduced by the generality of the GPU architectures. With runtime reconfiguration introduced, available resources can be customised for each configuration, based on function properties extracted from the hierarchical graphs. The dynamic designs achieve up to 130.7 GFLOPS throughput, run up to 1.55 times faster, and are 2.9 to 4 times more power efficient than the optimised GPU designs. It is worth mentioning that the performance of static designs is lower than or at the same level as the GPU performance. Although the general architecture of CPUs and GPUs introduces inefficiency for operations such as data access, the generality of such architectures enables the same computing units utilised by various application functions, which compensates the comparatively low performance for each function. The proposed approach enables resource sharing in the time dimension, with high performance for each application function.

Table 4.4: Comparison of the design performance of benchmark applications. Operating frequency is expressed with GHz, design throughput is expressed with GFLOPS, and power efficiency is expressed with MFLOPS/W. T and O respectively indicate execution time and reconfiguration time.

	Barrier Option Pricing				Particle Filter				Reverse Time Migration			
	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn
frequency (GHz)	2.67	1.15	0.1	0.1	2.7	1.15	0.1	0.1	2.67	1.15	0.1	0.1
T (s)	631	33.9	55.9	28.0	10	8.50	8.90	7.80	661	104	99.2	66.1
O (s)	0	0.43	0.80	1.53	0	1.50	1.10	2.20	0	0.59	1.22	2.38
throughput[1]	12.3	102	61.2	119	2.2	39.3	26.5	58.2	13.3	84.3	99.9	130.7
throughput	1x	8.3x	5.0x	**9.6x**	1x	18.0x	12.2x	**26.7x**	1x	6.34x	7.5x	**9.8x**
power (W) [2]	280	365	145	145	253	291	130	130	245	369	141	142
efficiency	44.0	280	422	819	8.6	135	204	448.0	54.2	228	709	920
efficiency	1x	6.4x	9.6x	**18.6x**	1x	15.7x	23.7x	**52.0x**	1x	4.22x	13.1x	**17.0x**

[1] Throughput is calculated with all data transfer time and device configuration time included.
[2] Power consumption includes both static power and dynamic power.

4.9 Related Work

Previously, previous work for function-level runtime reconfiguration includes design approaches to partition applications and to schedule runtime configurations. (1) When available resources cannot accommodate all functions at the same time, application functions are partitioned into multiple subprograms. These subprograms are sequentially reconfigured into an FPGA to ensure subprogram can fit into the FPGA. In these partitioning approaches, DFGs are used to represent function nodes [PB99], and an Integer Non-linear Programming (INLP) model is introduced to formulate the partitioning problem [KV98]. (2) Runtime reconfiguration scheduling approaches select the time points to download configurations into reconfigurable devices, to reduce reconfiguration overhead and to increase execution time. In [FC05], a knapsack-based scheduler is proposed to select the configurations (design kernels) with maximum speedup, under fixed reconfiguration intervals. This scheduling approach is further improved by adaptively adjusting the reconfiguration intervals [FC08], which reduces the overall scheduling overhead by 85%. In the SCORE project [CDW01], a page scheduler is developed to decide which compute page (runtime reconfiguration) is executed, to minimise overall execution time.

In this chapter, we propose a design approach to handle idle resource units at the function level. This approach automatically detects idle functions with high-level analysis, and eliminates these idle functions with runtime reconfiguration. Instead of partitioning application functions to fit partitioned functions into available resources, we group functions active at the same time to gain more resources to replicate active functions. The proposed approach can benefit applications with idle functions, as long as such applications can be accelerated by parallelising the execution of the application tasks. The design objective is to achieve the maximum application performance on the target reconfigurable platform, bounded by available resources. Compared with previous scheduling algorithms, the runtime evaluator in this work use a performance model to be aware of improved performance of the optimised reconfiguration designs, as well as the introduced reconfiguration overhead. Experiments show that a large speedup can be achieved on existing reconfigurable platforms, for real-life applications.

4.10 Limitations and Future Work

The efficiency of the proposed approach is limited by reconfiguration overhead. Rather than aiming for the optimal case that any idle functions are eliminated by dynamically configuring the applications functions, the current approach uses a evaluator to estimate the benefits and overhead gained from applying runtime reconfiguration, and only applies runtime reconfiguration when the overall execution time can be reduced. In other words, for scenarios that function units become idle at high frequency, for example, application functions stay idle every two out of ten clock cycles, the idle function modules will not be reconfigured. For large-scale applications, the reconfiguration overhead is negligible and high design efficiency can be achieved. However, for small-scale applications that need frequent reconfiguration, the overall execution time of a runtime reconfigurable design is dominated by runtime reconfiguration time, which leads to low design efficiency.

In the future, reconfiguration techniques with reduced reconfiguration overhead will be explored, to generate reconfigurable designs (partitions) with improved reconfiguration granularity. Reconfiguration overhead is a common issue for the proposed design approaches at all the three design levels. We discuss the impacts of the reconfiguration overhead and possible future improvements in more detail in Chapter 6.

4.11 Summary

A function-level design approach is proposed in this chapter. By introducing reconfiguration into the design method, computational resources not contributing to outputs all the time are automatically identified and utilised to further improve system performance. Runtime reconfiguration enables effective exploitation of computational resources which would otherwise stay idle, and we show that opportunities for such exploitation can be automatically identified and optimised. Three applications — barrier option pricing, particle filter, and reverse time migration — are used in evaluating the proposed approach. The runtime solutions approximate their theoretical peak performance by eliminating idle functions, and are 1.31 to 2.19 times faster than optimised static designs. FPGA designs developed with the proposed approach are up to 43.8 times faster than optimised CPU reference designs, and up to 1.55 times faster than optimised GPU designs.

Chapter 5

System-Level Optimisation for Runtime Reconfigurable Designs

5.1 Introduction

A reconfigurable system consists of various FPGAs. In this work, we assume each FPGA can only be used by one application at the same time[1]. Therefore once used by one application, an FPGA remains unavailable until the application finishes. As applications are launched by system users from time to time, FPGA availability remains indeterministic during runtime. Lack of sufficient runtime information during design time, hardware designs need to make assumptions about runtime FPGA availability. When the assumed runtime scenarios do not match the actual resource status, idle FPGAs are introduced. At the system level, an idle resource unit refers to an FPGA that is available during the execution of a reconfigurable design, while not being used by the application.

In order to match a reconfigurable design to the indeterministic resource availability, we introduce runtime reconfiguration at the system level to dynamically coordinate FPGAs that become

[1]While there have been work that tried to map multiple applications into one FPGA at the same time [CSZ+14], this technology cannot be directly applicable to existing reconfigurable designs, with one of the major limitations being the necessity to do floorplanning manually. In the future when the device-sharing technique is mature enough, the system-level approach can still benefit reconfigurable designs, with improved reconfiguration granularity.

available during runtime. Theoretically, the optimal system efficiency can be achieved as there will be no idle FPGAs in a reconfigurable system: whenever an FPGA is freed by an application, one another application running in the system will use this FPGA for its computational tasks.

Outline. In this chapter, we propose a design approach that enables the users of reconfigurable systems to develop dynamic designs at a high-level, with design tools and models handling the low-level details automatically. Section 5.2 provides an overview of the approach. Section 5.3 and 5.4 respectively present the compile-time and runtime techniques applied to support dynamic designs at the system level. The design objective is to fully exploit all available FPGAs during runtime, with minimised reconfiguration overhead. Section 5.5 presents the two benchmark applications used in this chapter. To evaluate the generality of the proposed approach, we choose two applications with diverse communication patters. Finally, Section 5.6 presents the experiment results, Section 5.7 compares the related work, Section 5.8 discusses the limitations of the approach, and Section 5.9 summarises this chapter.

5.2 Approach Overview

5.2.1 Reconfigurable System

The last few years have given rise to large computer infrastructures, such as systems and datacentres, which provide ample compute resources. In contrast to the increase in the requirements for high-performance datacentre services, the rate of performance increase in datacentres has slowed down significantly, mainly due to power limitation [Sut05]. Fabricating Application-Specific Integrated Circuit (ASICs) for datacentre services is overkill for this problem, due to the rapid evolution of datacentre services and the large cost for fabricating an ASIC. FPGAs provide a platform to develop reconfigurable designs in hardware. A reconfigurable design is captured in hardware languages, synthesised by vendor tool chains, and downloaded into FPGAs to execute. The reconfigurablility of FPGAs balances the requirements for high-performance customised designs and the necessity to adapt to data service evolvement. In a reconfigurable system proposed by Microsoft [P+14], the Bing web search engine is improved by 95% in throughput and reduced by 29% in latency.

Applications are launched into a system from time to time, and sharing resources in such environment adds complexity in the development process: applications must not only efficiently exploit a given set of compute resources, but also adapt dynamically to available resources at run time. When we assume a reconfigurable system with FPGA nodes in different generations occupied and released by various computational tasks, for a given design, throughput can be potentially increased if more FPGAs are available to perform the design computation. We illustrate the basic idea of this chapter with a motivating example.

5.2.2 Motivating Example

The effectiveness of current static design methods is limited by unpredictable runtime conditions. Due to non-deterministic starting time of applications, node availability and the amount of computational resources in available nodes are unknown during compile time. In this example, FPGA nodes A, B, C and D are released by other applications at time 0, 2, 3 and 4, respectively; node A, B and D possess 1 resource unit and process 1 data unit per second, while node C can process two data per second; application with 8 data to process is launched into the system. Linear scalability is assumed for executed tasks, i.e., execution time is halved if the number of utilised resource units increases from 1 to 2.

Two static designs are illustrated in Figure 5.1. The OneNode Design will make use of only one node, so would take 8 seconds to complete. The FourNode Design will take all 4 nodes when all of them become available at time 4, and would take 2 seconds to complete. Only half of the computational capacity in node C is utilised, as the FourNode Design pre-defines that one resource unit is used in each runtime node. The Dynamic Design, in contrast, can start at time 0 when node A becomes available; then at time 2, after node A processes two data, node B becomes available too, so both nodes process another two data in the next second. At time 3 node A, B and C are available, completing the processing of the 4 remaining data.

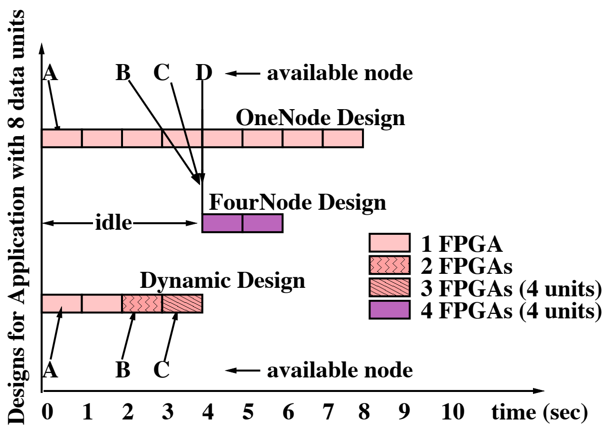


Figure 5.1: Execution of various designs for a computational task with 8 data items to process, when node A, B, C, and D are released. Performance of three designs for the same applications is presented.

A dynamic design, as discussed in the motivating example, can fully exploit available resources in a reconfigurable system. Developing such a dynamic design requires designers (1) to customise the design for various FPGA devices in the system (in this example, the same FPGAs are used in node A, B and D, while node C contains a different FPGA), (2) to manage the customised designs to adapt to runtime resource availability variations, and (3) to schedule the computation and communication operations of the adapted designs to ensure linear performance scalability, when more FPGA nodes are involved. In correspondence to the three challenges, the proposed approach includes three design steps: compile-time optimisation and runtime scaling. The compile-time optimisation handles the first challenge, and the runtime scaling handles the latter two challenges.

5.2.3 Design Flow

The proposed approach starts from a C program developed by users, then compiles the computation kernels into optimised hardware descriptions, and generates a reconfigurable design that adapts to available resources at runtime. As shown in Figure 5.2, this approach includes two steps: compile-time optimisation and runtime scaling. Table 5.1 summarises the design parameters used at the system level.

The compile-time optimisation is automated with a compiler built with the ROSE infrastruc-

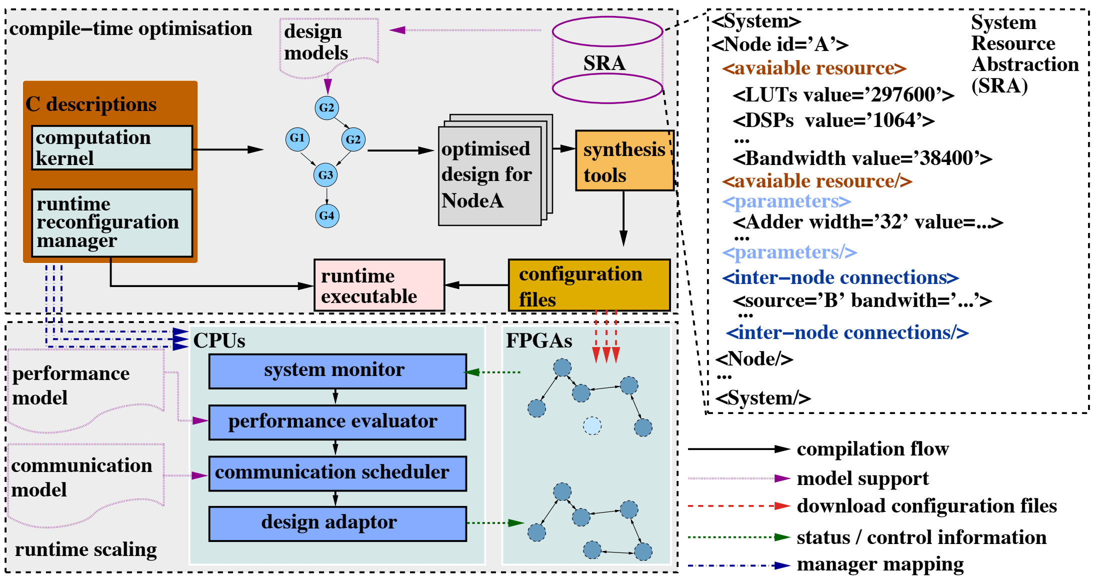


Figure 5.2: An overview of the System-level approach.

ture [Qui00]. The compiler front-end translates the computational kernels in a C program into hardware descriptions, with Data-Flow Graphs (DFGs) extracted as intermediate representations. In a DFG, the graph nodes represent arithmetic operations, and the graph edges represent data access operations. The compiler back-end optimises the translated hardware descriptions to fully exploit available resources in FPGA nodes. The back-end is underpinned by a design model and a System Resource Abstraction (SRA). The design model estimates the resource usage of the generated hardware design, by analysing the extracted DFGs. The SRA stores the system node properties, which include available resources, design parameters, and inter-node connections. When a reconfigurable system contains FPGA nodes with different properties, the compiler back-end generates multiple hardware designs with different optimisation parameters. We use vendor tools to synthesise the optimised designs into configuration files. Besides the computational kernels, the C program also contains a runtime reconfiguration manager, which is compiled and linked to the configuration files.

The runtime scaling refers to the reconfiguration of a hardware design, when new system nodes become available during the execution time of this hardware design. The runtime reconfiguration manager, as shown in Figure 5.2, consists of a system monitor, a performance evaluator, a communication scheduler and a design adaptor. The system monitor reports the currently available FPGA nodes in the system. Once new nodes are detected, the performance evaluator

Table 5.1: Variables and parameters in the system-level approach.

variables		**parameters**	
optimisation model			
P	design parallelism	A	available resources
I	infrastructure resources	BW	available bandwidth
communication model			
D_{ev}	number of FPGAs	T_{arr}	arrival time of dependent data in node D
$T_{del,cmp}$	scheduled computation delay time	$T_{del,dat}$	scheduled data access delay time
C	network bandwidth margin		
domain-specific aspects			
stencil computation			
$_{fro}$	front halo data	$_{end}$	end halo data
n_D	dimension D size	w_D	stencil size in dimension D
performance model			
RT_{bne}	runtime benefits	D_{ev}	number of FPGAs
O_{rf}	reconfiguration overhead	P	design parallelism
θ	data transfer throughput	ϕ	(re)configuration throughput
γ	configuration file size per resource unit	R	design resource usage

estimates whether scaling the current design can improve design performance. The communication scheduler supports asynchronous communication operations to minimise communication time. If the performance evaluator determines to scale the design over the new nodes, the communication operations are rescheduled to ensure correct computation results. Finally, the reconfiguration adaptor redistributes workload, downloads corresponding configuration files into the new nodes, updates the rescheduled communication parameters, and resumes the computation.

5.3 Compile-Time Optimisation

The compile-time optimisation, like the optimisation model used at the circuit and the function level, aims to exploit all available resources in an FPGA computing node to achieve maximum throughput. The optimised configurations generated at the function level integrates customised operators and eliminates idle functions. Therefore, the generated configurations are capable of fully exploiting the available resources in an FPGA node. One additional issue at the system level is the heterogeneous FPGA nodes in a reconfigurable system: a reconfigurable system often contains FPGAs from different generations and different vendors, and the available resources in these FPGAs vary from one to another.

At the system level, we handle this issue with a System Resource Abstraction (SRA) file. An

SRA file stores the information for each FPGA under separated tags, as shown in Figure 5.2. For the optimisation model parameters in Table 5.1, the available resources A and BW, the infrastructure resource usage I, and the design parameters $R_{s,o}$ and $B_{s,o,bit,fix}$ are updated based on FPGA node specifications ($R_{s,o}$ and $B_{s,o,bit,fix}$ are discussed in Section 3.5). During compile time, the optimisation model goes through all FPGA nodes in a system, updates the model parameters based on the stored information in the SRA file, and generates the optimised configurations. The optimisation model of an FPGA node i therefore can be expressed as follows.

FPGA node i

$$\textbf{minimise: } \frac{ds \cdot dm_{ds}}{P_i \cdot f_{dp}} \tag{5.1}$$

subject to:

$$L_{LUT/FF/DSP,i} \cdot par \cdot dm_{LUT/FF/DSP} + I_{LUT/FF/DSP,i} \leq A_{LUT/FF/DSP,i} \tag{5.2}$$

$$M_{s,i} \cdot dm_{Ms} + I_{Ms} \leq A_{Ms,i} \tag{5.3}$$

$$M_{bw} \cdot dm_{BW} \leq BW_i \tag{5.4}$$

The current system-level approach covers two application domains: Monte-Carlo simulations and stencil computations. The optimisation techniques have been discussed at the function level, and the same domain-specific parameters are integrated into the optimisation model. After generating the optimised configurations for system FPGAs, the computational capacity $P_i \cdot f_{dp}$ of each FPGA is collected for runtime scaling.

5.4 Runtime Scaling

The runtime scaling process aims at utilising FPGAs that turn to be available during the execution time of a reconfigurable design. Algorithm 11 shows the runtime scaling processing of a dynamic design, Figure 5.2 presents the main components used in runtime scaling. The scaling process includes three steps. First, a system monitor reports the variations in resource availability (line 2), and a performance evaluator estimates performance improvements when new FPGA nodes become available (line 3∼4). If the reduction in overall execution time outweighs the scaling overhead, the scaling algorithm stalls computation, and triggers the following steps (line 5). Second, during the scaling process, we use an asynchronous communication model to minimise communication overhead, and reschedule the communication parameters in the model to ensure correct functionality (line 6). Finally, a design adaptor reconfigures the new node, redistribute application workload, resumes computation, and goes back to the monitoring step (line 7). The algorithm is executed iteratively to adapt to the dynamic design to runtime resource variations.

Algorithm 11 Runtime scaling algorithm of the system-level approach.

```
1: while design execution not finished do
2:   if system monitor: detects an available node $n$ then
3:     performance evaluator: calculate data distribution based on Eq.5.5.
4:     performance evaluator: calculate runtime benefits $RT_{bne}$ based on Eq.5.6
5:     if $RT_{ben} > 0$ then
6:       communication scheduler: schedule asynchronous communication operations based
         on Algorithm 13.
7:       design adaptor: reconfigure the node $n$ into current dynamic design.
8:     end if
9:   end if
10: end while
```

5.4.1 Performance Evaluation

Workload Distribution

Workload distribution balances the execution time of involved FPGAs, so that all FPGAs in a reconfigurable design finish their task at the same time. At the system level, various design configurations are used in a single reconfigurable design to coordinate the heterogeneous FPGAs. The processing capacity of an FPGA depends on the customised parallelism of implemented configurations, which is determined by the compile-time optimisation process. Given a reconfigurable design using D_{ev} configurations (i.e. D_{ev} FPGAs) at the same time, the workload for configuration (device) i can be expressed as:

$$wl_i = \frac{ds \cdot dm_{ds}}{\sum_{j=1}^{D_{ev}} P_j} \cdot P_i \tag{5.5}$$

where for FPGA i, wl_i indicates the assigned workload, P_i indicates its computational capacity, and $ds \cdot dm_{ds}$ accounts for the overall workload.

Performance Model

While capturing more computational resources to process the workload in parallel, scaling a dynamic design also involves context switching and device reconfiguration, which introduce reconfiguration overhead. We develop a performance model to determine whether a dynamic design should scale onto new FPGA nodes that become available during runtime. The performance improvements for exploiting available FPGAs depend on the increment in computational capacity and the amount of remaining workload. As an example, if an FPGA becomes available right after a dynamic design is launched, a large speedup can be achieved. On the other hand, if an FPGA is available when a dynamic design is about to finish, the performance improvement introduced by the new FPGA is negligible.

Runtime benefits refer to the reduction in execution time for the remaining application tasks, when a dynamic design expands over more FPGAs. The application tasks often involve processing workload data iteratively, and therefore is expressed as $wl \cdot it$, where it is the remaining iterations. After scaling a dynamic design over new FPGAs, the number of iterations to process it stays the same, while the distributed workload wl for each FPGA reduces, based on Eq.5.5. Runtime benefits RT_{bne} can be expressed as:

$$RT_{bne} = \frac{it \cdot (wl_{[i,cur]} - wl_{[i,nex]})}{P_i \cdot f_{dp}} - O_{rf} \tag{5.6}$$

where $wl_{[i,cur]}$ and $wl_{[i,nex]}$ indicate the distributed workload before and after runtime scaling. The difference between $wl_{[i,cur]}$ and $wl_{[i,nex]}$ represents reduced workload in FPGA i due to increased computational capacity. Since the workload is distributed based on processing capacity $P_i \cdot f_{dp}$, the reduction in execution is the same for all involved FPGAs.

The reconfiguration overhead refers to time consumed to reconfigure FPGAs and to switch context. The context switching refers to redistributing intermediate results in current dynamic design into FPGAs in the scaled dynamic design. The intermediate results of current FPGAs are loaded from off-chip memories back to host memories; corresponding configuration files are configured into new FPGAs; and the intermediate results are redistributed into FPGAs in the expanded dynamic design, to ensure application contexts are preserved in involved FPGAs. The reconfiguration overhead can be expressed as:

$$O_{rf} = \max(\frac{R \cdot \gamma}{\theta}, \frac{wl_{[i,cur]}}{\phi}) + \frac{wl_{[i,nex]}}{\phi} \tag{5.7}$$

where $\frac{wl_{[i,cur]}}{\phi}$ and $\frac{wl_{[i,nex]}}{\phi}$ respectively indicate the time to load and redistribute memory data, through PCI-e channels with bandwidth ϕ. The reconfiguration time can be estimated with configuration file size and throughput of reconfiguration interface θ. The configuration file size is calculated with resource usage R and configuration file size per resource unit γ. Since memory controllers and streaming architectures are configured into the same FPGA in current designs, context data can only be written into new FPGAs nodes when runtime reconfiguration is finished. The loading of context data, on the other hand, is executed in parallel with reconfiguration operations. Therefore, the O_{rf} only includes the upper bound of reconfiguration time and data loading time.

5.4.2 Asynchronous Communication Scheduling

We divide the data dependencies in a design into two categories: intra-iteration dependencies and inter-iteration dependencies. As shown in Figure 5.3(a), intra-iteration dependencies indicate the computation in an iteration depends on input data in the same iteration, while inter-iteration dependencies indicate the computation in an iteration depends on the results from previous iterations. The communication model in this work covers both intra-iteration and inter-iteration data dependencies.

Algorithm 12 An example algorithm.

```
1: for $it = 0 \leftarrow$ nt-1 do
2:     for $i = 0 \leftarrow 199$ do
3:         c(i) = (a[i-1] + a[i] + a[i+1]) * b[i];
4:     end for
5:     c = a;
6: end for
```

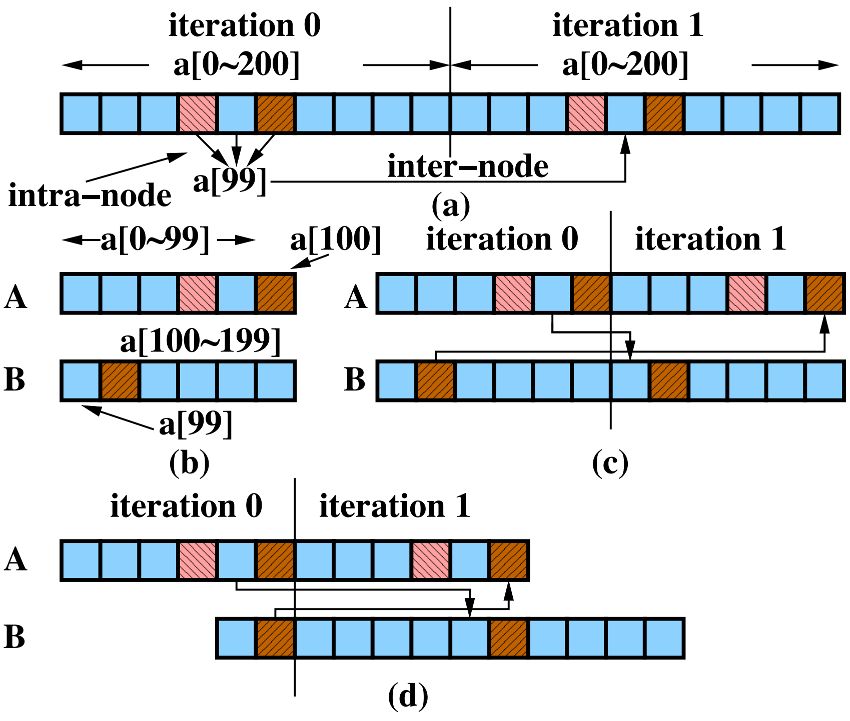


Figure 5.3: (a) Intra-iteration and intra-iteration data dependencies, for the example in Algorithm 12. The same example is used in Figure 3.8, Chapter 3. (b) We use workload distribution to resolve intra-iteration dependencies. For the (c) inter-iteration data dependencies after workload distribution, (d) we schedule communication operations to resolve this issue.

Intra-Iteration Dependency

The intra-iteration dependencies need to be protected when design workload is distributed across multiple FPGA nodes. As an example, in Algorithm 12 (see Figure 5.3(a)), the computation of $c[i]$ depends on $a[i-1]$, $a[i]$ and $a[i+1]$. Based on the design model at the circuit and the function level, $mem_{a,max} = 1$, and $mem_{a,min} = -1$. The communication model distributes workload based on the computational capacity of each involved FPGA node (P), to ensure the computation of all involved nodes finishes at the same time. In addition, for applications with intra-iteration dependencies, $mem_{a,max} - mem_{a,min}$ additional data are assigned, when one more FPGA node is involved in a dynamic design. As an example, for a dynamic design with FPGA nodes A and B where $P_A = P_B$, the 200 elements in array a are evenly split between A ($a[0] \sim a[99]$) and B ($a[100] \sim a[199]$), as shown in Figure 5.3(b). Since the computation of $c[99]$ requires $a[100]$, $a[100]$ ($99{+}mem_{max}$) is assigned to node A. Similarly, $a[99]$ ($100{+}mem_{min}$) is assigned to node B. By keeping all the dependent data local, the involved nodes in a reconfigurable design can run in parallel in a single iteration.

Inter-Iteration Dependency

While distributing additional dependent data to local nodes solves the intra-iteration dependencies, the dependent data ($a[100]$ in node A and $a[99]$ in node B) need to be updated before they are used in the next iteration. However, these data cannot be updated locally. Our communication model protects the inter-iteration dependencies by transferring dependent data during runtime. As shown in Figure 5.3(c), the dependent data are updated in a remote node, and transferred into the local node that uses the dependent data. For the same dependent data, we name the data copy in the remote node as `remote data`, and name the local data copy as `local data`. As an example, for $a[99]$, the `local data` refers to the $a[99]$ in node B, and the `remote data` refers to the $a[99]$ in node A.

To minimise the communication time, the communication model supports asynchronous communication operations so that the communication operations of an application overlap with its computation operations. Dependent data are updated in current iteration, and used in the next iteration. We use timing constraints to ensure the `remote data` can arrive in time: the `remote data` must arrive after the usage of the `local data` in the current iteration, which otherwise will overwrite the data to be used; in addition, the `remote data` must arrive before the usage of the `local data` in the next iteration. As an example, as shown in Figure 5.3(c), the `remote data` in node B are updated at the beginning of iteration 0, and used in the end of iteration 1. If transferred into node A once updated, the `remote data` in node B overwrite the `local data` in node A, before the `local data` are used in iteration 0. This violates the earliest timing constraint. We express the timing constraint for an FPGA node m as follows.

$$\frac{d_{loc,m}}{P_m} + T_{del,cmp,m} < T_{arr,m} < \frac{d_{loc,m} + ds_m}{P_m} + T_{del,cmp,m} \tag{5.8}$$

where $T_{arr,m}$ is the arrival time of `local data` into node m, $d_{loc,m}$ indicates the position of the `local data` ($d_{loc} = 0$ in node B), ds_m is the distributed workload size, and $T_{del,m}$ is the scheduled delay in node m. $\frac{d_{loc,m}}{P_m}$ indicates the data usage time in the current iteration (the earliest arrival time), and $\frac{d_{loc,m}+ds_m}{P_m}$ indicates the data usage time in the next iteration (the latest arrival time).

In correspondence to the timing constraints, the arrival time of dependent data from node n to node m can be expressed as:

$$T_{arr,m} = \frac{d_{rem}}{P_n} + \frac{ds_{dep} \cdot M}{bw_{n,m}} + T_{del,cmp,n} + T_{del,dat,m} \tag{5.9}$$

where d_{rem} indicates the position of the `remote data`, ds_{dep} indicates the dependent data size, $bw_{n,m}$ indicates the communication bandwidth between node n and m, and M is a margin factor for the communication operations.

In order to meet the timing constraints, we develop a communication scheduler to tune the scheduled communication delay ($T_{del,cmp}$) and data update delay ($T_{del,dat}$). In an FPGA node, the communication delay $T_{del,cmp}$ refers to an initial delay before the first iteration starts, and the data update delay $T_{del,dat}$ refers to the delay in the update of `local data`, when `remote data` are ready. $T_{del,cmp}$ and $T_{del,dat}$ have three impacts on communication operations. (1) Inserting $T_{del,cmp}$ delays the computation operations in local nodes, and thus adds an extension in local timing constraints (see Eq.5.8). (2) Inserting $T_{del,dat}$ delays the update time (i.e. arrival time) of `remote data` (see Eq.5.9). (3) When the local data in node m comes from node n, inserting $T_{del,cmp}$ in remote node n delays the arrival time of `remote data` in node m (see Eq.5.9).

The communication scheduler updates data access delay $T_{del,dat}$ and computation delay $T_{del,com}$ to satisfy the timing constraints. There are two cases for timing constraint violations. (1) If dependent data arrive too early, a data access delay $T_{del,dat}$ is inserted to postpone the update time of the dependent data in local memory. (2) If dependent data arrive too late, the dependent data cannot be scheduled to arrive earlier. Instead, starting time of the communication operations is delayed to postpone the latest timing constraints.

Our communication scheduler, as shown in Algorithm 13, goes through workload distribution and timing constraint check to ensure correct functionality when a dynamic design scales over new FPGA nodes. Since the computation delay in one node impacts the arrival times in its neighbouring nodes, we check the latest timing constraints first to update $T_{del,com}$. After updating the computation delay, we check the earliest timing constraints to schedule the data access delay $T_{del,dat}$. The scheduling algorithm traverses all FPGA nodes until there is no timing constraint violation. For the unscheduled communication operations in Figure 5.3(c), the `remote data` in node B are transmitted once updated, breaking the earliest timing constraint in node A. After communication scheduling, an initial delay is added to ensure the `remote data` in node B arrives after the usage of the `local data` in node A in iteration 0. Typically, high performance applications have thousands to millions of iterations, the inserted initial delay has negligible effect on design performance.

Algorithm 13 Communication scheduling algorithm for a runtime reconfigurable design at the system level.

input: the number of FPGAs in detected FPGA path: D_{ev}
output: scheduled computation delay $T_{del,com}$ and data access delay $T_{del,dat}$ for each FPGA.

```
 1: for $i \in 0 \to D_{ev}$ do {workload distribution}
 2:     $wl_{[i]}$ = workload_distribution()
 3:     $T_{arr,i}$ = unscheduled_arrive()
 4: end for
 5: for $i \in 0 \to D_{ev}$ do {schedule for the latest timing constratins}
 6:     if latest_arrive() $< T_{arr,i}$ then
 7:         $T_{del,com,i} = \max(T_{arr,i} -$ latest_arrive()
 8:         $T_{arr,i+1} += T_{del,com,i}$
 9:     end if
10: end for
11: for $i \in 0 \to D_{ev}$ do {schedule for the earliest timing constratins}
12:     if earliest_arrive() $> T_{arr,i}$ then
13:         $T_{del,dat,i} = \max($earliest_arrive() - $T_{arr,i})$
14:     end if
15: end for
```

5.4.3 Domain-Specific Aspects

Monte-Carlo Simulation

In a Monte-Carlo simulation path, the computation operations depend on the input data generated by a RNG. Inside the DFGs for Monte-Calro applications, the design model considers

RNGs as internal graph nodes, and the output edges from RNGs are implemented as on-chip wire connections. There are neither intra-iteration nor inter-iteration dependencies for Monte-Carlo simulations. Therefore, an FPGA can be directly used without concerning the communication constraints. No overlapped data distribution or asynchronous communication operation scheduling is required, since all dependent data are generated locally. In addition, the overall data size for a Monte-Carlo application remains the same during runtime, regardless of the number of involved FPGAs and the applied optimisation techniques. To dynamically reconfigurable a Monte-Carlo design at the system level, the runtime scaling process adapts workload distribution as more and more FPGAs are involved, following Eq.5.5.

Stencil Computation

Known to be communication intensive, stencil computation has complex intra-iteration and inter-iteration data dependencies. The data exchange between two neighbouring FPGAs is illustrated in Figure 5.4(a). The domain-specific aspects include workload distribution, timing constraints, and data arrival time.

Similar to the general communication model, the intra-iteration dependencies can be protected with workload distribution. Within a stencil iteration, the computation of a datum requires its neighbouring data, with the number of required neighbouring data determined by stencil size w_i. In order to protect the data dependencies, the halo data between two neighbouring FPGA nodes are distributed to both the FPGAs, as shown in Figure 5.4(b). In terms of the overall data size, besides spatial blocking and temporal blocking, two additional layers of halo data are introduced once one more FPGA is involved. Given a stencil problem with stencil size w_i, spatial blocking ratio sk, temporal blocking ratio tk, and D_{ev} FPGA involved, the distributed data for FPGA i can be expressed as:

$$wl_i = \frac{ds \cdot dm_{ds}}{\sum_{j=1}^{D_{ev}} P_j} \cdot P_i + 2 \cdot w_N \cdot s_{li} \tag{5.10}$$

$$s_{li} = \prod_{D=1}^{N_D-1} (\frac{n_i}{sk_i} + 2 \cdot w_i \cdot tk) \tag{5.11}$$

where w_N indicates the stencil size at the slowest dimension (e.g. the z dimension in Figure 5.4(a)), s_{li} indicates the size of one data slice at the slowest dimension after spatial and temporal blocking, and n_i is the unblocked dimension i size.

After workload distribution, the inter-iteration dependencies are protected by exchanging dependent between neighbouring FPGAs. As shown in Figure 5.4(b), each data block wl_i contains two dependent data regions, with the local copies named as the `front local data` and the `end local data`. Similarly, the data copies in remote FPGAs are named as the `front remote data` and the `end remote data`. In the unscheduled design in Figure 5.4(b), the `end remote data` arrive too early and overwrite the local data before they are used, rendering all following computation incorrect. Based on the general timing constraints presented in Eq.5.8, the timing constraints for these two dependent data regions in FPGA node m can be expressed as:

$$\begin{cases} 0 + T_{del,cmp,m} & < T_{arr,m,fro} & < \frac{wl_i}{P_m} + T_{del,cmp,m} \\ \frac{wl_i - s_{li} \cdot w_N}{P_m} + T_{del,cmp,m} & < T_{arr,m,end} & < \frac{2 \cdot wl_i - s_{li} \cdot w_N}{P_m} + T_{del,cmp,m} \end{cases} \tag{5.12}$$

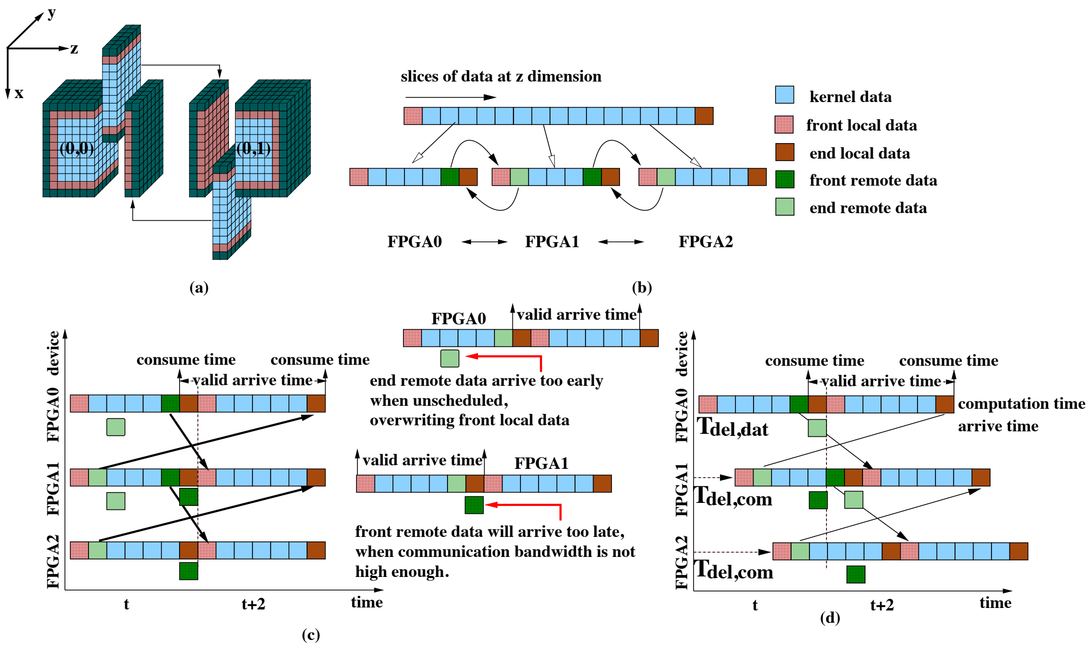
where $\frac{wl_i}{P_m}$ indicate the execution time of an iteration in node m, 0 and $wl_i - s_{li} \cdot w_N$ respectively indicate the `front local data` position and the `end local data` position ($d_{loc,m}$ in Eq.5.8), $T_{del,cmp,m}$ is the scheduled computation delay in node m, and the front and end data are labelled with $_{fro}$ and $_{end}$.

Given the separated timing constraints, the arrival times of dependent data are estimated based on the computational capacity P of the FPGA nodes, the dependent data position, and the scheduled computation delay in the remote node.

$$\begin{cases} T_{arr,m,fro} = \frac{wl_n - w_N \cdot s_{li}}{P_n} + \frac{w_N \cdot s_{li} \cdot M}{bw_{n,m}} + T_{del,cmp,n} + T_{del,dat,m} \\ T_{arr,m,end} = \frac{2 \cdot w_N \cdot s_{li}}{P_n} + \frac{w_N \cdot s_{li} \cdot M}{bw_{n,m}} + T_{del,cmp,n} + T_{del,dat,m} \end{cases} \tag{5.13}$$

where the remote data positions d_{rem} in Eq.5.9 are replaced as $d_{rem,fro} = wl_n - w_N \cdot s_{li}$ and $d_{rem,end} = 2 \cdot w_N \cdot s_{li}$, the dependent data size $ds_{dep} = w_N \cdot s_{li}$.

The communication scheduling process follows Algorithm 13. Within an FPGA node, the



(0,0)
(0,1)
(a)
slices of data at z dimension
kernel data
front local data
end local data
front remote data
end remote data
FPGA0
FPGA1
FPGA2
(b)
consume time
valid arrive time
end remote data arrive too early when unscheduled, overwriting front local data
front remote data will arrive too late, when communication bandwidth is not high enough.
t
t+2
time
(c)
Tdel,dat
Tdel,com
computation time
arrive time
(d)

Figure 5.4: (a) Decomposed data for three FPGAs, and the corresponding communication and computation operations in time dimension if (b) unscheduled and (c) scheduled. Each grid in the figure represents one data slice. Data dependency, valid times and scheduled delays are labelled in figure.

two dependent data regions share the same computation delay $T_{del,cmp}$, while the data access delay $T_{del,dat}$ can be scheduled separately. The scheduling algorithm traverses all involved FPGA nodes until all the timing constraints are met. For the example in Figure 5.4(a), before scheduling, (1) the `front remote data` in FPGA0 and FPGA1 are on the edge of the latest timing constraints, and will arrive too late if the communication throughput is not as high as computation throughput. In addition, (2) the `end local data` in FPGA0 and FPGA1 are updated too early, violating the earliest timing constraints (see Figure 5.4(b)). In this example, we assume the communication throughput between neighbouring FPGAs is one third of the computational capacity, and $w_N = 1$. After scheduling, (1) the scheduling algorithm first adds the computation time of two data slices $T_{del,cmp} = \frac{2s_{li}}{P_i}$, to ensure the `front remote data` in FPGA0 and FPGA1 will not arrive too late. In addition, (2) a $T_{del,dat} = \frac{2s_{li}}{P_i}$ data access delay is inserted to the update time of the `end local data` in FPGA0 and FPGA1, to meet the earliest timing constraints.

5.5 Benchmarks

5.5.1 Bond Option Pricing

Monte Carlo simulations are widely used in the finance industry to model interest rate to price fixed income products. In the past two decades, the field has evolved from modelling a single instantaneous interest rate [HbL29] to modelling the dynamics of an entire forward rate curve [HJM05]. A forward rate curve is modelled as:

$$\mu(t,T) = \sigma(t,T)\int_t^T \sigma(t,u)du \tag{5.14}$$

$$df(t,T) = \sigma(t,T)\int_t^T \sigma(t,u)dudt + \sigma(t,T)^T dW(t) \tag{5.15}$$

where $f(t,T)$ is the forward rate at time T started from time t; $\sigma(t,T)^T$ is the forward volatility column vector; $W(t)$ is a random variable under standard normal distribution. For each Monte Carlo path, a random $W(t)$ is used to construct a forward rate curve. The generated forward curves are used to value fixed income financial products, as shown in Algorithm 14.

A bond option is a financial instrument which provides the owner of the option with the right to buy or sell a bond at a fixed price K in the future. A call option allows owners to buy asset, while a put option allows owners to sell asset. Based on the valued price in Algorithm 14, the payoff of the bond option at time T $v(t,T)$ can be expressed as:

$$v(t,T) = \max(exp(-\int_t^T f(t,u)du) - K, 0) \tag{5.16}$$

5.5.2 Reverse Time Migration

Reverse Time Migration (RTM) is an advanced seismic imaging technique to detect terrain images of geological structures, based on the Earth's response to injected acoustic waves. The wave propagation within the tested media is simulated forward, and calculated backward,

Algorithm 14 A single Monte-Carlo path of bond option pricing.

```
Input: f(0,T)=initial forward curve, $\sigma$ volatility model
Output: f(t,T)= forward surface
1: for $t = 0 \leftarrow t_{max}$ do
2:     for $T' = 0 \leftarrow T'_{max}$ do
3:         Calculate drift: obtain $\gamma(t, T)$ and get $\mu(t - \delta t, t + T')$ with Eq.5.14
4:         Update forward Surface: obtain $f(t, t + T')$ using Eq.5.15
5:         Price derivative: use $f(t, t + T')$ to price the target derivative d
6:     end for
7:     Price derivative: use results d to price the target derivative
8: end for
```

forming a closed loop to correct the velocity model, i.e. the terrain image. The propagation of injected waves is modelled with the isotropic acoustic wave equation [AP+11]:

$$\frac{d^2 p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \tag{5.17}$$

The propagation involves stencil computation, as the partial differential equation is approximated with the Taylor expansion. A fifth-order approximation is implemented in our experiment.

5.6 Results

Starting from simple design descriptions, the proposed approach generates runtime scalable designs for reconfigurable systems. We evaluate the developed designs in three aspects: resource exploitation, design scalability and runtime adaptivity, which respectively reflect the efficiency of compile-time optimisation, the communication model, and the runtime scaling process proposed in this chapter. Hardware designs are described with MaxCompiler version 2012.1, implemented on Xilinx Virtex-6 SX475T FPGAs, each hosted by one of the four MAX3424A systems in an MPC-C500 computing node from Maxeler Technologies. The clock frequency is 100 MHz.

5.6.1 Resource exploitation

We evaluate the resource exploitation in terms of resource usage and achieved design throughput. Resource usage and design throughput of the optimised designs are presented in Figure 5.5 and Figure 5.6. The resource usage is normalised against available resources, and the resource usage when design parallelism is 0 indicates the resources consumed by communication infrastructures (I).

The Bond Option Pricing (BOP), driven by Monte-Carlo (MC) paths, is inherently parallel. As shown in Figure 5.5, the performance and the resource usage increase linearly with the number of replicated data-paths. Bounded by LUT usage, 26 MC data-paths are replicated, achieving 52.7 GFLOPS throughput.

For the RTM design, before off-chip memory channels are saturated by $p_{ar} = 16$, the replicated data-paths generate one results per data-path per clock, with design throughput and data-path resource usage scaled linearly. Temporal blocking ratio tk is increased to 2 when the memory bottleneck is hit. One more on-chip memory with 16 attached data-paths are replicated, doubling the performance as well as resource usage. Design variables par, tk, sk_x and sk_y of the optimised design are respectively configured as 16, 2, 6 and 5. The optimised design consumes 270816 LUTs, 323134 FFs, 952 DSPs and 989 BRAMs, with the optimisation model estimating the design to consume 255936 LUTs, 357120 FFs, 806 DSPs and 947 BRAMs. For both applications, the optimisation model can capture variations in resource usage wth more than 90% accuracy, and the design is optimised to fully utilise on-chip and off-chip resources.

Design performance is listed in Table 5.2. Reference single-device designs include parallelised CPU designs executed on a 4-core Intel i7-870 CPU, customised GPU designs running on an NVIDIA Tesla C2070, a GPU design optimised by NVIDIA [Mic09] and customised for NVIDIA Tesla C2070, and a FPGA design developed with MaxGenFD [PBD^{+}13]. Unlike the optimisation for BOP, the optimisation techniques for RTM come with overhead. The actual performance of the optimised RTM is reduced from 156.8 GFLOPS to 130.67 GFLOPS, due to the additional data introduced by spatial and temporal blocking. The performance of CPU and

GPU designs is limited by their fixed data-presentation formats and general-purpose memory system. For the RTM design, runtime profiling shows that the optimised GPU design can only achieve 35% memory efficiency, i.e., loading one new data needs 3 clock cycles. Performance of MaxGenFD design is limited by the memory bandwidth due to lack of temporal blocking in its optimisation configurations. The optimised designs are up to 1.4 to 11.2 times faster and 1.7 to 17 times more power efficient than the reference designs.

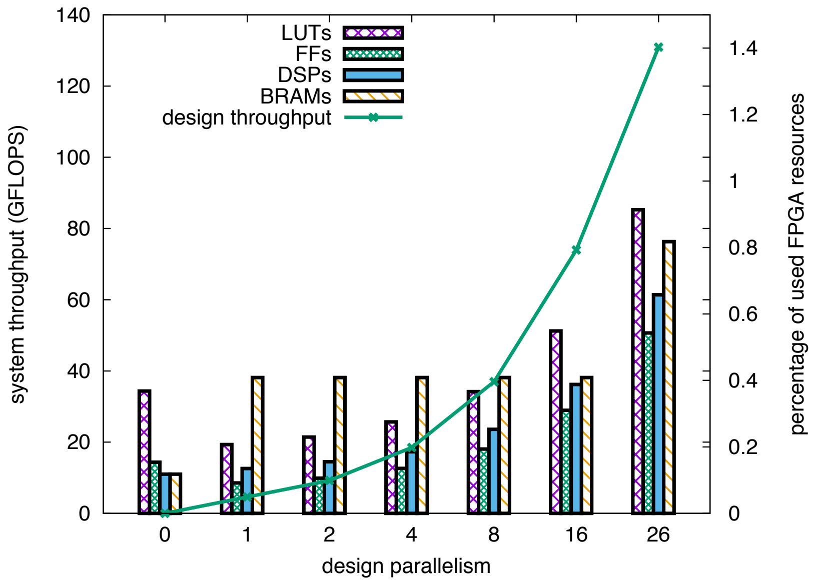


Figure 5.5: Design throughput and resource usage of the BOP design. The design model increases design parallelism to 26 until no more data-paths can be accommodated. Customised RNGs are implemented to provided input data to the replicated data-paths.

5.6.2 Design scalability

Design scalability when a reconfigurable design uses multiple FPGAs reflects effectiveness of the asynchronous communication model. For a reconfigurable system based on the MPC-C500 computing node, point-to-point communication channels with 3.2GB/s bandwidth are provided to support inter-FPGA data exchange, while inter-node data in FPGAs are exchanged through 1GB/s PCI-e channels to CPUs, and then moved through 1GB Ethernet channels to the target node. Therefore, the available communication bandwidth in a dynamic design is 1GB/s (bw=1GB/s).

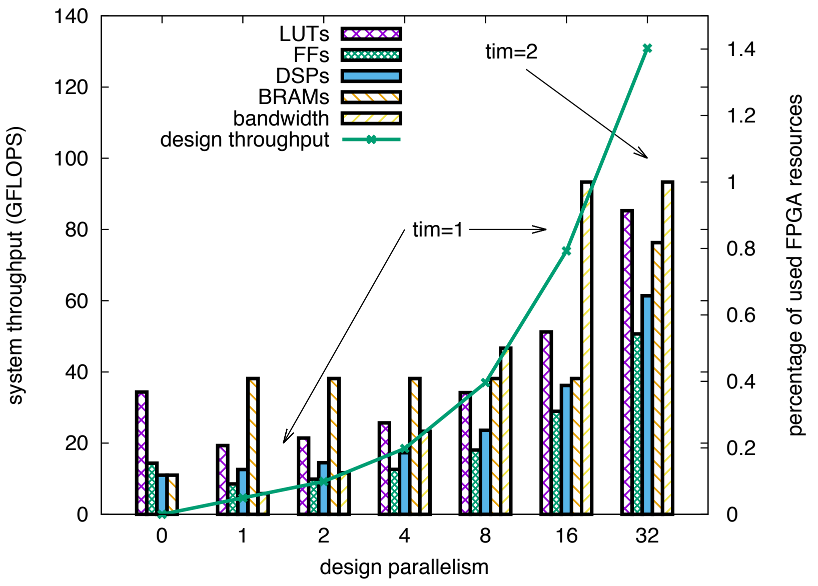


Figure 5.6: Design throughput and resource usage of the RTM design. The optimisation increases design parallelism to 16 until the bandwidth bottleneck is hit, and increase the temporal blocking ratio to utilise left resources.

For the BOP design, no communication operations are involved, and therefore linear scalability is achieved when multiple FPGAs are used. For RTM designs, in the asynchronous communication model, the computation delay $T_{del,com}$ in involved FPGAs is scheduled to be 10 data slices to reduce the bandwidth requirement to 0.4 GB/s, with margin factor $M = 2$. Data access delay $T_{del,dat}$ is scheduled to ensure local halo data are consumed before being overwritten. Limited by available FPGAs in current platform, our current design scales up to 4 FPGAs. Based on computation throughput of utilised FPGAs and available bandwidth, performance of the dynamic design when more FPGAs are involved is simulated. Table 5.2 lists the simulated and measured results for multi-FPGA designs. Previous large-scale designs on Blue Gene/P [PLL+12], Blue Gene/Q [LM13] and Cray XK6 [RMNM+12]. are also introduced to provide comparison. As shown in Table 5.2, the measured results scale in accordance with simulated results, and overall design throughput reaches 0.49 TFLOPS when 32 FPGAs are involved, outperforming the reference designs by 2 to 88 times. Besides throughput, power consumption in large-scale systems determines the maintenance cost such as cooling infrastructures and electricity bill, and plays an important role in large-scale designs. Power efficiency numbers are not provided in previous work [PLL+12, LM13, RMNM+12]. If we make a conservative consumption that the Tesla X2090 GPUs in Cray XK6 consumes the same power as

Tesla C2070 design in Table 5.2, the dynamic design is 5.2 times more efficient than the stencil design running on Cray XK6, with all infrastructure power consumption included.

Table 5.2: Single-device and multi-device performance comparison.

	Bond Option Pricing			Reverse Time Migration			
single-device	CPU[1]	GPU[2]	FPGA[3]	CPU[1]	GPU[2]	MaxGenFD	FPGA[3]
frequency (GHz)	2.93	1.15	0.16	2.93	1.15	0.1	0.1
throughput (Gflops)	4.71	38.6	52.7	13.3	83.4	71.3	130.67
speedup	1x	8.2x	**11.2x**	1x	6.27x	5.36x	**9.82x**
power (Watt)	182	240	137	245	369	137	142
efficiency (Gflops/Watt)	0.03	0.16	0.38	0.054	0.23	0.52	0.92
efficiency	1x	5.3x	**12.7x**	1x	4.2x	9.6x	**17.0x**
multi-device	CPU[1]	GPU[2]	FPGA[3]	Blue Gene/ Q [LM13]	Cray XK6 [RMNM+12]	MaxGenFD	FPGA[3]
4-node (Gflops)	12.9	154.4	210.8	76.8	387	196.8	523
16-node (Gflops)	n/a	617.6	843.2	307.2	1548	n/a	2091
64-node (Gflops)	n/a	2470	3372.8	1289	6192	n/a	8368
speedup	n/a	1x	**1.37x**	1x	4.8x	n/a	**6.49x**

[1] Reference single-device CPU designs are running on a 4-core Intel i7-870 CPU, and the multi-device CPU designs are based on a Dell PowerEdge R610 machine, with 24 Intel(R) Xeon(R) X5660 cores running at 2.67GHz.

[2] The benchmark GPUs are NVIDIA Tesla C2070. The 64-node performance is linearly extrapolated based on single-device performance. For RTM, previous design optimised by NVIDIA [Mic09] is used and customised for Tesla C2070.

[3] Limited by available resources, performance for FPGA designs with more than 4 FPGAs is simulated. When 1 to 4 FPGAs are involved, measured performance confirms the simulated results. The communication bandwidth is set based on the measured bandwidth of the existing system: 1GB/s via Ethernet and PCI-e channels, and 3.2 GB/s via dedicated inter-FPGA channels.

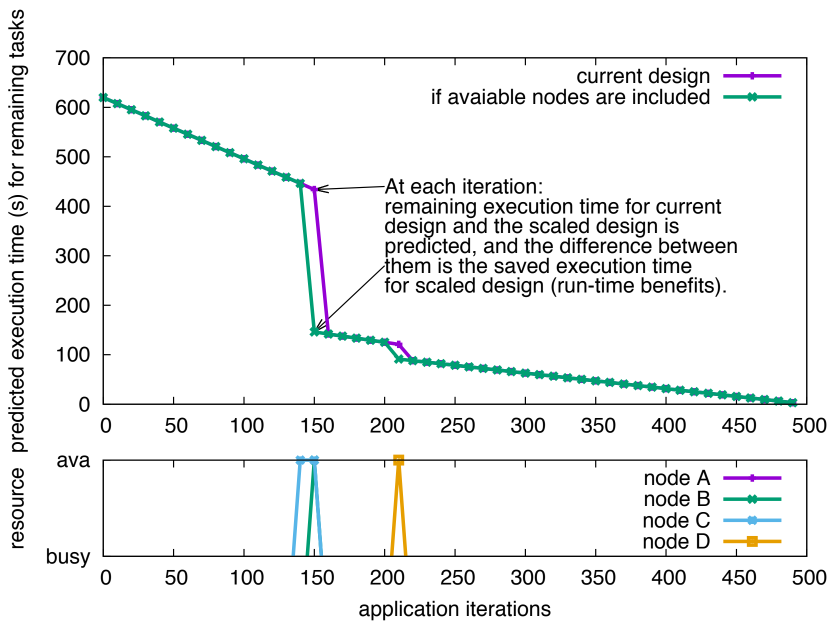


Figure 5.7: Evaluation and prediction of the runtime performance model during one of the test case, at the application iteration (time step) dimension. The resource status is measured from target system. 'ava' stands for available, and 'busy' indicates the FPGA node is currently not available. The accurate predication of runtime benefits and overhead enables a dynamic design to scale when additional nodes become available (such as at iteration 148, 152, and 212).

5.6.3 Runtime adaptivity

We evaluate the runtime adaptivity of the developed designs in terms of design performance and device-level hardware efficiency, when the developed dynamic design is mapped into the reconfigurable system. Since an RTM design has more complex runtime overhead and communication operations, we use the RTM design to demonstrate how a dynamic design scales during runtime. For the available 4 FPGAs, static designs with 1, 2, 3 and 4 device-level parallelism are developed and executed to provide comparison. Runtime status during 10 separated time periods is measured and used as 10 test cases in this experiment. Figure 5.7 and 5.8 demonstrate the performance evaluation process during runtime. The runtime performance model predicts the execution time for remaining tasks for current design as well as the scaled design. When new nodes become available, the difference between the two predictions indicates the runtime benefits. FPGA node A is available when the application is launched, and node B, C and D are released by other computational tasks at 150, 142 and 209 iterations, respectively.

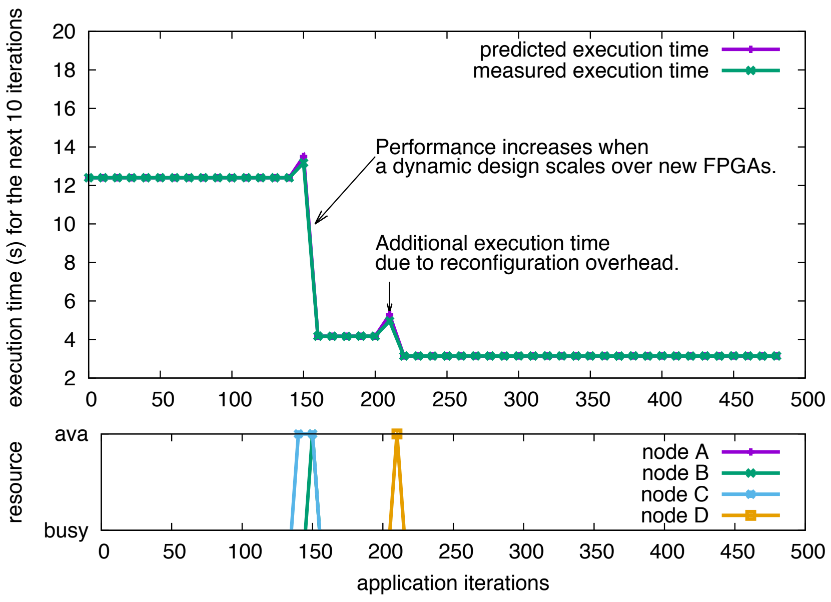


Figure 5.8: Predicted and measured execution time for each 10 iterations, in the runtime scenario in Figure 5.7

Although node C becomes available earlier than node B, the dynamic design first expands when B is released, due to a lack of communication channels between node A and node C. If node B and C are included in the dynamic design, execution time for the following tasks is reduced by 357.4 s, with 0.71 s runtime overhead introduced. As the benefit outweighs the overhead, node B and node C are reconfigured to cooperate with the existing node A. The runtime scaling process (see Algorithm 11 on page 128) redistributes context data into the new nodes, and updates design variables with Algorithm 13 (on page 134) to ensure linear scalability and correct functionality when the dynamic design scales. Similarly, the node D is included dynamically when it becomes available. As shown in Figure 5.8, the measured performance aligns with predicted execution time for remaining tasks, showing high accuracy of the performance model. Moreover, as the performance use a general approach to estimate performance with data size and peak throughput, the high model accuracy indicates optimised designs approximate their peak performance. Device-level parallelism for static design using 1 FPGA is limited to 1, while the static designs using more FPGAs need to wait for release nodes to start. The dynamic design finishes 490 time steps in 297 seconds, outperforming the static designs by 1.67 to 2.72 times.

Hardware efficiency is calculated with measured performance and the theoretical performance upper bound, where the theoretical performance refers to the overall performance if FPGAs

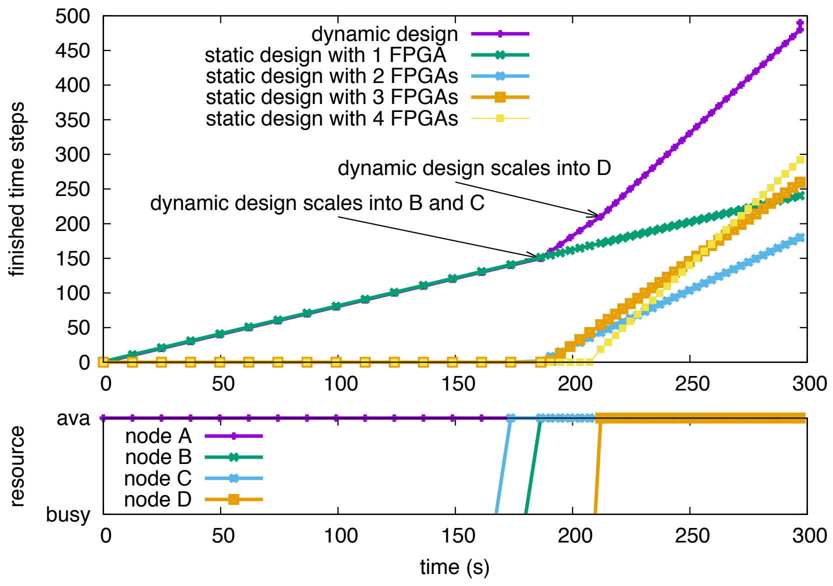


Figure 5.9: Performance of dynamic design and static design with 1, 2, 3, 4 FPGAs, at the time dimension.

are used in the design once released by another application. The measured performance and resource utilisation for the 10 test cases are shown in Figure 5.10. We apply the first 5 cases to the BOP designs, and test the RTM designs with the remaining runtime cases. The averaged hardware efficiency for the dynamic design is 0.91. The gap between the achieved hardware efficiency and the optimal efficiency level 1 is introduced by the reconfiguration overhead and communication infrastructure. As shown in the test case in Figure 5.9, node C remain idle until the dynamic design expands into node B, as there is no communication channels between node A and node C. Resource utilisation for static designs is limited between 0.4 to 0.49. In other words, limited by mismatch between compile-time exceptions and runtime environment, half of resources in the system remains idle. Due to the high resource utilisation of the dynamic design, averaged system performance is 1.82 to 2.28 times faster than the static designs.

5.7 Related Work

At the system level, previous work on Monte-Carlo simulations and stencil computations focused on hardware acceleration and the development framework. Common techniques to accelerate high-performance accelerations include precision optimisation, and architecture cus-

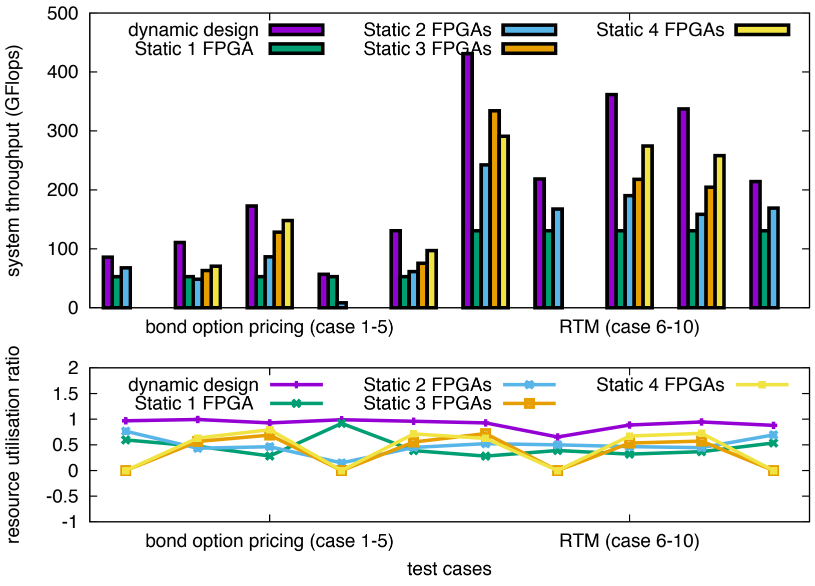


Figure 5.10: Design performance and hardware efficiency for the 10 test cases.

tomisation. For Monte-Carlo simulations, customised RNGs [TL08b] and precision optimisation techniques [CTJ+12] have been proposed to reduce the resource usage of Monte-Carlo data-paths. For stencil applications, customised communication patterns in CPU-based systems [PLL+12, LM13], data reuse and communication scheduling techniques for GPU-based systems [Mic09, PF10b, RMNM+12], and customised memory architectures [FC11b] and data-paths [S+11] for FPGA-based systems have been proposed. These design techniques, efficient as they are, require high-level expertise and manual optimisation.

Development frameworks for Monte-Carlo simulations and stencil computations enable non-expert developers to utilise the various computing resources. These frameworks often handle code generation and design parameter tuning to improve productivity. A multi-level customisation framework [JDT+12] for financial Monte-Carlo simulations supports various applications by allowing users to tune reconfigurable design parameters at different levels. Parallel GPU codes are generated in [HPS12] to optimise stencil applications based on properties of GPU architectures. Spatial blocking is optimised to balance workload among parallel threads [KBB+07], and auto-tuners are built to search for the optimal blocking strategies for various resources [DMV+08] and data structures [KCO+10]. Temporal blocking is supported with a blocking algorithm [NSC+10], and the design space is searched with various searching algorithm to minimise execution time for CPU and GPU designs. The auto-tuners, which are widely used for general-purpose processors such as CPUs and GPUs, require a long execution time to traverse their search space. Runtime construction and adaptation of designs requires rapid update in design configurations, therefore the auto-tuning process is not suitable. MaxGenFD provides a design interface for users to specify design parallelisation and spatial blocking ratios during compile time. Such a semi-automatic approach requires going through the time-consuming synthesis tool chain multiple times to optimise designs, and design parallelism in utilised FPGAs is statically configured.

Compared with previous work, the work in this chapter handles a new optimisation opportunity: additional computing resources become available during the execution of a reconfigurable design, and will remain idle if static designs are used. We introduce runtime reconfiguration at the system level to adapt reconfigurable designs to such resource availability variations. An automatic design approach is proposed to handle the system-level design issues with compile-time optimisation and runtime scaling. Optimisation techniques, either general or domain-specific, are integrated in design models and communication models to ensure the generated runtime reconfigurable designs can achieve high performance.

5.8 Limitations and Future work

The limitations of the current design approach mainly come from its single-task considerations: a runtime reconfigurable design tends to occupy all available resource during its execution, which may not be the optimal solution if maximum overall performance of multiple tasks is targeted; idle nodes due to lack of communication channels to existing runtime reconfigurable design can be occupied by other computational tasks, which can further increase resource utilisation. In the future, dynamic design methods at multi-task and multi-user layers will be built on top of current system-level approach, to exploit more complex runtime scenarios.

5.9 Summary

For large-scale reconfigurable systems, the effectiveness of conventional static design methods that pre-define communication patterns and hardware configurations is limited by unpredictable runtime conditions. This work is inspired by the experience that a design using multiple FPGAs normally needs to wait for released devices to execute, even in a small-scale reconfigurable system. As a system scales and the number of launched computational tasks increases, we believe this will become a bottleneck in large-scale designs that not only limits the overall design execution time but also the hardware efficiency of the underlying infrastructure.

In this chapter, we propose a novel approach that statically optimises target applications for various FPGA nodes, and dynamically constructs executable design when resource status varies. Experimental results show that high throughput and significant resource utilisation can be achieved with dynamic designs, which can dynamically scale into nodes that become available during their execution. When statically optimised and initialised, the dynamic design is 1.4 to 11.2 times faster and 1.8 to 17 times more power efficient than reference CPU, GPU, MaxGenFD, Blue Gene/P, Blue Gene/Q and Cray XK6 designs; when dynamically scaled, the hardware efficiency of the dynamic design reaches 91%, which is 1.8 to 2.3 times higher than their static counterparts.

Theoretically, the system-level approach can benefits any applications that (1) can improve design performance by using more computational resources, and (2) are launched into a system that resource availability varies during runtime, such as matrix processing, N-body simulation, and K-means clustering [NTL11]. In order to deeply exploit available resources for these applications, more domain-specific aspects need to be developed. As an example, N-body simulation requires N-to-N communication patterns, and therefore needs customised specifications in the communication models of the system-level approach.

Chapter 6

Conclusion

6.1 The Challenge

As discussed in the introduction (Section 1.1), compared with general architectures such as CPUs, customised architectures developed for a specific application can better exploit instruction-level parallelism. Moreover, due to the overhead to support reconfigurability, there is a performance gap between designs mapped into reconfigurable fabric and designs implemented as ASICs (assuming the same design techniques and technology are used). In practice, the efficiency of computing architectures is reduced by three limitations.

1 Due to limited hardware resources, the computational efficiency reduces as the problem size increases. The problem size can be related to the number of instructions (instruction cache miss), the processed data size (data cache miss), or the computation kernel complexity (large instruction delay).

2 In order to cover a wide range of computational scenarios, computer architectures often compromise hardware efficiency. As an example, compared with ASIC designs that only need to cover one application, GPPs often achieve lower performance given the same technology and resources as multiple applications need to be supported.

3 Since there is often little runtime information during application development, application developers compromise computational efficiency to handle possible runtime scenarios (with operations such as branches).

The first limitation can often be resolved by optimisation techniques, such as loop tiling, temporal blocking, and design parallelisation. For the latter two issues, despite the optimisation techniques applied, the more scenarios an architecture needs to support, the lower hardware efficiency can be achieved. The proposed runtime reconfiguration approaches enable a reconfigurable design to only support one runtime scenario of a computational scenario at each time, and eliminate the idle resource units that were introduced to ensure design generality.

We summarise the throughput results collected in this thesis in Figure 6.1, where the throughput of optimised software implementations are used as reference. For each benchmark application, there are four different implementations.

- Optimised CPU designs, even parallelised to use multiple cores and optimised with vendor compilers, achieve relatively low design throughput, mainly due to the high generality of CPUs. We use the CPU design throughput as a reference performance of an application (i.e. performance improvement ratio is 1).
- Static reconfigurable designs improve design throughput as the implemented hardware is customised for a specific application. The static design throughput shows the performance level before applying the proposed runtime reconfiguration approaches.
- ASICs achieve higher performance compared with static reconfigurable designs. The reconfigurability of FPGAs leads to increased design area and reduced clock frequency for the same problem. Limited by the large design efforts to develop ASICs, we use averaged performance gap reported in previous work [KR07] to estimate ASIC performance.
- Dynamic reconfigurable designs are optimised at the three design levels with the proposed approaches. For each application, we show the performance improvements gained at the three design levels step by step.

While the design throughput of CPU designs and reconfigurable designs in Figure 6.1 is measured from design experiments, we estimate ASICs performance based on the experiment results in [KR07], which show ASICs on average consume 18 times less area and run at 3 times higher clock frequency, compared with reconfigurable designs. We make three assumptions to estimate ASIC design performance: (1) all applications have the same performance gap between ASIC designs and reconfigurable designs; (2) the measured results based on a 90-nm technology [KR07] are still valid for 40-nm technology; and (3) application throughput increases linearly with parallelism P, i.e., the 18 times reduction in area leads to a 18 times increase in throughput. Therefore, as shown in Figure 6.1, the ASIC designs are 54 times faster than the conventional reconfigurable (static) designs. In practice, the first and the second assumptions are difficult to fulfil, and there will be fluctuation in the performance gap, which varies from one application to another. In addition, not all applications have linear throughput scalability. As an example, for stencil applications, replicating more data-paths with temporal blocking introduces additional data to process. The third assumption therefore overestimates the performance of ASIC designs. However, the performance variations due to the unmet assumptions are often minor [KR07]. The estimated ASIC performance in Figure 6.1 works as an indication for the ASIC design performance level.

6.2 Performance Improvements and Technology Gap

6.2.1 Performance Improvements for Eliminating Idle Resource Units

In this thesis, we focus on the reduced hardware efficiency due to supporting various runtime scenarios, and improve application performance by dynamically reconfiguring the optimised designs as runtime scenarios vary. We consider the idle resource units in a reconfigurable design as runtime reconfiguration opportunities. The idle resource units are divided into three levels. (1) At the circuit level, algorithm parameters are tuned to generate customised arithmetic operators. When the constant input change during runtime from time to time, the optimised arithmetic operators are dynamically reconfigured to achieve the same operator generality.

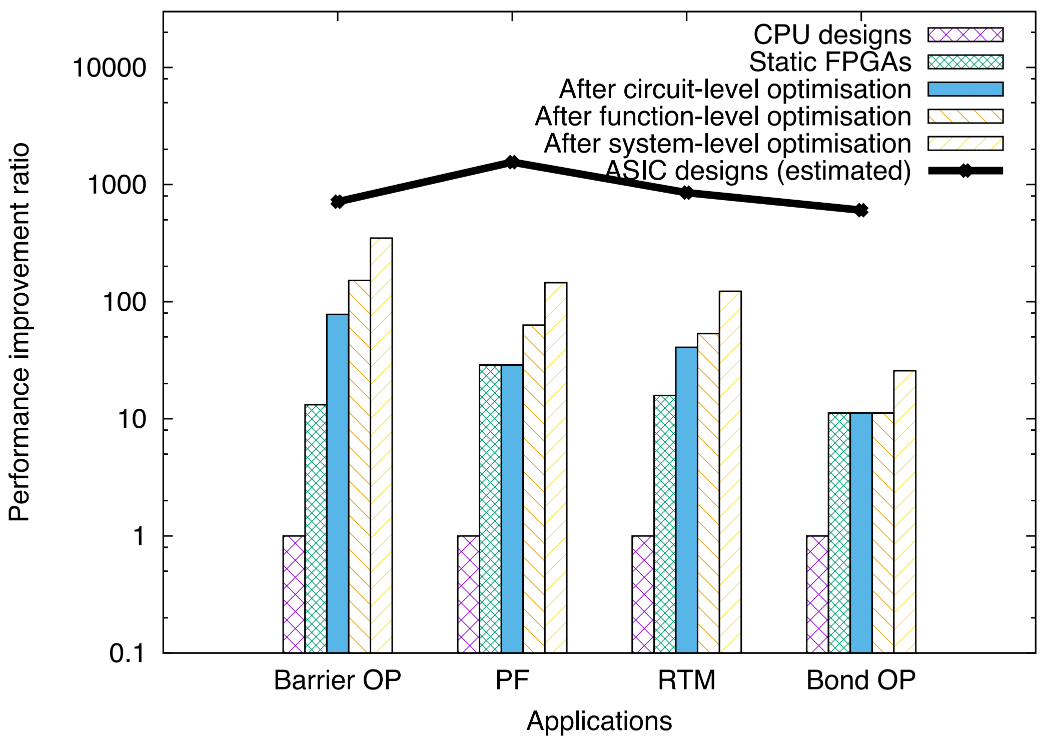


Figure 6.1: Overall design performance improvements after optimisation techniques at the three design levels applied. The option pricing used at the circuit level and the barrier option pricing used at the function level are considered as the same function, as the option pricing is one function in the barrier option pricing application.

(2) At the function level, application functions are separated into different configurations, such that only active functions are kept on-chip. Since idle functions are replaced with active resource units, the processing capacity in runtime scenarios is increased. (3) At the system level, reconfigurable designs dynamically adapts to resource availability variations. FPGAs that are busy when a design is launched can become available during the execution of the design. By dynamically scaling reconfigurable designs into FPGAs that become available during runtime, reconfigurable designs can efficiently utilise system resources that otherwise would remain idle.

After eliminating idle resource units with runtime reconfiguration, the performance improvements at the three levels are presented in Figure 6.1. The circuit-level technique is applicable to finite-difference algorithms, and the function-level and the system-level optimisation techniques cover both stencil computation and Monte-Carlo simulations. The performance improvements at the circuit level, the function level and the system level are respectively up to 5.9, 2.19 and 2.28 times. Among the benchmark applications, both Barrier Option Pricing (BOP, the single function kernel is named Option Pricing at the circuit level) and Reverse Time Migration (RTM) benefit from all the optimisation techniques. After optimisation, the runtime reconfigurable designs achieve up to 26.1 times speedup.

Given the estimated performance of ASICs, the dynamic designs proposed in this thesis almost halve the performance gap between reconfigurable designs and ASICs (26.1 out of 54). The remaining 2 times performance gap is mainly due to clock frequency. In [KR07], ASICs on average operate at 3 times higher clock frequency compared with reconfigurable designs, while the dynamic and the static designs in this thesis operate at the same clock frequency (100 MHz). In terms of reconfiguration overhead, as shown in the runtime evaluation results at each design level, the dynamic designs achieve maximum performance for large-scale applications with relatively long execution time. For such large-scale applications, the impacts of reconfiguration overhead are small. As an example, if we assume no reconfiguration time is required at the system level, the achieved hardware efficiency can be increased from 0.91 to up to 1, in others word, removing reconfiguration overhead only increases design performance by up to 1.09 times.

6.2.2 Partial Reconfiguration: the Good and the Reality

While runtime reconfiguration overhead has a small impact on the performance of large-scale applications, applying the proposed approach to small-scale applications calls for fine-grained reconfiguration operations (i.e. small reconfiguration time). As an example, if an application needs to be reconfigured every second, the large reconfiguration time of FR designs (typically 0.8 s) will dominate the overall execution time. Partial Reconfiguration (PR) is often considered as an effective way to reduce reconfiguration time. Compared to full reconfiguration, the fact that only a part of an FPGA is updated reduces reconfiguration time and keeps the infrastructure modules such as memory controller active during reconfiguration. In our experiment, partially reconfiguring a clock region in a Virtex-6 SX475T FPGA takes 50 ms instead of 0.8 s. In practice, there are mainly two limitations for the use of partial reconfiguration.

1 Reconfiguration time. The reconfiguration time of PR designs depends on partial config-

uration file size. Therefore, when a large portion of FPGA needs to be updated, partially reconfiguring a reconfigurable design still takes a long time. For example, at the circuit and the function level, it is often the case that more than 50% of FPGA resources need to be reconfigured, and therefore a partial reconfiguration operation takes around 0.4 s, which is still too long if we reconfigure designs every second. At the system level, as available FPGAs are previously used by a different application, the whole configuration file needs to be updated, and thus applying partial reconfiguration does not reduce reconfiguration time. In addition, when a very small part of FPGA needs to be updated, the partial reconfiguration time is bounded by its reconfiguration granularity, and therefore has a minimum level (often a few hundreds cycles).

2 Design clock frequency. Supporting PR modules in a reconfigurable design reduces design clock frequency, especially when communication infrastructures such as memory controllers need to be placed and routed on-chip. In a reconfigurable design, memory controllers need to be placed close to I/O pin columns in FPGAs, and need to be scattered across the whole FPGA chip to achieve the optimal clock frequency. For the RTM application, labelling function nodes B and C in Figure 4.10(c) as PR modules reduces the clock frequency from 100 MHz to 60 MHz, which outweighs the reduction in reconfiguration time.

We evaluate the potential of PR designs in Figure 6.2, where the x-axis is reconfiguration frequency f_{rec}, the y-axis is clock frequency reduction f_{red}, and the z-axis shows the hardware efficiency. f_{red} indicates the reduction in clock frequency after applying runtime reconfiguration techniques. (1) For FR designs, there is no reduction in clock frequency, since each optimised reconfigurable design goes through the standard synthesis flow. The measured reconfiguration time is 0.8 s, with another 1-2 s memory transfer time. We take the upper bound and assume the reconfiguration time to be 2.8 s. (2) For PR designs, the clock frequency is inevitably reduced. Since the reduction in clock frequency is application-specific, we show 10 cases in the y-axis with their clock frequencies reduced from 100% to 10% ($f_{red} = 1 \sim 0.1$). For reconfiguration time, we make an optimistic assumption that all PR circuits can be grouped in one clock region.

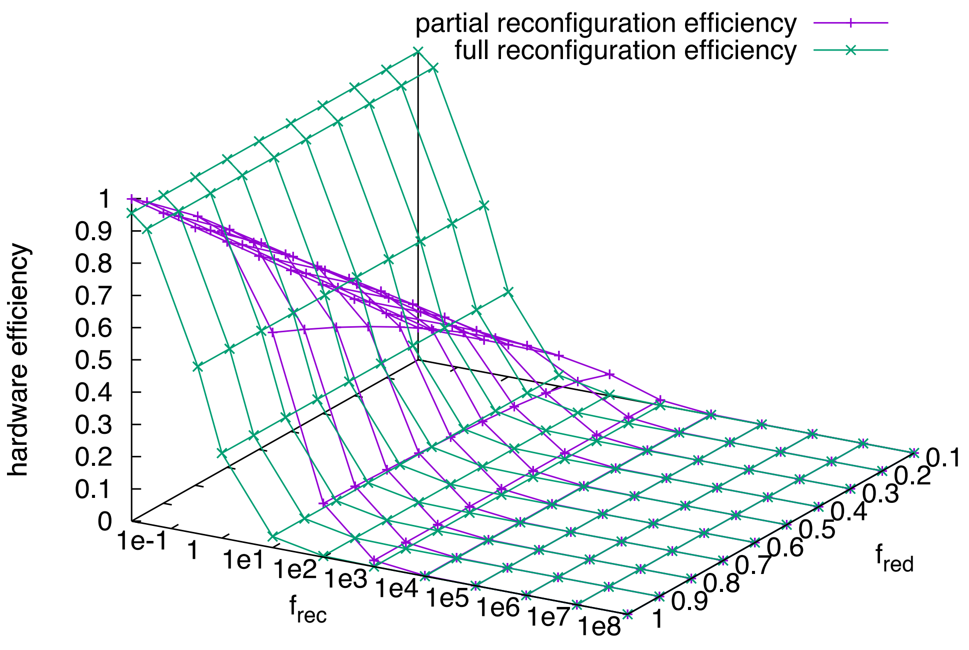


Figure 6.2: Hardware efficiency for runtime reconfigurable designs with FR and PR, when reconfiguration frequency f_{rec} and clock frequency reduction f_{red} increase. FR designs have no impacts on clock frequency.

Therefore the reconfiguration time is 0.05 s. The reconfiguration frequency f_{rec} indicates how frequent reconfiguration operations happen during runtime, and can be calculated as the reverse of the average execution time T_{exe} between two consecutive reconfiguration operations.

$$f_{rec} = \frac{1}{T_{exe}} \tag{6.1}$$

For the large-scale problems studied in this thesis, the execution time T_{exe} is at the scale of 60 s or more. In the x-axis, we reduce the execution time from 60 s to 1 clock cycle (10 ns for a 100 MHz clock frequency).

The hardware efficiency E, as defined in Eq.1.1, is the ratio between the achieved performance and the theoretical peak performance. As described in Eq.1.4 and 1.5, we calculate the optimal performance assuming all implemented data-paths are `well behaved` and no overhead is involved. In order to compare the performance of FR designs and PR designs, we estimate the overall execution time T_{mes} by accumulating the average execution time T_{exe} between consecutive reconfiguration operations and reconfiguration overhead:

$$T_{mes} = \frac{T_{exe}}{f_{red}} + T_{rf} \quad T_{the} = T_{exe} \tag{6.2}$$

$$E = \frac{T_{the}}{T_{mes}} \tag{6.3}$$

where f_{red} accounts for the reduction in clock clock frequency, and T_{rf} is the reconfiguration time. $f_{red} = 1$ for FR designs, and $f_{red} < 1$ for PR designs. The theoretical execution time T_{the} is calculated with full clock frequency and no reconfiguration overhead. We assume the implemented designs have been customised to achieve peak performance for each runtime scenarios ($T_{the} = T_{exe}$).

For applications with low-frequency reconfiguration operations, FR designs approximate the optimal efficiency 1, while the efficiency of PR designs is determined by the reduction in clock frequency f_{red}, since reconfiguration time is negligible in this case. As shown in Figure 6.2, for reconfiguration frequency lower than 0.05 (average execution time higher than 20 s), the efficiency of FR designs is more than 90%. When the reconfiguration frequency is beyond 1000, the reconfiguration overhead start to dominate the execution of both FR and PR designs, which reduces the efficiency to a very low level.

For PR designs to outperform FR designs, the reconfiguration frequency needs to be between 1 and 100, and the clock frequency ratio needs to be higher than 80%. In practice, such medium-frequency reconfiguration operations are rare. T_{exe} is at minutes to hours levels for large-scale applications, while for fine-grained reconfiguration operations hardware circuits need to be updated cycle by cycle. In addition, the clock frequency target is hard to meet for PR designs. To summarise, while requiring lots of design effort to develop, PR designs provide limited improvements in runtime reconfigurable designs, especially for designs that need memory controllers.

6.2.3 Limitations in Runtime Reconfiguration Techniques

By eliminating idle resource units at various design levels, runtime reconfigurable designs approximate the performance of ASIC designs for applications with low-frequency reconfiguration operations. However, as shown in Figure 6.2, the design efficiency significantly reduces as the reconfiguration frequency increases. Both FR designs and PR designs cannot support the applications with high reconfiguration frequencies. Therefore, limited by the current technologies, the proposed approaches cannot be applied to applications in the gap area, i.e., applications with high reconfiguration frequencies.

The motivation for high-frequency reconfiguration operations comes from idle resource units in computational intensive kernels, which need to be updated iteration by iteration. As an example, for a nested loop, there are conditional operations or dynamic pointers in the fastest loop. Therefore, multiple operations can be executed and data access can point at different positions during runtime, depending on runtime variables. As these operations are in the fastest loop, when implemented in hardware, these operations need to be modified up to cycle by cycle. Applying runtime reconfiguration can reduce design area as only the circuits active at current cycle are implemented. Moreover, these dynamic operators — if / case operations and dynamic pointers — are common in high performance applications. Thus supporting high-frequency runtime reconfiguration operations would enable reconfigurable designs to efficiently accommodate applications previously considered not preferable to hardware implementations.

In order to apply runtime reconfiguration techniques to applications with high-frequency reconfiguration operations, The main challenge is to reduce the reconfiguration time to nano-second level. Given the smallest addressable configuration size (1 configuration frame with 3232 bits) and maximum reconfiguration throughput (400 MB/s) in the latest devices [Xila, Xilc], the minimum reconfiguration time is 1.01 μs. When implemented circuits need to be updated cycle by cycle, reconfiguration techniques with such reconfiguration time is far from being useful. Multi-context FPGAs [DeH96, TCJW97] push the reconfiguration time into the nano-second level. However, storing multiple configuration files on-chip increase chip area. We use the area model in [BRM02] to estimate silicon area based on the drive strength of implemented transistors. For a typical multi-context FPGA with 8 replicated configuration memories, the overall chip area is increased by 4.4 times, which limits the benefits gains from applying runtime reconfiguration techniques. This challenge is currently unresolved.

Another barrier to using runtime reconfiguration is the productivity issue. As presented in Chapter 3, 4, 5, to exploit the runtime reconfiguration opportunities in a reconfigurable design, hardware designers need to be aware of runtime reconfiguration from low-level operator customisation to high-level system management, which would be overwhelming even for experienced hardware designers. In order to address this issue, all the approaches proposed in this work are automated, and can be integrated into high-level development tools. We have developed an initial prototype (available at `http://www.doc.ic.ac.uk/~nx210/tools/irue.zip`) for an automatic development tool, which starts from descriptions in the C language, goes into the tool back-end to integrate the function-level and the system-level approaches, and generates hardware descriptions supported by the reconfigurable systems provided by Maxeler Technologies. The circuit-level approach is currently not included, as the VHDL programs generated by FloPoco are not supported by the synthesis tool from Maxeler Technologies. The current tool, while capable of demonstrating the feasibility of the proposed design approaches, has three limitations:

- Supported high-level languages: the current tool supports a subset of C language. Operations such as dynamic pointers and data structures are not supported, due to the low efficiency of the mapped hardware operators.
- Hardware language generality: the current source-to-source translation ends with a specific high-level language (MaxCompiler). Since there are features not supported by the language, such as customised operators at the circuit level, design approaches make use of these unsupported features cannot be integrated into the current tool.
- Design verification: the current tool lacks a systematic approach to verify the generated runtime reconfigurable design.

6.3 Future Work

6.3.1 Approach Extension

The current approaches can be extended by supporting more application domains, exploring more runtime scenarios and enhancing the current tool flow. We investigate two application domains in this thesis: stencil computation and Monte-Carlo simulations. However, the applications that can benefit from the approach are not limited to these two application domains. The potential applications should meet at least two criteria: (1) reconfiguration frequency need to be relatively low, there should be at least 20 s between two reconfiguration operations, and (2) there are idle resource units in the target problem, the idle resource units could be customisable arithmetic operators, idle function modules, or FPGAs with indeterministic availability. For the circuit-level approach to be applicable, the target application should contain parameters that could affect the constant coefficients used in arithmetic operations.

The runtime scenarios in current reconfigurable systems are relatively straightforward: each FPGA accommodates one configuration, and a runtime reconfigurable design tries to utilise all the FPGAs that are available to it. In the future, the potential of more complex scenarios can be studied. Multiple configurations can share the same FPGA to better utilise different on-chip resources. For example, an application using 90% of DSPs and 10% BRAMs can be combined with an application using 10% of DSPs and 90% of BRAMs, to use left over resources. When an FPGA in a reconfigurable system becomes available, the FPGA node can be assigned to the reconfigurable design with the maximum performance improvement, to achieve the global optimum for system performance. These extensions can be built on top of the existing approaches, to further improve the flexibility and the performance of runtime reconfigurable designs.

In correspondence to the limitations in the current tool, future work to complete the tool chain includes:

- Investigating the possibilities of supporting a wider range of C operations with runtime reconfiguration. As discussed in Section 6.2.3, most of the unsupported operations are operations that would introduce a large amount of idle resource units (i.e., significantly reduce hardware efficiency), and applying runtime reconfiguration to these operations calls for reducing reconfiguration time to nano-second level;

- Using a more general hardware language as the target language for the source-to-source translation, for example VHDL or Verilog, and therefore supporting all the existing features of reconfigurable designs;

- Developing verification approaches for runtime reconfigurable designs, and implementing the verification approaches as additional tool modules.

6.3.2 Technique Enhancement

As discussed in Section 6.2.2, the use of partial reconfiguration is limited by the impacts on clock frequency and the minimal reconfiguration time. Possible enhancements to the partial reconfiguration techniques include hardening frequency-sensitive modules and using coarse-grained function units.

Frequency sensitive modules, such as on-chip memory controllers, refer to the hardware modules that are sensitive to available place and route resources in an FPGA. The clock frequency of these modules can be heavily affected by having PR regions in an FPGA. This issue can be resolved by integrating frequency sensitive modules in FPGAs as hard cores Therefore these frequency sensitive modules such as memory controllers can be directed mapped into FPGAs without placing and routing, with a fixed clock frequency.

Coarse-Grained Reconfigurable Architectures (CGRAs) often consist of an array of coarse-grained reconfigurable units interconnected by dedicated communication networks on-chip [MD96, W$^+$07, B$^+$07]. The coarse-grained reconfigurable units lead to fine-grained reconfiguration operations. As an example, to configure such a unit to become an adder, a subtracter, and a multiplier, it only takes 2 configuration bits to switch between different configurations. This leads to smaller configuration frames and less configuration time with the same reconfiguration throughput.

6.3.3 New Reconfiguration Techniques

Despite the possible enhancements discussed before, the fundamental barrier that leads to the technology gap between FPGAs and optimal reconfigurable architectures, in the field of high-frequency runtime reconfiguration, is still unresolved. The reconfiguration time increases linearly with configuration size, and the reconfiguration throughput is bounded by the number of I/O pins dedicated to transferring configuration files. As discussed in Section 6.2.3, if reconfigurable designs can be reconfigured cycle by cycle, a much wider range of applications can be accelerated with reconfigurable architectures. However, reconfiguring circuits cycle by cycle requires updating at least thousands to millions of configuration bits within a clock cycle, which calls for a configuration throughput at least 100 times faster than the latest technology. In order to bridge the current technology gap, fundamentally new reconfiguration techniques need to be developed. Based on our experience in applying runtime reconfiguration to high-performance applications, such techniques need to meet three requirements:

- To be able to update configuration information at the nano-second level, ideally, within one clock cycle.
- To have overhead as small as possible when integrated with the relevant reconfigurable architectures.
- To be compatible with existing synthesis tools and hardware languages, and to be transparent to applications without idle resource units.

To address these requirements, a new memory architecture that supports single-cycle reconfiguration has been proposed, which is designed to support applications with dynamic data access with low overhead [NLW15]. While the details of this architecture is beyond the scope of this thesis, we believe that it is a step towards the development of novel reconfigurable devices that address applications which have proved difficult for the current generation of FPGAs.

Bibliography

[AC86] Arvind and D. E. Culler. Dataflow architectures. *Ann. Rev. Comput. Sci.*, 1:225–253, 1986.

[AN87] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, pages 1–29, 1987.

[AP$^+$11] Mauricio Araya-Polo et al. Assessing accelerator-based HPC reverse time migr:ation. *IEEE Transactions on Parallel and Distributed Systems*, 22:147–162, January 2011.

[ATLJ13] James Arram, Kuen Hung Tsoi, Wayne Luk, and Peiyong Jiang. Reconfigurable acceleration of short read mapping. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*, pages 210–217, 2013.

[B$^+$07] Michael Butts et al. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *FCCM*, pages 55–64, 2007.

[BAS09] Karel Bruneel, Fatma Abouelella, and Dirk Stroobandt. Automatically mapping applications to a self-reconfiguring platform. In *Proc. DATE*, 2009.

[BBD05] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil D. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*, pages 335–340, 2005.

[BdDPT10] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. Multipliers for floating-point double precision and beyond on FPGAs. *SIGARCH Computer Architecture News*, 38(4):73–79, 2010.

[BJLW11] Tobias Becker, Qiwei Jin, Wayne Luk, and Stephen Weston. Dynamic constant reconfiguration for explicit finite difference option pricing. In *Proc. ReConFig*, 2011.

[Bre96] Gordon J. Brebner. A virtual hardware operating system for the xilinx XC6200. In *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, 6th International Workshop on Field-Programmable Logic, FPL '96, Darmstadt, Germany, September 23-25, 1996, Proceedings*, pages 327–336, 1996.

[BRM02] V Betz, J Rose, and A Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic Publishers, 2002.

[BS08] Karel Bruneel and Dirk Stroobandt. Automatic generation of run-time parameterizable configurations. In *FPL*, pages 361–366, 2008.

[BSPM09] Sheetal U. Bhandari, Shaila Subbaraman, Shashank S. Pujari, and Rashmi Mahajan. Real time video processing on FPGA using on the fly partial reconfiguration. In *ICSPS*, pages 244–247, 2009.

[CDW01] Eylon Caspi, Andr DeHon, and John Wawrzynek. A streaming multi-threaded model. In *In Proceedings of the Third Workshop on Media and Stream Processors*, pages 21–28, 2001.

[CGH09] Jason Cong, Karthik Gururaj, and Guoling Han. Synthesis of reconfigurable high-performance multicore systems. In *Proc. FPGA*, 2009.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[CSL12] Kit Cheung, Simon R. Schultz, and Wayne Luk. A large-scale spiking neural network accelerator for FPGA systems. In *Artificial Neural Networks and Machine Learning - ICANN 2012 - 22nd International Conference on Artificial Neural Networks, Lausanne, Switzerland, September 11-14, 2012, Proceedings, Part I*, pages 113–120, 2012.

[CSZ$^+$14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Computing Frontiers Conference, CF'14, Cagliari, Italy - May 20 - 22, 2014*, page 3, 2014.

[CTJ$^+$12] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip Heng Wai Leong, and David B. Thomas. A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays, FPGA 2012, Monterey, California, USA, February 22-24, 2012*, pages 57–66, 2012.

[CvN50] R. Charney, J. G.and Fjortoft and J von Neumann. Numerical integration of the barotropic vorticity equation. *Tellus*, 2:237–254, 1950.

[dDP09] Florent de Dinechin and Bogdan Pasca. Large multipliers with fewer dsp blocks. In *Proc. FPL*, 2009.

[dDP11] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[DeH96] André DeHon. DPGA utilization and application. In *FPGA*, pages 115–121, 1996.

[DeH99] André DeHon. Balancing interconnect and computation in a reconfiguable computing array (or, why you don't really want 100% LUT utilization). In *FPGA*, pages 69–78, 1999.

[DM74] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture, December 1974*, pages 126–132, 1974.

[DML12] François Duhem, Fabrice Muller, and Philippe Lorenzini. Reconfiguration time overhead on field programmable gate arrays: reduction and cost model. *IET Computers & Digital Techniques*, 6(2):105–113, 2012.

[DMV$^+$08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE, 2008.

[DRM14] Richard Dorrance, Fengbo Ren, and Dejan Markovic. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas. In *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, pages 161–170, 2014.

[EAGEG09] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. In *ACM Trans. on TRETS*, volume 1, 2009.

[EBTB63] G. Estrin, B. Bussel, R. Turn, and J. Bibb. Parallel processing in a restructurable computer system. *IEEE Trans. Elect. Comput.*, pages 747–755, 1963.

[FBS13] Brahim Al Farisi, Karel Bruneel, and Dirk Stroobandt. Staticroute: A novel router for the dynamic partial reconfiguration of FPGAs. In *FPL*, pages 1–7, 2013.

[FC05] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *FCCM*, pages 149–158, 2005.

[FC08] Wenyin Fu and Katherine Compton. Scheduling intervals for reconfigurable computing. In *FCCM*, pages 87–96, 2008.

[FC11a] Hoahuan Fu and Robert G. Clapp. Eliminating the memory bottleneck: an FPGA-based solution for 3d reverse time migration. In *Proc. FPGA*, 2011.

[FC11b] Hoahuan Fu and Robert G. Clapp. Eliminating the memory bottleneck: an FPGA-based solution for 3D reverse time migration. In *Proc. FPGA*, 2011.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972.

[GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. *High - Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic, 1992.

[GP89] G. R. Gao and Z. Parskevas. Compiling for dastaflow software pipelining. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.

[HBB04] Michael Hübner, Tobias Becker, and Jürgen Becker. Real-time lut-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI*, pages 28–32, 2004.

[HbL29] T. S. Y. Ho and S. b. Lee. Term structure movements and pricing interest rate contingent claims. *Journal of Finance*, 41(5):1986, 1011–1029.

[Her97] Brian Von Herzen. Signal processing at 250 mhz using high-performance FPGA's. In *FPGA*, pages 62–68, 1997.

[HJM05] D. Heath, R. Jarrow, and A. Morton. Bond pricing and the term structure of interest rates. *Econometrica*, 60(1):1992, 77–105.

[HK12] Chin Hau Hoo and Akash Kumar. An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. In *FPL*, pages 400–406, 2012.

[HLH$^+$98] Rhett D. Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas. Spatio-temporal partitioning of computational structures onto configurable computing machines. In *Proc. SPIE*, pages 62–71, 1998.

[HLM$^+$13] Ruining He, Guoqiang Liang, Yuchun Ma, Yu Wang, and Jinian Bian. Unification of PR region floorplanning and fine-grained placement for dynamic partially reconfigurable fpgas. *Journal of Circuits, Systems, and Computers*, 22(4), 2013.

[HLP11] Markus Happe, Enno Lubbers, and Marco Platzner. A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking. *Journal of Real-Time Image Processing*, pages 1–16, 2011.

[HMZB12] Ruining He, Yuchun Ma, Kang Zhao, and Jinian Bian. Isba: An independent set-based algorithm for automated partial reconfiguration module generation. In *ICCAD*, pages 500–507, 2012.

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[Hul05] J.C. Hull. *Options, Futures and Other Derivatives.* Prentice Hall, 6th edition, 2005.

[IDC] IDC. Extracting value from chaos. `http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf`. [Online; accessed 2-April-2015].

[Int] Intel. Intel Xeon processor 5600 series. `http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf`. [Online; accessed 2-April-2015].

[JBLT12a] Qiwei Jin, Tobias Becker, Wayne Luk, and David B. Thomas. Optimising explicit finite difference option pricing for dynamic constant reconfiguration. In *FPL*, pages 165–172, 2012.

[JBLT12b] Qiwei Jin, Tobias Becker, Wayne Luk, and David B. Thomas. Optimising explicit finite difference option pricing for dynamic constant reconfiguration. In *Proc. FPL*, 2012.

[JDT$^+$12] Qiwei Jin, Diwei Dong, Anson H. T. Tse, Gary Chun Tak Chow, David B. Thomas, Wayne Luk, and Stephen Weston. Multi-level customisation framework for curve based monte carlo financial simulations. In *Reconfigurable Computing:*

Architectures, Tools and Applications - 8th International Symposium, ARC 2012, Hong Kong, China, March 19-23, 2012. Proceedings, pages 187–201, 2012.

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, volume 42, pages 235–244. ACM, 2007.

[KCO+10] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[KR07] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[KT11] Dirk Koch and Jim Torresen. Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *FPGA*, pages 45–54, 2011.

[KTB+12] Dirk Koch, Jim Torresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on fpgas in practice - tools and applications. In *ARCS 2012 Workshops, 28. Februar - 2. März 2012, München, Germany*, pages 297–319, 2012.

[KV98] Meenakshi Kaul and Ranga Vemuri. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Proc. DATE*, 1998.

[LM13] Ligang Lu and Karen Magerlein. Multi-level parallel computing of reverse time migration for seismic imaging on blue gene/q. In *PPOPP*, pages 291–292, 2013.

[LNTT01a] John W. Lockwood, Naji Naufel, Jonathan S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (fpx). In *FPGA*, pages 87–93, 2001.

[LNTT01b] John W. Lockwood, Naji Naufel, Jonathan S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (fpx). In *FPGA*, pages 87–93, 2001.

[M$^+$02] M Montemerlo et al. Conditional particle filters for simultaneous mobile robot localization and people-tracking. In *Proc. ICRA*, 2002.

[MD96] Ethan Mirsky and Andre DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, pages 157–166, 1996.

[Mic09] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU*, pages 79–84, 2009.

[NCJ$^+$13a] Xinyu Niu, Thomas C. P. Chau, Qiwei Jin, Wayne Luk, and Qiang Liu. Automating elimination of idle functions by run-time reconfiguration. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*, pages 97–104, 2013.

[NCJ$^+$13b] Xinyu Niu, Thomas C. P. Chau, Qiwei Jin, Wayne Luk, and Qiang Liu. Automating elimination of idle functions by run-time reconfiguration. In *Proc. FCCM*, 2013.

[NFMM13] Matthew Naylor, Paul J. Fox, A. Theodore Markettos, and Simon W. Moore. Managing the FPGA memory wall: Custom computing or vector processing? In *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*, pages 1–6, 2013.

[NJL$^+$12a] Xinyu Niu, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. Exploiting run-time reconfiguration in stencil computation. In *Proc. FPL*, 2012.

[NJL$^+$12b] Xinyu Niu, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. Exploiting run-time reconfiguration in stencil computation. In *FPL*, pages 173–180, 2012.

[NJL$^+$12c] Xinyu Niu, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. Exploiting run-time reconfiguration in stencil computation. In *FPL*, pages 173–180, 2012.

[NLW15] Xinyu Niu, Wayne Luk, and Yu Wang. EURECA: On-chip configuration generation for effective dynamic data access. In *Proceedings of the ACM/SIGDA 23rd International Symposium on Field Programmable Gate Arrays, FPGA 2015, Monterey, California, USA, February 22-24, 2015*, 2015.

[NSC$^+$10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE Computer Society, 2010.

[NTL11] Xinyu Niu, Kuen Hung Tsoi, and Wayne Luk. Reconfiguring distributed applications in FPGA accelerated cluster with wireless networking. In *International Conference on Field Programmable Logic and Applications, FPL 2011, September 5-7, Chania, Crete, Greece*, pages 545–550, 2011.

[Nvi] Nvidia. Tesla c2050 / c2070 gpu computing processor. `http://www.nvidia.co.uk/object/product_tesla_C2050_C2070_uk.html`. [Online; accessed 2-April-2015].

[P$^+$14] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale data-center services. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA*, pages 13–24, 2014.

[PAA$^+$06] Wesley Peck, Erik K. Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. Hthreads: A computational model for reconfigurable devices. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*, pages 1–4, 2006.

[PB99] Karthikeya Gajjala Purna and Dinesh Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Trans. on Computers*, 48:579–590, 1999.

[PBD$^+$13] Oliver Pell, Jacob Bower, Robert Dimond, Oskar Mencer, and M Flynn. Finite difference wave propagation modeling on special purpose dataflow machines. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):906–915, 2013.

[PF10a] Everett Phillips and Massimiliano Fatica. Implementing the himeno benchmark with CUDA on GPU clusters. In *Proc. IPDPS*, 2010.

[PF10b] Everett H. Phillips and Massimiliano Fatica. Implementing the himeno benchmark with CUDA on GPU clusters. In *IPDPS*, pages 1–10, 2010.

[PLL$^+$12] Michael Perrone, Lurng-Kuo Liu, Ligang Lu, Karen Magerlein, Changhoan Kim, Irina Fedulova, and Artyom Semenikhin. Reducing data movement costs: Scalable seismic imaging on blue gene. In *IPDPS*, pages 320–329, 2012.

[PWW97] Alex Peleg, Sam Wilkie, and Uri C. Weiser. Intel MMX for multimedia pcs. *Commun. ACM*, 40(1):24–38, 1997.

[Qui00] Daniel J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[Rei60] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:261–265, 1960.

[RMNM$^+$12] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch, Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. Forward and adjoint simulations of seismic wave propagation on emerging large-scale gpu architectures. In *SC*, page 38, 2012.

[S$^+$11] Kentaro Sano et al. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Proc. FCCM*, 2011.

[SB01] Alfred Strey and Martin Bange. Performance analysis of intel's MMX and SSE: A case study. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par*

Conference Manchester, UK August 28-31, 2001, Proceedings, pages 142–147, 2001.

[SB08] Hayden Kwok-Hay So and Robert W. Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using BORPH. *ACM Trans. Embedded Comput. Syst.*, 7(2), 2008.

[SFG06] Antonio Di Stefano, Giuseppe Fiscelli, and Costantino G. Giaconia. An FPGA-based software defined radio platform for the 2.4ghz ism band. In *Research in Microelectronics and Electronics*, pages 73–76, 2006.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[SWP04] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Computers*, 53(11):1393–1407, 2004.

[TCJW97] Steven Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In *FCCM*, pages 22–29, 1997.

[Tec] Maxeler Technologies. Maxcompiler. `http://www.maxeler.com/products/software/maxcompiler/`. [Online; accessed 13-March-2015].

[TL06] David B. Thomas and Wayne Luk. Non-uniform random number generation through piecewise linear approximations. *Proc. FPL*, 2006.

[TL08a] David B. Thomas and Wayne Luk. Credit risk modelling using hardware accelerated monte-carlo simulation. In *16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008, 14-15 April 2008, Stanford, Palo Alto, California, USA*, pages 229–238, 2008.

[TL08b] David B. Thomas and Wayne Luk. Fpga-optimised high-quality uniform random number generators. In *Proceedings of the ACM/SIGDA 16th International Sym-*

posium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008, pages 235–244, 2008.

[TW04] R. H. Turner and R. F. Woods. Highly efficient, limited range multipliers for lut-based fpga architectures. *IEEE Trans. VLSI Syst.*, 12(10):1113–1118, 2004.

[VADG02] J. Vermaak, C. Andrieu, A. Doucet, and S.J. Godsill. Particle methods for bayesian modeling and enhancement of speech signals. *Speech and Audio Processing, IEEE Transactions on*, 10(3):173 –185, mar 2002.

[vN93] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15:27–75, October 1993.

[W$^+$07] David Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[WB04] John W. Williams and Neil W. Bergmann. Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'04, June 21-24, 2004, Las Vegas, Nevada, USA*, pages 163–169, 2004.

[WKW02] Grant B. Wigley, David A. Kearney, and David Warren. Introducing reconfigme: An operating system for reconfigurable computing. In *Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream, 12th International Conference, FPL 2002, Montpellier, France, September 2-4, 2002, Proceedings*, pages 687–697, 2002.

[Xila] Xilinx. 7 series FPGAs configuration user guide. `http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf`. [Online; accessed 12-January-2015].

[Xilb] Xilinx. Logicore ip axi hwicap (v2.01.a). `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf`. [Online; accessed 12-January-2015].

[Xilc] Xilinx. Logicore ip axi hwicap (v2.01.a). `http://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_01_a/ds817_axi_hwicap.pdf`. [Online; accessed 12-January-2015].

[Xild] Xilinx. Planahead software tutorial. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/PlanAhead_Tutorial_Partial_Reconfiguration.pdf`. [Online; accessed 14-March-2015].

[Y$^+$03] Steve Young et al. A high I/O reconfigurable crossbar switch. In *FCCM*, pages 3–10, 2003.

[YPS11] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *CoRR*, abs/1103.2405, 2011.

[YSR08] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 61–70, 2008.

[ZP05] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, FPGA 2005, Monterey, California, USA, February 20-22, 2005*, pages 63–74, 2005.