Co-funded by the European Commission within the Seventh Framework Programme

Project no. 318521

HARNESS

Specific Targeted Research Project
HARDWARE- AND NETWORK-ENHANCED SOFTWARE SYSTEMS FOR CLOUD COMPUTING

Characterisation Report
D3.1

Due date: 30 September 2013
Submission date: 30 October 2013

Start date of project: 1 October 2012

Document type: Deliverable
Activity: RTD
Work package: WP3

Editor: Gabriel Figueiredo (IMP)

Contributing partners: SAP, IMP, MAX

Reviewers: Oliver Pell, Wayne Luk, Guillaume Pierre

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Description
0.1	2013/06/12	John McGlone	SAP	Initial outline
0.2	2013/07/01	Gabriel Figueiredo	IMP	Initial background material
0.3	2013/07/04	John McGlone	SAP	Delta merge overview
0.4	2013/07/08	Eoghan O'Neill	SAP	GPU and OpenCL background material
0.5	2013/07/10	Eoghan O'Neill	SAP	SHEPARD overview and results
0.6	2013/07/10	Peter Sanders	MAX	Background on reconfigurable hardware
0.7	2013/07/16	Gabriel Figueiredo	IMP	Updates to Chapter 3
0.8	2013/07/19	Peter Sanders	MAX	MaxelerOS prototype description
0.9	2013/07/19	Gabriel Figueiredo	IMP	Aspect-oriented DFE design-flow description
1.0	2013/07/26	Oliver Pell	MAX	Description on management of heterogeneous resources
1.1	2013/08/01	Eoghan O'Neill	SAP	Added delta merge results
1.2	2013/08/07	Peter Sanders	MAX	Updated MaxelerOS prototype description
1.3	2013/09/27	Gabriel Figueiredo	IMP	Updates based on Oliver, Guillaume and Wayne's reviews
1.4	2013/10/01	Gabriel Figueiredo	IMP	Consistency checks
1.5	2013/10/20	Alexander Wolf	IMP	Final review and edits by Coordinator

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.1	Define heterogeneous computation resources model	SAP*, IMP, MAX
T3.2	Design/develop computation management infrastructure	IMP*, MAX, SAP, ZIB

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

The aim of Deliverable D3.1 is to describe how to characterise computation resources to optimise the mapping of applications (Task T3.1). However, we consider that this characterisation should be done in the context of how applications are mapped and managed on compute resources (Task T3.2), as oppose to looking at these parts in isolation. Therefore, this deliverable reports, in the context of tasks T3.1 and T3.2, the general methods in which we characterise and manage compute resources, and how heterogeneous compute resources can be programmed to leverage their potential.

The work reported in this deliverable is the result of the efforts of three *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) partners: namely *Maxeler Techologies* (MAX), *SAP AG* (SAP), and *Imperial College London* (IMP). It includes the development of two novel and complementary approaches:

1. A **two-tier run-time computation management system** that allocates heterogeneous compute resources to adapt workload *horizontally* across multiple compute nodes, and *vertically* by exploiting multi-core heterogeneous resources available in a compute node.

2. A **unified heterogeneous programming approach** that decouples functional descriptions from non-functional concerns. In our approach, functional descriptions can capture multiple semantics to describe a computation using the same language, such as imperative and dataflow, allowing a more efficient mapping to heterogeneous compute resources such as *central processing unit*s (CPUs) and *dataflow engine*s (DFEs), which are compute resources exploiting *field-programmable gate array*s (FPGAs). Non-functional descriptions, on the other hand, allow user knowledge and expertise to be codified with *aspects*. Aspects can then be applied and even reused to automatically optimise functional descriptions.

As a result of this research, we have developed three infrastructures covering the above run-time and compile-time approaches, and have evaluated them using the HARNESS validation use cases, namely *reverse time migration* (RTM), Delta Merge and AdPredictor:

- **SHEPARD run-time management infrastructure.** *Scheduling for Heterogeneous Platforms using Application Resource Demands* (SHEPARD) targets OpenCL devices. SHEPARD has the ability to adapt allocation and to share out workload, thus removing fixed or static allocation of tasks in application code. When allocating tasks to resources based on the column size in the Delta Merge validation use case, our managed approach can adequately choose the device that yields the lowest execution time. Concurrent tasks are implicitly shared between multiple devices based on expected execution time and current device load.

- **MaxelerOS run-time management infrastructure.** One of the key features of this prototype is the use of the *groups* abstraction, which allows a cluster of dataflow engines to be managed as a single compute entity, offering a low-level elastic platform that can automatically adapt to workload.

- **Aspect-oriented dataflow design infrastructure.** This programming infrastructure supports an aspect-driven design flow for deriving optimised DFE designs. More specifically, we focus on codifying aspects that minimise development effort, exploit DFE architectural features, and explore variant implementations and run-time reconfiguration.

In addition, we have developed two prototypes targeting RTM and AdPredictor to investigate how these applications scale on dataflow engines and to help us derive performance models for CPUs and DFEs:

- **Dynamic stencil computation prototype.** This prototype implements a *dynamic stencil*, which is a special type of design that implements stencil computations and that scales automatically and dynamically on a platform with multiple DFEs connected to each other. Experimental results with RTM show that high throughput and significant resource utilisation can be achieved with dynamic stencil designs, which can dynamically scale into reconfigurable computing nodes as they become available during their execution. When statically optimised and initialised, the dynamic stencil design is 1.8 to 88 times faster and 1.7 to 92 times more power efficient than reference CPU, *general-purpose graphics processing unit* (GPGPU), MaxGenFD, Blue Gene/P, Blue Gene/Q and Cray XK6 designs; when dynamically scaled, resource utilisation of the design reaches 91%, which is 1.8 to 2.3 times higher than their static counterparts.

- **AdPredictor parallel training prototype.** The AdPredictor prototype allows the evaluation of multithreaded CPUs and DFE designs targeting the 2012 KDD Cup dataset format. More specifically, this tool can be used to derive performance models for both CPUs using an arbitrary number of threads and for the corresponding DFE design. Initial results show that the 100Mhz DFE can run 149 times and 9 times faster than a single-threaded and a 20-threaded CPU implementation (with each thread running at 2.6Ghz) when processing 10 million impressions.

This deliverable will be followed by Deliverable D3.2 at M24 in which we will extend the work above to develop effective methods of allocating hardware resources and characterisation jobs within the context of the HARNESS platform and all three validation use cases. This work will be driven by the industrial requirements and the validation plan defined in deliverables D2.2 and D2.3, respectively.

Contents

Acronyms

AOP *aspect-oriented programming*. 13

API *application programming interface*. 8, 10–12, 26, 53, 56, 87, 88

ASIC *application-specific integrated circuit*. 5

BRAM *block random-access memory*. 67, 70

ConPaaS *Contrail platform-as-a-service*. 33

CPU *central processing unit*. i, ii, 1–5, 8, 10, 19, 20, 22, 27, 29, 32, 33, 35, 47–49, 55, 58, 60, 63, 65, 69, 70, 72, 77–81, 83–85, 87, 89, 93, 94

CUDA *compute unified device architecture*. 7–9

DFE *dataflow engine*. i, ii, 1, 2, 5, 6, 20, 21, 24, 29, 31, 41, 44–48, 54–56, 60, 83–85, 87–90, 93, 94

DSC *domain-specific computation*. 26–28, 33–38, 48, 49, 56

DSP *digital signal processor*. 4, 37, 51, 52, 57, 58, 67, 70

FAST *Facile Aspect-driven Source Transformations*. 26, 28, 33–35, 37, 38, 48–52, 56, 58

FPGA *field-programmable gate array*. i, 3–5, 10–12, 18, 19, 22, 25, 30, 31, 33, 35, 38, 49–51, 58, 60–62, 65, 67–75, 83

GCC *GNU compiler collection*. 20, 34, 35, 37

GPGPU *general-purpose graphics processing unit*. ii, 1, 3–10, 18–22, 24, 25, 29, 30, 33, 38, 55, 63, 72, 73, 80, 87, 90, 94

HARNESS *Hardware- and Network-Enhanced Software Systems for Cloud Computing*. i, ii, 2, 19, 21, 25–27, 40, 44, 55, 56, 77–80, 85, 89, 90, 93, 94

HLSL *high-level shader language*. 7

IMP *Imperial College London*. i, 89

IOB *input/output block*. 4

JVM *Java virtual machine*. 31

LUT *look-up table*. 4, 51, 52, 57, 58, 67, 70

MAX *Maxeler Techologies*. i, 87

NUMA *non-uniform memory access*. 4

OS *operating system*. 9

PaaS *platform-as-a-service*. 19, 33

PCIe *peripheral component interconnect express*. 4, 70, 72

PDE *partial differential equation*. 62

PGAS *partitioned global address space*. 9, 10

RAM *random-access memory*. 4, 5

RTM *reverse time migration*. i, ii, 2, 55, 56, 58–60, 70, 72, 73, 93, 94

SAP *SAP AG*. i, 87

SHEPARD *Scheduling for Heterogeneous Platforms using Application Resource Demands*. i, 41, 42, 78, 93

SIMD *single instruction, multiple data*. 4, 7

SLiC *Simple Live CPU*. 6, 44

SLO *service-level objective*. 90

SMT *simultaneous multithreading*. 4

SPMD *single program, multiple data*. 10

SPU *synergistic processing unit*. 4

SWAR *SIMD within a register*. 4

UMA *uniform memory access*. 4

1 Introduction

In this deliverable we present the general methods by which we characterise and manage compute resources, and how heterogeneous compute resources can be programmed to leverage their potential. In particular, we are investigating (i) how applications can dynamically exploit heterogeneous compute resources at run time and (ii) how applications can be programmed to allow a more flexible and efficient mapping to heterogeneous platforms at compile time. In the context of this work, we are developing two novel and complementary approaches:

1. **Two-tier run-time computation management**. We are designing a run-time system that supports heterogeneous compute resources using two management processes: A top-level management process that dynamically allocates workload across distributed compute nodes, and a low-level management process that dynamically adapts the workload across a set of heterogeneous multi-core processors available within a compute node, such as *central processing unit*s (CPUs), *dataflow engine*s (DFEs) and *general-purpose graphics processing unit*s (GPGPUs). By supporting two management tiers, we are able to focus on specific concerns of each domain: the top-level management can focus on horizontal scale concerns such as faults, security, and jittering, while the low-level management can focus on vertical scale concerns, such as exploiting the specific features of heterogeneous physical resources. This organisation also allow us to experiment how jobs can be mapped efficiently on distributed heterogeneous compute nodes: jobs traditionally scaled horizontally can now be scaled vertically and vice versa, providing trade offs in terms of performance, energy consumption and resource utilisation. We discuss our run-time management approach in more detail in Section 3.3.1.

2. **Unified heterogeneous programming model**. We are also developing a new programming model in which developers can express alternate algorithms and semantics for existing software functions using the same language. This approach provides a non-invasive and non-fragmented description of an application, one that captures multiple variant descriptions for existing software code bases, allowing a more flexible and efficient mapping of software applications to heterogeneous compute resources. In our approach, the software and variant code bases can be part of the same application description. When using a compiler like *gcc*, only the software layer is compiled. However, when we employ our extended compilation flow, the connection between the software and variant layers is enacted through pragma annotations, and a hybrid application targeting multiple heterogeneous compute resources can be generated. In this first year, we have focused on the dataflow semantic to allow efficient mappings of applications to DFEs.

   This unified heterogeneous programming approach also incorporates an *aspect-oriented* design-flow, in which we decouple functional concerns (as described in the software and variant layers) from non-functional concerns, such as performance and energy consumption. With aspects, we capture strategies that convey user knowledge and expertise in a way that can be applied in an automated and systematic way to derive optimised functional descriptions. We discuss the unified heterogeneous programming model in more detail in Section 3.6.

The work above has been largely driven by the six general compute requirements defined in Deliverable D2.1 [31]. These requirements have been divided into three groups according to the target outcomes of WP3 (Section 3.2):

- the **run-time management infrastructure** has three general requirements: to support the characterisation of applications and resources (R11), to enable the virtualisation and sharing of heterogeneous compute resources (R12), and to optimise the management of resources to satisfy specific constraints (R14);
- the **programming infrastructure** must support the generation of multi-target design variants (R10); and
- the **platform integration** has three general requirements: the implementation of a discovery resource protocol to inform the cloud platform of available compute resources (R9), and resource monitoring to support the cloud platform's decision-making process (R13).

As a result of this research, we have developed two run-time infrastructures that manage OpenCL devices (Section 4.1) and dataflow engines (Section 4.2). In addition, we developed a programming infrastructure supporting an aspect-driven design-flow for DFE design (Section 4.3). Furthermore, we have implemented two reference prototypes targeting *reverse time migration* (RTM) and AdPredictor to investigate how these applications scale on DFEs, as well as to derive performance models for these applications on CPUs and DFEs. We have evaluated our work on the three *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) validation use cases, namely RTM (Section 5.2), Delta Merge (Section 5.3) and AdPredictor (Section 5.4).

The structure of this document is as follows. Chapter 2 presents background and related work. In Chapter 3 we provide an overview of our methods of characterisation, management and development related to our run-time and compile-time approaches. Chapter 4 provides details about the implementation of the three infrastructures covering the above run-time and compile-time approaches. Chapter 5 reports the evaluation of these infrastructures using the HARNESS validation use cases. Further, it presents the evaluation of the AdPredictor and RTM designs on DFEs, respectively. Finally, Chapter 6 presents our planned activities for Year 2.

2 Background

2.1 Heterogeneous Compute

The paradigm shift from single-core to multi-core heterogeneous architectures has offered the possibility of accelerating computationally intensive applications while bypassing technological issues related to the continual increase of compute density, such as heat dissipation and leakage currents. For three decades, advances in single-core technology meant that increasing the number of transistors in integrated circuits and raising clock rates resulted in performance scaling without much programming effort.

This time, however, exploiting multi-core heterogeneous devices requires considerable expertise and effort. So what are the reasons for this programming complexity? To start with, performance gains on multi-core heterogeneous processors must take into account multiple levels of parallelism (from node-level to instruction-level) and the core features of the architecture, such as specialised hardware accelerators (*field-programmable gate array*s (FPGAs) and GPGPUs). This translates into a number of concerns for developers: how to conveniently use the features of more powerful compute nodes including exploiting parallelism at different levels of the architecture, as well as to deal with data location having to consider how fast data can be moved in and out across different types of memories (near and fast vs large and remote).

One of the key challenges of our current compute landscape is how to efficiently map highly demanding applications from domains such as medical imaging, finance, oil and gas and data warehousing, to heterogeneous multi-core platforms. Currently, much of the development and optimisation process requires human effort to address different types of parallelism, the specific capabilities of specialised architectures, and the hierarchical memories associated with user-managed data transfers.

In this chapter we explore current trends in compute technology, their basic features and how applications are built and deployed in these systems.

2.1.1 Architectural features

Moore's Law [79] is still relevant today: increasing compute density allows larger numbers of components and/or more complex components in a single die. Part of the chip's area can be dedicated to developing more complex cores, but with faster clock speeds and faster transistors comes increased heat buildup. To overcome this limitation, a new solution was found: to replicate the number of cores.

Replicating cores can be done in different ways. For instance, a multi-core system can harbour large and fast general-purpose processors in which the core count is kept relatively low. This is the case of desktop and server CPU micro-architectures that currently support 2 to 16 cores, such as the Intel Haswell [51], the AMD Piledriver [4], Oracle SPARC T5 [70] and IBM POWER7 [49]. Other architectures combine a limited number of large and fast CPU cores with a collection of lightweight cores that are smaller and less powerful. In this model, all cores are general-purpose and share a common instruction subset, but already exhibit a degree of heterogeneity, where the more complex cores (acting as the host) are used to perform sequential, I/O and memory bound computations more efficiently,

while lightweight cores (acting as co-processors) are employed to exploit parallel computations. This architecture is supported by the current generation Xeon Phi™ co-processors [52], which include 30 to 60 Pentium cores, and can be attached to the CPU host via a *peripheral component interconnect express* (PCIe) bus. Another example of this type of architecture is ARM's big.LITTLE,™ [7] which combines a high-performance ARM CPU with an energy efficient ARM CPU. With this architecture, workload is dynamically transitioned to the appropriate CPU based on performance needs.

We can also combine general-purpose processors with specialised cores (Section 2.1.2), such as FPGAs, GPGPUs and Cell's *synergistic processing unit*s (SPUs) [50] that, working as co-processors, can offer performance orders of magnitude faster than general-purpose processors while being more energy efficient. However, specialised resources lack the flexibility of general-purpose processors, and thus, such gains can only be achieved if workload can be conveniently adapted to these architectures.

Memory architectures also differ in the context of multi-core systems. The simplest memory organisation is a shared-memory architecture in which multiple cores can access the same memory device to allow communication and avoid redundant copies. Shared-memory architectures can be *uniform memory access* (UMA) in which access time to a memory location is not dependant on which processor makes the request. Alternatively, in *non-uniform memory access* (NUMA) designs, a processor can access its own local memory cache or *random-access memory* (RAM) faster than memory that is local to other processors, while exposing a single logical address space. In contrast to shared-memory architectures, disjoint-memory systems have cores accessing different memories over a shared bus. Most heterogeneous systems that combine general-purpose and specialised processors support disjoint-memory systems, while integrated CPU and GPGPUs systems are moving towards a unified memory design [3].

Current multi-core systems support parallelism across different levels of the architecture. On a general-purpose processor, the basic levels of parallelism are:

1. instruction-level parallelism, item *SIMD within a register* (SWAR) [50, 52],
2. hardware-supported multithreading (e.g., hyper-threading), and
3. core-level parallelism.

With these technologies, modern day CPU cores speed up sequential programs by reordering independent operations, predicting branch execution, supporting vector instructions and supporting *simultaneous multithreading* (SMT).

GPGPUs are much simpler in terms of execution logic, with much of their real estate supporting a large number of cores to support *single instruction, multiple data* (SIMD) computations. More specifically, GPGPUs have very wide SIMD processors, with N cores that are capable of operating on N floats in parallel.

FPGAs, on the other hand, contain an array of logic components called *logic blocks* and a hierarchy of interconnects that allow these blocks to be connected. Both logic blocks and interconnects can be configured to realise a design, with logic blocks implementing arbitrary combinational and sequential logic functions. In most FPGAs [86], logic blocks include memory elements, such as flip flops acting as registers, and *look-up table*s (LUTs) to implement Boolean functions. FPGA packages may include complete blocks of memory, dedicated functional resources such as *digital signal processor*s (DSPs), high-speed *input/output block*s (IOBs) and even embedded processors. Tens of thousands of logic blocks, currently available in modern FPGAs, can be executed at the same time, allowing fine-grained parallel

computations to be realised with techniques such as pipelining and functional replication. On top of general-purpose processors and specialised co-processors (GPGPUs and FPGAs), there is socket-level parallelism, in which an application can leverage multiple CPUs, FPGAs and GPGPUs connected to each other in a single machine. Finally, we have node-level parallelism, where hundreds of nodes can be connected using interconnects such as InfiniBand or Ethernet to form a computer cluster.

In the next section, we focus on two important types of specialised accelerators: DFEs and GPGPUs.

2.1.2 Accelerators

Dataflow engines

The spectrum of computation options has at one end a solution provided by *application-specific integrated circuit*s (ASICs) and, at the other end, a software solution running on general purpose CPUs. Reconfigurable hardware provides the benefits of performance and power consumption usually associated with a hardware system while at the same time providing the flexibility to change the logic to accomplish different tasks. FPGAs are an example of reconfigurable hardware. They are suitable for computation problems due to their large number of resources, variety of logic blocks, reliability and performance.

An FPGA alone is not enough to provide a self-contained and reusable computation resource. It requires logic to connect the device to the host, RAM for bulk storage, interfaces to other busses and interconnects, and circuitry to service the device. This complete system is called a DFE [72]. The DFE contains an FPGA as the computation fabric and can be easily integrated with a host system or shared with more than one host system.

A DFE operates on streams of data. Data are streamed from memory into the computation fabric where the operations are performed. The operations are performed on the data stream as it flows through the computation fabric, the data being forwarded spatially from one functional unit to another. Results of the computation are streamed back to the memory. This dataflow model is depicted in Figure 2.1. Other memories can also be a source or sink for the data stream, as can a DFE interconnect.

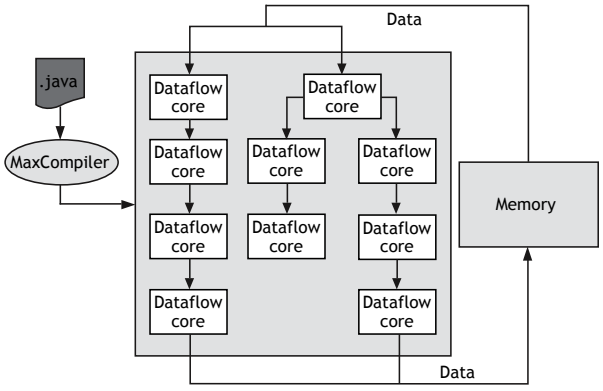


Figure 2.1: Example of a dataflow architecture with data flowing from memory through the computation fabric.

The implementation of an algorithm for a system utilising DFEs requires the identification of the computational bottlenecks that can benefit from the performance improvement delivered by DFEs. The relevant parts can then be described using a dataflow model [27], the new dataflow implementation converted into hardware data paths, or DFE *kernels*.

In the dataflow model, the programmer thinks of the implementation as a graph with data flowing through operators. Operations on independent streams within the implementation can be performed in parallel, resulting in a reduction in the compute latency. All the dependencies are resolved statically at compile time, so the resulting design can be highly parallelised and deeply pipelined. If there are resources available on the DFE, then entire implementations can be duplicated, providing multiple parallel implementations (pipes), that can improve the performance of the implementation on the DFE.

Together with the DFE dataflow configuration, the programmer must provide run-time software for the host system that will reconfigure the hardware and initiate the transfer of data to the DFE.

On a Maxeler system, the dataflow configuration is written using Java with Maxeler language extensions, called MaxJ, shown in Figure 2.2. The result of compiling a MaxJ program with MaxCompiler is a configuration file (the `.max` file) that can be compiled into the host run-time software. The programmer then uses the Maxeler *Simple Live CPU* (SLiC) interface to control when DFEs are reconfigured and used.

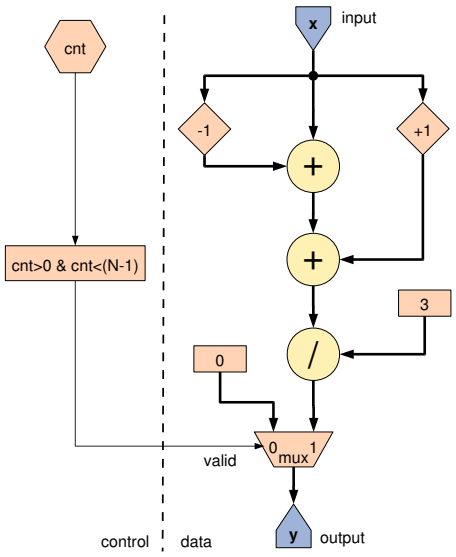
```
DFEType type = dfeFloat(8,24);
DFEVar x = io.input("x", type);

DFEVar x_prev = stream.offset(x, -1);
DFEVar x_next = stream.offset(x, +1);

DFEVar cnt = control.count.simpleCounter(32, N);
DFEVar valid = (cnt > 0) & (cnt < (N-1));

DFEVar y = valid ? (x_prev + x + x_next) / 3.0 : 0.0;
io.output("y", y, type);
```

(a) MaxJ Source



(b) Dataflow graph

Figure 2.2: Example of a simple DFE kernel showing the MaxJ source code and the dataflow graph.

GPGPU architectures

GPGPU processors have become a popular tool for accelerating computationally intense functions in many programming areas. Traditionally GPGPU processors are concerned solely with graphics rendering

operations, whereby thousands of floating point operations are performed in parallel. This has lead to an architecture best suited to problems that can be expressed as SIMD. GPGPUs exhibit a very large number of small cores, allowing them to efficiently run thousands of threads to operate over a single dataset.

GPGPUs can gain considerable speed-up on appropriate codes, but do also come with their own set of challenges. GPGPUs are broken down into a number of streaming multi-processors that contain a number of small processing cores and shared resources. In order to execute work on a GPGPU, threads are run in groups. Each group runs independently within a streaming multi-processor such that synchronisation between groups is not possible while they are executing. Since the groups run independently, the GPGPU can schedule them in any order and thus code executing on a GPGPU cannot make any assumptions about the scheduling of groups. This does, however, allow the GPGPU to hide latency by scheduling a group to execute while another is stalled or waiting on memory access. GPGPUs also come with a hierarchy of memories and it is vital that accesses to these various levels of memory are performed appropriately.

Off-chip memory exists as the largest volume of memory on a GPGPU accelerator. This memory is usually GDDR5 and in the order of four to six gigabytes for high-end GPGPU cards. Although faster than typical host memory, it is important that accesses to this memory comes in the form of coalesced calls in order to allow the GPGPU to efficiently service the thousands of running threads. Calls to off-chip memory introduces latency into execution and thus the GPGPU aims to schedule groups of threads to the GPGPU cores so that this latency is hidden. Thus, it is important that there are a high number of threads, usually hundreds to thousands, to allow the GPGPU to effectively hide the latency. Shared memory is on-chip memory and is limited in size, typically between 16 to 48 KB on NVIDIA cards. This memory is shared between cores within a streaming multi-processor and can be used for efficient communication between threads within a group. As this memory is shared within a streaming multi-processor, there exists a trade off between the amount of resources a thread uses and the number of simultaneous groups that can be scheduled within a streaming multi-processor. Recent revisions of GPGPUs have also introduced managed caches to ease the requirement to fine tune all codes to manually optimise access to the various GPGPU memories.

What has been detailed in this section so far represents some of the main architectural factors with which a developer must be concerned when targeting code to the GPGPU. These optimisations may require significant development effort. Furthermore, manual optimisations made for today's GPGPUs may not run efficiently on the GPGPU architectures of tomorrow. In fact, NVIDIA publishes new tuning guides for each new architecture to help developers adapt their codes. In general, GPGPUs are treated as accelerators to offload certain tasks within an application. Because there can be significant effort involved in creating performant code on the GPGPU, as detailed previously, it is important that developers can adequately identify hot spots within existing codes that can be effectively parallelised to make best use of the GPGPU. This is an extra overhead for a typical application developer and often requires a complete re-engineering of the code to fit the GPGPU.

Initial research into programming GPGPUs involved using a *high-level shader language* (HLSL). HLSL is a low-level, domain-specific language for writing graphics shaders and therefore not well suited to general-purpose applications. Therefore, the level of effort required to create programs for GPGPUs was initially very high.

The first breakthrough technology came in the form of *compute unified device architecture* (CUDA) from NVIDIA. CUDA, a C-based language, allows developers to offload sections of their code to the GPGPU. Developers create special functions, called *kernels*, that are run on the GPGPU. CUDA requires

the developer to program at a relatively low level, often fine tuning code to match the characteristics of the target GPGPU model. Successive revisions of GPGPU architectures and the CUDA language have eased this somewhat by introducing managed caches to reduce the penalties of random memory access. CUDA still requires most of the manual tuning described previously, allowing developers to finely optimise their code where required. Another limitation of CUDA is that it only supports NVIDIA GPGPUs.

OpenCL [44], on the other hand, is a cross-platform programming framework supported by many hardware vendors such as AMD and NVIDIA. It includes a language (based on C99) for writing kernels that execute on OpenCL devices and an *application programming interface* (API) that defines and manages target platforms. OpenCL's model supports parallel computing using task-based and data-based parallelism. The run-time environment works through a queue-based mechanism, in which applications queue work items to be executed on a specific device. A queue is required for each device used by an application, and the developer can specify dependencies between tasks. Compute devices from the same vendor can be grouped to share memory. OpenCL requires the developer to explicitly choose which devices to use and when, which is typically decided at design time.

As GPGPUs have become popular, other technologies have emerged to attempt ease the overheads of creating programs to run on them. Outlined here are some of the popular technologies and research projects that aim to ease the overheads of utilising GPGPUs as accelerators.

Libraries. Libraries are among the simplest ways to interact with the GPGPU, providing the required tasks can be achieved using the functions the library provides.

- Thrust [47] is a library for GPGPUs that mimics the standard template library for C++. Developers call library functions that implement various functions such as sort, reduce and scans. The library completely hides the implementation from the developer allowing the code to remain at a higher level.

- MAGMA [87], developed at the University of Tennessee Innovative Computing Lab, is a highly tuned dense linear algebra library, similar in function to LAPACK. Libraries such as this allow programmers to access highly optimised implementations of complex math functions without any implementation requirement.

Compilers. A number of projects exist that attempt to use compilers to automatically generate GPGPU implementations directly from source code.

- GPGPU Ocelot [28], from the Georgia Institute of technology, aims to automatically generate NVIDIA PTX code that will allow programs to run on NVIDIA GPGPUs. This project aims to remove entirely the overhead of recreating code to target GPGPUs, allowing programmers to reuse existing code bases.

- Other technologies have attempted to abstract implementations of GPGPU code by allowing the application programmer to use pragmas to annotate the sections of code they wish to have accelerated on the GPGPU, similar to OpenMP for CPUs. The primary example in this area is OpenACC [93]. With this approach, developers generally target loops that can be accelerated on the GPGPU. The compiler is then tasked with generating the necessary code to enable execution on the GPGPU.

Run-time frameworks. To enable managed use of GPGPU technologies, a number of frameworks have been created in the research community to tackle the problems of creating programs to utilise GPGPUs and offload tasks appropriately.

- Anthill [30] is a framework that allows developers to construct programs as tasks and memory streams between tasks. The framework then attempts to overlap memory transfers with execution to improve overall throughput.
- StarPU [9] is a framework that uses a pre-fetching mechanism to pre-load data for kernel execution in a multi-GPGPU environment.
- Qilin [59] dynamically recompiles applications to create tuned execution on a heterogeneous platform that supports Intel Threading Building Blocks (TBB) and NVIDIA CUDA. Each application is tuned in isolation without taking into account other co-resident applications.

Operating systems. Other projects aim to enable the use of accelerators such as GPGPUs at the *operating system* (OS) level.

- Barrelfish [11] is a project to build an OS from the ground up designed to run on all available devices. The OS works by having each device run its own driver kernel, capable of scheduling threads. Communication is then achieved via message passing between devices to maintain system state.
- Mosix [10] is a virtual cluster platform that allows OpenCL applications to access all OpenCL-compliant devices within a cluster as if they were local.

2.1.3 Programming heterogeneous designs

Modern multi-core systems differ from each other on the number of cores, core complexity, level of heterogeneity, communication topology, memory hierarchy and processing features. All these elements influence how programs are developed and deployed in multi-core systems. Concerns such as hardware/software partitioning (selecting parts of the code that are better suited for specialised resources), locality (maximising data reuse), layout (keeping data close together where it is used), copying (minimising data movement), processing features (exploiting underlying parallelism, using specialised resources and instructions) need to be taken into account by programmers to fully exploit the underlying architecture.

Conventional parallel programming models for multi-core systems can be split into three main approaches: shared memory, message passing and *partitioned global address space* (PGAS).

In the shared-memory model, a program is a collection of control threads. Each thread has its own set of private variables and a set of shared variables. Threads communicate implicitly by writing and reading shared variables, and synchronising with each other. Examples of shared-memory libraries/systems based on threads include: OpenMP [17], Intel Cilk Plus [37] and Phoenix++ [83].

The message-passing model is based on a set of independent tasks that use their own local memory during computation, and that can reside in the same physical machine or across different machines. Tasks exchange data through communication primitives that send and receive messages. In this model, where

MPI [43] is the industry standard for message passing, programmers are responsible for determining task parallelism and for coordinating the interaction between tasks.

In the case of PGAS, the model is a global memory address space that is logically partitioned in a distributed system, in a way that parts of this memory space may have an affinity for a particular process. Approaches that support the PGAS programming model, such as Cray's Chapel [16] and IBM's X10 [18], attempt to combine the benefits of *single program, multiple data* (SPMD) programming in distributed systems with the memory referencing semantics of shared-memory systems. All these approaches offer an abstract view of the architecture, since they do not provide fine-grained control over mapping onto heterogeneous resources.

There are three common programming approaches that address heterogeneity in multi-core systems. The first approach offers a **uniform programming framework**, with compile-time and run-time support, that allows developers to describe the application only once using a single language to target multiple computational resources, such as CPUs, GPGPUs and FPGAs. An example of this type of framework is IBM's Liquid Metal [48]. Liquid Metal provides an object-oriented programming language (Lime [8]) and a run-time environment for programming heterogeneous multi-core processors. The binaries targeting different processors are derived using the same tool chain (from the point of view of the developer). Liquid Metal is architecture-agnostic, thus exhibiting higher levels of design productivity and maintainability, as well as providing greater flexibility for mapping parts of the computation to different processing elements dynamically and adaptively. However, while a single model of computation provides portability of implementations, it usually lacks the expressiveness to exploit specialised resources.

The second approach for programming multi-core heterogeneous systems is to use **hardware/software partitioning** techniques, in which programmers [45, 72] or compiler tools [60] select parts of the application (kernels) to be accelerated on specialised resources, such as FPGAs and GPGPUs. In this context, code to be offloaded is usually written using different models of computation that are better suited for the target processing element. This means that selecting kernels involves some degree of user expertise and profiling tools to select code that can exploit accelerator capabilities, and when considering disjoint memory systems, programmers also need to consider the overhead of transferring data in and out. In particular, when considering offloading computations to FPGAs, programmers need to take into account that (i) data in FPGAs are processed by spatially distributed computations rather than temporally sequencing and (ii) the functionality of the computational units and interconnects can be adapted at run time. This means that loops with a large iteration space that can be described in a dataflow form are natural candidates for offloading, as they allow the synthesis of efficient pipelined architectures. This approach provides the means to conveniently exploit different architectures, but requires a larger learning curve and effort, and exhibits a fragmented view of the application making it less flexible to move computations from different heterogeneous devices.

As a complement to the above approaches, projects such as MERGE [57], Intel's QuickAssist Technology Accelerator [23], and the Elastic Computing Framework [91] provide programmers a **uniform acceleration abstraction** allowing accelerator-specific implementations to be automatically selected at run time by invoking domain-specific and user-provided APIs. While accelerator-specific implementations can be expressed in different languages and models of computation, developers use a single language to describe the functionality of an application. The MERGE framework uses the map/reduce pattern to share work across processors. It requires user-supplied implementations, with predicate annotations indicating suitability and optimality criteria, for devices that exist on the target platform. QuickAssist

Technology Accelerator provides a standard interface for Intel to deploy domain-specific APIs that leverage supported heterogeneous resources. The Elastic Computing Framework supports user-provided information and a performance model associated to each implementation to make efficient decisions at run time. The uniform acceleration abstraction provides developers both implementation portability and efficient resource utilisation, although implementations must be adapted to work on a specific middleware framework.

2.1.4 Design-space exploration

Specialised resources, such as FPGAs, allow a multitude of architectural solutions for the same functional specification. The process of generating one hardware design is time consuming (possibly taking several hours to complete), and while it is possible to estimate performance (along with other design metrics) early in the design process, the effects of optimising a hardware description can only be fully understood once designs have been fully synthesised.

Thus, design-space exploration is often a part of a compilation design-flow targeting FPGAs, involving design parameterisation (e.g. number of replicated data paths) and compiler options (e.g. scheduling effort level). In general, design space exploration is the process of deriving and/or analysing implementations that are functionally equivalent in order to identify feasible and optimal solutions [15, 81], thus potentially benefiting all compute platforms. In the context of heterogeneous systems, for instance, design-space exploration can help formulate an execution plan to identify the most suitable heterogeneous implementations for different combinations of resources, input parameters and user-defined metrics [91].

There are three problems to take into account when performing design space exploration [84]. First, we must select the **exploration strategy** to traverse a potentially vast search space and find feasible solutions. Optimisation search techniques can range from randomised search [19], to iterative heuristic-based techniques such as tabu search [56], and exact techniques such as using integer linear programming formulation [1]. The choice of exploration strategy is dependent on a number of factors, including: the optimality requirements, the time and effort to find feasible solutions, the size of the search search and the proximity of neighbour solutions.

The second problem relates to evaluating non-functional properties in a **multi-objective design exploration**, for instance performance and energy consumption. In this case, there may not exist a single solution that simultaneously optimises all of the objectives, and thus we may find non-dominated, Pareto optimal, solutions in which none of the objective functions values can be improved without minimising the quality of some other objective value. The set of Pareto optimal solutions, or *Pareto front*, allows developers to choose designs that exhibit a clear trade off between different objective metrics.

The third problem in design-space exploration relates to **evaluating the quality of a design point**. This evaluation can be: (i) faster using an estimator to evaluate the quality of a possible solution; (ii) slower but more accurate by invoking the lower stages of the tool chain to arrive closer or to the final solution; or (iii) a combination of both evaluation approaches by only invoking the more accurate design flow at specific points of the design exploration process [85].

Design-space exploration can also be performed at run time using *auto-tuning* systems, in which empirical techniques are used to evaluate a set of alternative mappings that best suit a particular architecture. Auto-tuning systems can be grouped into three categories: (i) self-tuning libraries in specific domains such as ATLAS [92], OSKI [90] and FTTW [38]; (ii) compiler-based auto-tuners that target specific class of kernels, such as signal processing [74] and stencil computations [53]; and

(c) application-level auto-tuners, which automate search across a set of algorithms, algorithmic parameters, and transformations proposed by programmers, such as PetaBricks [5] and ADAPT [89].

When dealing with large design spaces, as in the case of multi-core systems with heterogeneous devices, design-space exploration requires great flexibility in terms of tools and programming models to allow seamless integration of user- and system-specific evaluators and a highly efficient exploration system (see Section 2.2.1).

2.1.5 Scaling computations

To optimise and parallelise an application, programmers are usually aware of the target architecture, including the number of autonomous processing elements. However, what if the number of compute nodes are unknown at compile time? This is a typical scenario when an application is deployed to a cloud platform. In this case, it is desirable that the application's workload is dynamically adapted to provisioned resources.

One way for an application to dynamically scale at run time is to use **algorithmic skeletons** [21]. Algorithmic skeletons are a higher-level functional-based programming model that employ commonly used patterns that abstract the complexity of parallel computation, communication and interaction. With this programming model, managing and synchronising parallel computations are implicitly defined by skeleton patterns. Because coordination, synchronisation and interaction are not expressed by programmers, a run-time scheduler has the flexibility to scale the application by mapping user-defined computations to a given number of nodes at run time. There are a number of well-known skeleton patterns, which include FARM (master/slave), PIPE (pipeline) and D&C (Divide and Conquer), and have been supported by a number of frameworks [39, 55]. Perhaps the most well-known pattern is D&C, which has been applied in Google MapReduce [26] and also in the context of a reconfigurable cluster [88].

An alternate way to codify scalable computations is to describe a kernel in a high-level description language based on a code pattern or template, and use **customised compilation tools** to derive a scalable design based on optimisation models. Examples of this approach are discussed by Niu et al. [66], in which a C description of a stencil computation is automatically translated to a scalable design that can dynamically expand or contract to a given number of FPGAs. Additionally, scalability can be achieved with a uniform acceleration abstraction (described in Section 2.1.3), where API calls are handled by a scheduler that decides which implementations to use to adapt the workload across available compute resources.

2.2 Aspect-Oriented Programming

Consider the example of a programmer who has worked extensively on a specific application domain and a target platform. Over the years, they will have acquired enough expertise to build a portfolio of strategies that allows them to improve desired requirements, such as performance, resource usage and energy efficiency. Such strategies could involve applying complex schemes, such as hardware/software partitioning, code specialisation, source code transformations and even insertion of monitoring modules to expose optimisation opportunities at compile time or at run time. However, in traditional development, this user knowledge is often ingrained within the application code, which captures its main functionality. As a result, this knowledge, as well other non-functional concerns, cannot be reused and applied systematically

and automatically to either the same application in different contexts or across applications. Furthermore, the resulting code becomes polluted, difficult to maintain and non-portable, as different domains may require different strategies. Design exploration is also difficult to achieve because there is no obvious mechanism to parameterise these strategies.

In this section we present LARA [12], a novel aspect-oriented language developed in the context of the FP7 REFLECT project [76] that addresses the problems mentioned above. The main purpose of LARA is to aid the process of mapping computations to computational systems, with emphasis in (but not limited to) heterogeneous systems with reconfigurable hardware. A key element of LARA, and a distinguishing feature from existing approaches, is its ability to support the specification of non-functional requirements and user knowledge in a non-invasive way in the exploration of alternative transformations and in the mapping of these alternative implementation either to software-only or a combined hardware/software co-design solutions.

LARA has been inspired by languages for *aspect-oriented programming* (AOP), such as AspectJ [54] and AspectC++ [82]. However, there are some profound differences with these aspect-oriented approaches. First, the weaving process is traditionally performed at run time, targeting a limited number of execution points (e.g. function calls) to introduce secondary functionalities. With LARA, the weaving process is performed at compile time and on top of source code. This allows the code to be analysed and manipulated more aggressively as a result of an aspect description to support non-functional concerns. Furthermore, LARA has been designed so that it is not tied to a specific language, but instead the weaver can be adapted to support various programming languages. LARA includes both declarative and imperative semantics. The semantics of pointcut expressions and associated advices are fully declarative, but the semantics of function definitions and weaving statements are described in an imperative style.

2.2.1 Weaving process

One of the key innovations of the LARA aspect-oriented approach is that it decouples functional concerns focusing on the algorithmic features of the design from non-functional concerns. Examples of non-functional concerns include desired qualities of a design, such as increasing performance, improving resource efficiency and minimising energy consumption. Through a process called **weaving**, the LARA-guided design flow combines, in an automated fashion, non-functional and functional concerns leading to the desired implementation. Figure 2.3a depicts the weaving of an application code, described in a high-level programming language such as C or MATLAB, with a LARA specification to derive a transformed (woven) application, which combines the features of both functional and non-functional descriptions.

There are several benefits to the weaving process as pursued in the LARA-guided design flow. First, it allows each concern, functional and non-functional, to be independently specified (and thus maintained) from the original application source code. This decoupling promotes a clean separation between an algorithmic description and the non-functional concerns leading to a cleaner and thus easier to maintain source code basis. This approach is in stark contrast with current design practises where by the abusive use of annotations such as `#define` and conditional compilation directives, code sources become harder to read over time, and thus more difficult to maintain and test. Second, and equally important, LARA aspects enable the codification of strategies that describe systematic transformational steps to achieve different non-functional requirements thus leading to potentially remarkably distinct design solutions (as depicted in Figure 2.3b). Aspects can thus be introduced, updated, and removed from the design flow

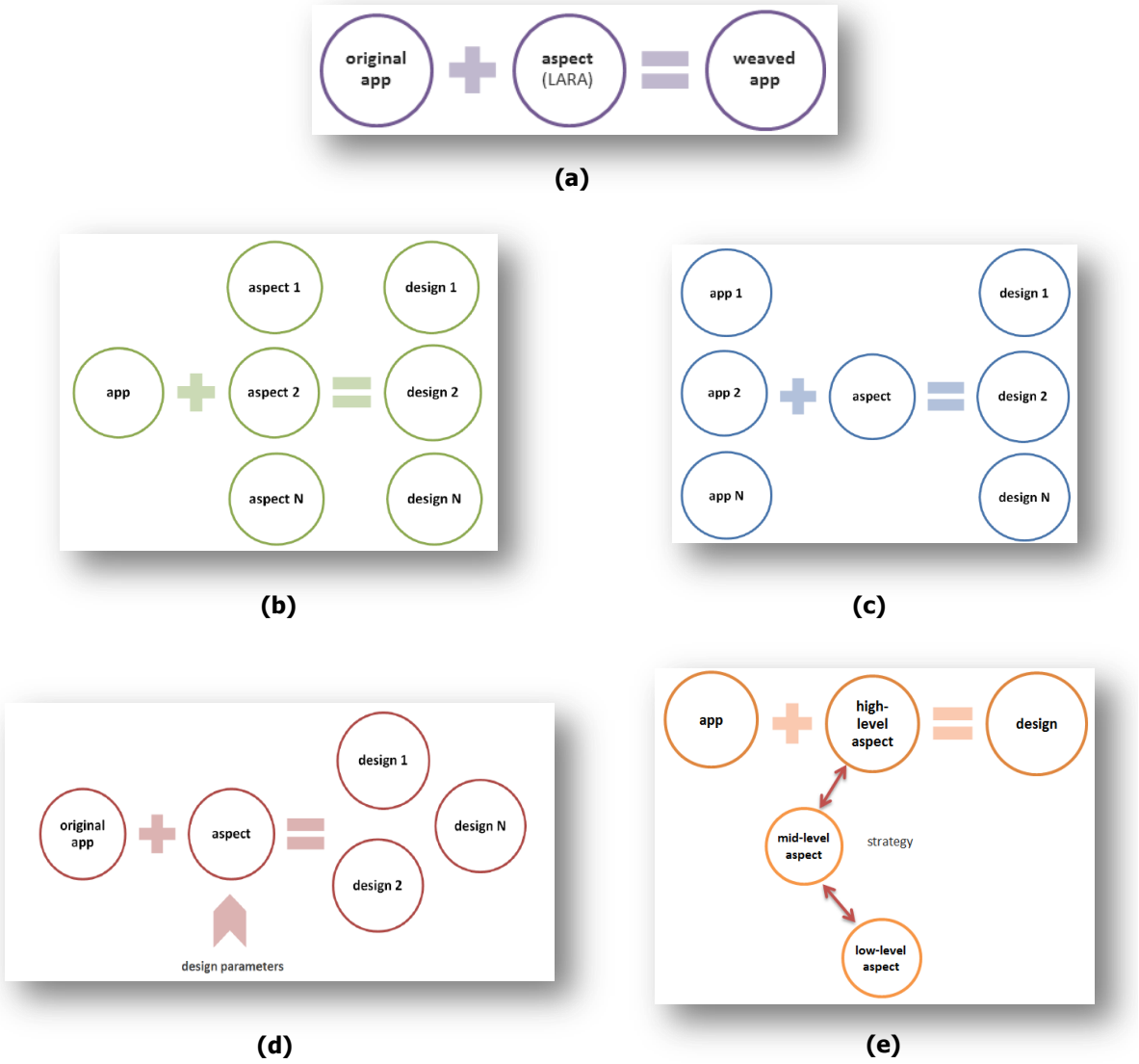


Figure 2.3: The benefits of LARA aspect-oriented design: (a) decoupling functional from non-functional descriptions, (b) customising an application, (c) reusing a strategy, (d) aspect parameterisation for design-space exploration, and (e) aspect modularisation for capturing complex strategies composed of target dependent (low-level) and independent (high-level) implementations.

based on user and system requirements without directly affecting the original source code. This feature of the design flow substantially improves overall **design portability** and **code maintainability**.

In addition to the benefits of a single-code/multi-design design flow, aspects can be developed in an application-independent way and, therefore, reused in the context of multiple application codes (as suggested by the illustration in Figure 2.3c). This reuse of aspects allows developers to capture common transformations and design patterns geared towards specific target architectures, thus promoting **design productivity** to accelerate the process of developing designs that exhibit specific non-functional concerns, such as performance.

The ability to specify generic and parameterisable aspects in LARA is particularly useful for describing hardware- and software-based design and transformation patterns as well as templates, thus facilitating design-space exploration (as depicted in Figure 2.3d). Examples of aspect parameters include application- and domain-specific information, such as function names and iteration space sizes. A key mechanism in LARA is the support for modular composition, where a non-functional concern can be satisfied through multiple aspect definitions. This allows developers to compose a strategy using different levels of abstraction (as conceptually represented in Figure 2.3e). Typically, higher-level aspects capture top-level non-functional requirements that are propagated through mid-level aspects and lower-level aspects, the latter dealing with increasingly more domain-specific tasks. For instance, a high-level aspect can capture a performance goal that is achievable through a combination of compilation sequences and design patterns (mid-level aspects), which in turn can trigger a set of code transformations specified in low-level aspects.

2.2.2 LARA language

The LARA language is based on the ECMAScript scripting language (ECMA-262 specification and ISO/IEC 16262) [29], and offers extensions to support aspect-oriented mechanisms. The main computational unit in LARA is an aspect definition. An aspect definition is composed of a number of optional sections (see Figure 2.4):

1. **Interface section:** This section defines the input and output parameters of an aspect which are key elements in aspect composition.

2. **Initialise and finalise sections:** These two sections are automatically invoked at the beginning and at the end of an aspect's life cycle, even when an exception is thrown. The initialise and finalise sections are equivalent, respectively, to the concepts of constructor and destructor in object-oriented languages.

3. **Check section:** This section specifies a condition that when evaluated to false disables the execution of an aspect. The check section is executed after the initialise section is completed.

4. **Static section:** This section declares variables and functions associated with the aspect definition across multiple instances of the same aspect. These definitions are thus not associated with any particular instance of an aspect, but rather are associated directly with the aspect definition, and can be accessed at any time during the life cycle of the weaver. This feature is useful when encapsulating utility methods and variables that do not require object instantiation, and is equivalent to static declarations in C++ languages.

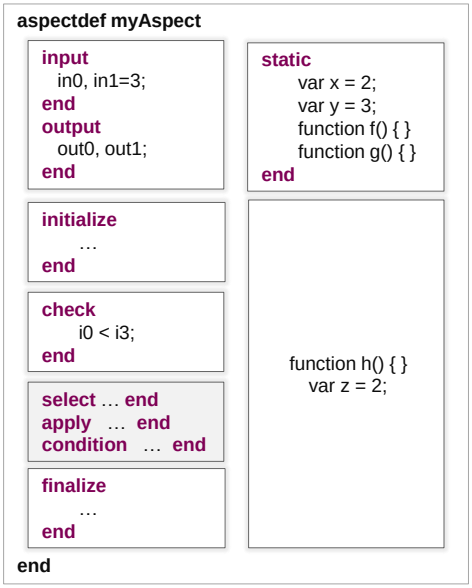


Figure 2.4: Sections of a LARA-based aspect definition. The highlighted section corresponds to aspect-oriented mechanisms. JavaScript code (functions, statements) can be introduced in any part of the aspect definition.

5. **Weaving statements:** There are three weaving statements, namely: select, apply and condition. The select statement operates on a set of syntactic objects of the application code called join points. The condition statement filters or selects a subset of these join points. The apply statement performs actions on the filtered set of join points.

6. **Code statements:** These statements consists of executable JavaScript statements including variable and functions declarations. These statements can occur in an aspect definition and inside the initialise, finalise and apply sections.

LARA weavers execute one aspect at a time starting from the **main** aspect definition that, by default, is the first aspect definition found in the LARA source file. The execution of an aspect can, in turn, trigger the execution of multiple aspects through explicit aspect invocation calls. The sequence of execution of an aspect definition starts with: loading the input parameters, executing the initialise section and evaluating the check section. If the check section yields true, the remaining statements of the aspect are invoked in the textual order in which they appear. The finalise section is executed before the aspect execution terminates. The static sections of all aspects are evaluated at the start of the weaving process by the order in which they appear in the aspect source code.

The following code shows a simple LARA aspect definition that instruments any C code to output a message whenever a non-system function call is invoked. This is an example of a monitoring aspect that relies on source-to-source instrumentation to automatically generate a new version of the application supporting this new concern. In line 1, we select all function calls in the application. The **select** statement expression (`function.call`) reflects how the information collected is structured: a table with two columns (`$function` and `$call`) in which rows contain all the function calls found in the application

and the respective function definitions that enclose them. This table is then filtered in line 5 by removing all rows in which the **condition** expression is false, that is, when referencing system calls. Next, all the actions in the **apply** section (lines 2-4) are executed for every row in the table. We instrument using the *insert* action (line 3) by placing a `printf` statement just before each call statement, outputting the name of the function definition and the name of function being invoked.

```
1 select function.call end
2 apply
3    $call.insert before %{printf("[[$function.name]]()->[[$call.name]]()\n");}%;
4 end
5 condition !$call.is_sys end
```

Consider the following C code:

```
1 void f() {
2     a();
3 }
4 void g() {
5     f();
6     int *x = malloc(...); ...
7 }
```

When the aspect above is applied to the C description, the weaving process generates the following code as the result. Note that the statement containing the malloc function call is not instrumented.

```
1  void f() {
2       printf("f()->a()\n");
3       a();
4  }
5  void g() {
6       printf("g()->f()\n");
7       f();
8       int *x = malloc(...);
9       ...
10 }
```

One of the strengths of LARA is that we can improve the capabilities and the applicability of the LARA weaving process without making changes to the language. In the example above, the *insert* action performs instrumentation in the context of a source-level translator, however other actions have been added such as *optimise* to perform compiler optimisations [14], *map* has been added to perform hardware/software partitioning, and *wlot* to perform word-length optimisation in the context of reconfigurable hardware [24].

LARA is currently being used to target other languages (MATLAB and C++), and hardware platforms [13]. All these extensions leverage the same *select/apply/condition* mechanisms to support an aspect-oriented design methodology at compile time.

2.3 Summary

In this chapter we have presented an overview of our current compute landscape, including a number of architectures and programming approaches. It is clear to us that two trends are emerging that help overcome the limits of frequency scaling. The first trend is that the number of processing cores will keep increasing and heterogeneity will become mainstream. Amdahl's law has been adapted [46] to show

that systems can greatly benefit from the co-processor model, in which a limited number of fast large cores are employed to execute sequential code very efficiently, while a collection of smaller cores and/or specialised cores (FPGAs and GPGPUs) acting as co-processors are used to execute application hot spots. In particular, specialised cores can have a profound impact on performance and energy efficiency if parts of the application workload can be conveniently adapted to these resources, while computations can exploit smaller general-purpose cores if they are conveniently parallelised. The second trend is that computing scalability will keep growing by offering a large number of loosely coupled multi-core heterogeneous nodes.

How can we combine both trends and exploit heterogeneous multi-core technologies in the context of the cloud platform? This will be the subject of the next chapter.

3 Methods of Characterisation, Management and Development

3.1 Motivation

Cloud computing offers a new paradigm for dynamically provisioning large-scale computation, communication and storage resources to perform a specific task. By focusing on best-resource utilisation, a cloud platform also looks at maximising the availability of resources and the quality of services in a multi-tenancy environment.

The ability for applications to scale across provisioned resources is a key requirement for the HARNESS platform. Much of the current cloud computing landscape realises elasticity through **horizontal scaling**, where virtual and physical machine instances are allocated to complete a job in a distributed setting. **Vertical scaling**, on the other hand, is seldom used in the *platform-as-a-service* (PaaS) context, where parallelised applications meet performance goals by dynamically exploiting available resources within a machine instance [80]. Vertical scaling is mostly relegated to storage, which grows or shrinks according to user requirements.

Another desired property for the HARNESS platform is the use of specialised resources (Section 2.1), which have became increasingly popular because different types of computation can run more efficiently, in terms of energy efficiency and performance, when compared to CPUs. Examples of heterogeneous architectures include: specialised microprocessor cores, GPGPUs and FPGAs. However, the benefits of using specialised resources come with a price: flexibility and ease of programming; not all computations can be mapped efficiently onto accelerators, and they require expertise to conveniently exploit them. Thus, mainstream heterogeneous compute is largely based on the **co-processor model**, in which specific parts of the application, usually computationally intensive, are offloaded to accelerators, while the remaining part (control, memory and I/O dominated parts) remain on general-purpose processors working as the host.

In this chapter, we focus on one of the challenges of WP3 in the context of the HARNESS project: how to optimise workload allocation to heterogeneous multi-core compute nodes. With vertical scaling, an application would be capable of *scaling up* to exploit available resources within a machine, addressing parallelisation and specialised features of heterogeneous resources. With horizontal scaling, an application would *scale out* across multiple machines, dealing with a homogeneous environment that is more vulnerable to faults, security attacks and jittering. The combination of both types of scaling enable us to maximise resource utilisation and efficiency. For instance, jobs currently distributed across multiple nodes (horizontal scaling) can be scheduled to exploit vertical scaling on a machine with specialised resources, thus potentially increase performance while considerably reduce energy consumption.

To realise this approach, we have developed two novel and complementary approaches that focus on: (i) how applications can dynamically exploit heterogeneous compute resources at run time and

(ii) how applications can be programmed to allow a more flexible and efficient mapping to heterogeneous platforms at compile time:

1. **Two-tier run-time computation management system.** We are designing a run-time system that supports heterogeneous compute resources using two management processes: A top-level management process that dynamically allocates workload across distributed compute nodes, and a low-level management process that dynamically adapts the workload across a set of heterogeneous multi-core processors available within a compute node, such as CPUs, DFEs and GPGPUs. By supporting two management tiers, we are able to focus on specific concerns of each domain: the top-level management can focus on horizontal scale concerns such as faults, security, and jittering, while the low-level management can focus on vertical scale concerns, such as exploiting the specific features of heterogeneous physical resources. This organisation also allow us to experiment how jobs can be mapped efficiently on distributed heterogeneous compute nodes: jobs traditionally scaled horizontally can now be scaled vertically and vice versa, providing trade offs in terms of performance, energy consumption and resource utilisation.

2. **Unified heterogeneous programming model.** We are developing a new programming model in which developers can express alternate algorithms and semantics for existing software functions using the same language. This approach provides a non-invasive and non-fragmented description of an application, one that captures multiple variant descriptions for existing software code bases, allowing a more flexible and efficient mapping of software applications to heterogeneous compute resources. In our approach, the software and variant code bases can be part of the same application description. When using a compiler like *GNU compiler collection* (GCC), only the software layer is compiled. However, when we employ our extended compilation flow, the connection between the software and variant layers is enacted through pragma annotations, and a hybrid application targeting multiple heterogeneous compute resources can be generated. In this first year, we have focused on the dataflow semantics to allow efficient mappings of applications to DFEs.

   This unified heterogeneous programming approach also incorporates an aspect-oriented design flow, in which we decouple functional concerns (as described in the software and variant layers) from non-functional concerns, such as performance and energy consumption. With aspects, we capture strategies that convey user knowledge and expertise in a way that can be applied in an automated and systematic way to derive optimised functional descriptions.

We believe that the run-time and compile-time systems described in this chapter allow us to exploit current trends of technology, one that is becoming increasingly more parallel, more heterogeneous and more distributed.

The remainder of this chapter is structured as follows. In Section 3.2, we cover the requirements and main outcomes of our research. Section 3.3 provides an overview of our run-time and compile-time approaches and how they integrate with the larger platform. Sections 3.4 and 3.5 provide a more in-depth view of how the two-tier run-time management organisation supports heterogeneity and scalability, respectively. Finally Section 3.6 provides more details about the unified heterogeneous programming model.

3.2 Requirements and Outcomes

In WP3 we have identified three main outcomes to implement the integrated HARNESS platform. First, to develop a run-time computation management system that is capable of scaling computations over distributed heterogeneous multi-core machines. In Deliverable D2.1 [31] we specified three general requirements that relate to the run-time management system:

Outcome 1: Run-Time Management System	
R11. Provide characterisation of heterogeneous applications.	To characterise low-level tasks, by heterogeneous resource used, based on the quantity of data, data transfer speed and other metrics, such that the benefit of using such a resource is known at run time.
R12. Enable virtualisation and shared use of heterogeneous computational resources.	To virtualise heterogeneous compute resources, such as GPGPUs and DFEs, in order to support resource sharing between multiple jobs and maximise the benefits of heterogeneous computational units.
R14. Optimise the use of resources according to optimisation goals within constraints provided by the cloud platform layer.	To satisfy performance goals on provisioned resources set by the cloud platform, by choosing to run compute tasks on the best available resources at run time. Performance goals may include minimising job completion time, use the highest numerical precision or the lowest energy consumption.

Second, to address the complexity of programming accelerators and to improve design productivity and maintainability, we wish to develop the next generation *compiler tools and programming models* that enable users of the cloud platform and domain experts in hardware platforms and applications to generate designs that exploit the features of our run-time management system as explained above. In Deliverable D2.1, we identified one general requirement:

Outcome 2: Programming Infrastructure for Heterogeneous Platforms	
R10. Develop a compiler infrastructure to derive multi-target design variants.	To develop a compiler infrastructure that captures domain-specific knowledge about the application and/or hardware platform to be codified and subsequently applied in a systematic and automated way to generate multiple designs that satisfy different non-functional concerns such as performance, resource utilisation and energy efficiency.

Third, to integrate our run-time computation management system (outcome 1) and programming infrastructure (outcome 2) with the cloud platform being developed in WP6. In Deliverable D2.1, we identified two general requirements related to platform integration:

Outcome 3: Cloud Platform Integration	
R9. Develop a heterogeneous computational resource discovery protocol.	To provide the cloud platform layer with an inventory of available computational resources such as CPUs, FPGAs and GPGPUs and relevant attributes, so that the cloud platform can make resource provisioning decisions that will affect the run-time computation management process when scheduling tasks.
R13. Provide monitoring data to support decision making at the cloud platform layer.	To provide monitoring data about resource utilisation and performance to the cloud platform layer so that it can learn from and improve prior provisioning decisions.

3.3 Approach Overview

In this section we provide an overview of our proposed approach, including: (i) the run-time computation management system; (ii) the programming infrastructure for heterogeneous platforms; and (iii) cloud platform integration.

3.3.1 Run-time computation management system

Our two-tier run-time computation management system is presented in Figure 3.1. When executing an application, the cloud platform must first allocate and provision physical compute resources for this purpose. These resources (e.g. FPGAs and GPGPUs) can belong to a single machine or to multiple machines. In addition, the cloud platform must perform the following actions:

- **Initialise the host.** Launch an instance of the run-time computation management system on a virtual or physical machine acting as the host.
- **Initialise compute resources.** Make local and remote compute resources available to the host machine. Like storage resources using NFS and SMB mounts, compute resource can be *local* (attached to the host machine) or *remote* (attached to other machines).
- **Launch one or more jobs**. Dispatch jobs to the compute management system.

For the purpose of this report and without loss of generality, a **job** is defined as a unit of work running (parts of) a cloud application. Jobs dispatched to the compute management system run on a host machine using a general-purpose processor (multi-core CPU). The application code may contain calls to a special type of function, which we call a **task**, that is managed by a process called the executive. More complex representations of the application can include a *task graph*, which captures dependencies between tasks along with other information that can help optimise the scheduling process.

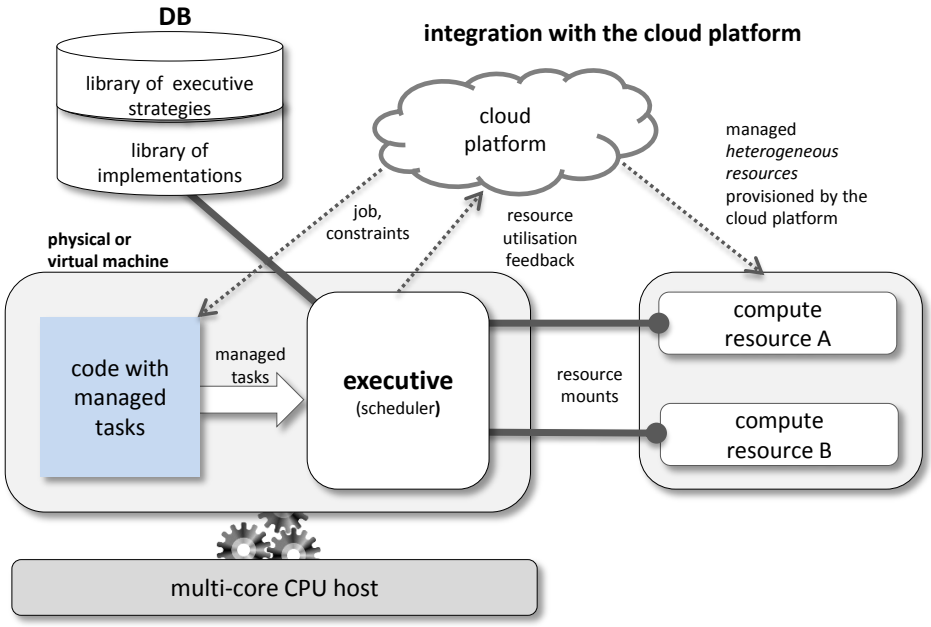


Figure 3.1: The run-time computation management system provides a framework that decouples the functionality of an application launched on a cloud platform from the distributed heterogeneous infrastructure that lies underneath it. The novelty of this run-time system lies in a two-tier management system: an *executive module* that decides which compute resources to use to complete a particular job to perform horizontal scale, and *compute resources*, which manage one or more physical compute elements residing on the same machine to perform vertical scale. Compute resources can be available from remote sites, allowing the executive running on the host machine to allocate workloads across remote machines.

Management method

The **executive** is responsible for deciding for each task invocation, which of the available compute resources mounted on the host machine is the most appropriate to achieve a particular execution goal and satisfy given constraints. Decisions are made primarily on the availability of implementations that realise the task on a specific **compute resource**. In our current organisation, an **implementation** is associated to a specific task and can be executed on a single type of compute resource. Implementations and their characterisation are stored in a database (DB) and accessed by the executive to create a list of candidate implementations to realise a given task.

The executive must make a decision about: (i) which implementation to use and (ii) on which resource to execute that implementation. While there is a one-to-one mapping between an implementation and a resource type, there can be multiple resources of the same type. The algorithm that makes the decision is called an **executive strategy**. There can be different strategies that can be applied according to the objectives established by the platform when processing a job. We envision that a trade off between the time to make a decision and the quality of that decision can be leveraged to adapt to a particular run-time context. For instance, jobs that take hours may benefit from a decision that optimises resource utilisation, while jobs that require milliseconds to complete may benefit from fast decisions. The compute resource, on the other hand, provides a low-level management for one or more physical devices, such as DFEs and GPGPUs. In addition to allowing these devices to be shared, they also hide heterogeneity by providing a standard interface to the executive to dispatch tasks and acquire monitoring information. Sections 3.4 and 3.5 provide a more in-depth view of how our two-tier run-time management organisation manages heterogeneity and scalability, respectively.

Characterisation method

Given a set of implementations, an executive strategy can choose the most appropriate implementation and resource based on a combination of factors, which may include:

- **Size and type of input data.** The size and type of input data can influence the choice of implementations that best realise a task. This is especially true in the context of the proposed run-time computation management system, in which compute resources act as co-processors and require data transfer from and to the host. Implementations supporting different data types, including those with different floating-point precision, can be chosen on the basis of user-desired numerical accuracy.

- **User characterisation of input data.** In addition to size and data types, there may be other characterisations of input data that are tied to the functionality of the task, and can help the executive make better optimisation decisions. For instance, the executive may choose an insertion sort over a quicksort if input data has been characterised to be partially sorted, since quicksort has worse time complexity in this case.

- **Availability of compute resources.** An implementation may be available for a particular task, but it can only be used if compatible resources are mounted and available to the executive. In addition, an available resource may be busy, which may force the executive to find an alternative mapping solution.

- **Performance models.** Each implementation may be associated with a performance model that allows the executive to estimate the run-time performance on its target resource. For instance, a performance model could be used to estimate the execution time of a task implementation as a function of the input data size. Other performance models can support other goals such as power consumption and resource utilisation, and take into account multiple input arguments and their characterisation.
- **Accrued historical data.** During run time, the initial performance model may be updated based on run-time monitoring to provide a more accurate estimation of performance.
- **Location of compute state.** To optimise job completion time, intermediate results (state) may be kept inside the resource during multiple task executions. This avoids unnecessary data movements, but the executive needs to keep track of the state location to ensure that the correct resource is selected, or move state to a new resource if required.

The following requirements from Deliverable D2.1, which relate to the run-time management system, are addressed as follows:

R11. Our approach supports *implementations*, which we define as the mapping between an application kernel and a specific compute resource. Implementations and their characterisation (such as performance models) are stored in a database, allowing the run-time management system to use that information to make a decision about which implementation to use to execute a task. Our compile-time system is capable of generating implementations and partially characterise implementations based on profiling and synthesis reports (e.g. performance and power estimates).

R12. Our approach supports *compute resources*, which are virtualised compute nodes that manage one or more physical processing elements, such as FPGAs and GPGPUs. The key benefit of compute resources is providing a low-level management layer that enables resource sharing between multiple jobs.

R14. The executive component is responsible for executing a job by employing the compute resources allocated by the cloud platform, and satisfying optimisation goals and constraints specified by the platform. One key feature of the executive is that strategies can be customised, allowing the use of scheduling algorithms that best suit a particular problem, and to support a trade off between quality of the scheduled solution and the time to complete the schedule.

3.3.2 Programming infrastructure for heterogeneous platforms

We propose a novel programming infrastructure that addresses key areas of design development for heterogeneous platforms at compile-time. In particular we aim to: (i) facilitate the generation of variant implementations from kernel descriptions and (ii) accelerate existing applications by mapping legacy source code to the HARNESS cloud platform using our run-time management system.

Our proposed programming infrastructure is being designed to support the following properties:

1. **Performance:** to capture descriptions that that can facilitate the mapping to specialised hardware in order to achieve significant speedup over sequentially executed implementations on conventional general purpose processors;

2. **Maintainability:** to improve maintainability of optimised designs by separating functional and non-functional concerns;

3. **Portability:** to simplify translation of existing applications to high-performance designs to facilitate the integration of the proposed design flow with existing (predominantly imperative) application code; and

4. **Productivity:** to improve developer productivity by providing a way to capture and codify optimising and parameterising compilation strategies, including design-space exploration, that can be reapplied through an automated mechanism.

Development method

To meet these objectives we are developing a unified programming model based on two language descriptions. First, the ***Facile Aspect-driven Source Transformations* (FAST) language** [42], developed within the HARNESS project, captures functional (algorithmic) descriptions using a single language specification to support multiple programming semantics. The syntax of FAST is C99, however an application can be composed of two code- base layers that can coexist in a non-invasive way: the software layer and the variant layer. The variant layer is composed of *domain-specific computation* (DSC) modules that are governed by alternative semantics that lead to a more natural mapping to specialised resources. For instance, to leverage dataflow engines (Section 2.1.2) we employ DSC modules that conform to dataflow semantics. By maintaining compatibility with the C99 syntax, we improve developer productivity by providing a familiar language and allow software and hardware specifications to be combined.

Second, we use the **LARA aspect language** to capture non-functional concerns. By using an aspect-driven compilation flow we decouple optimisation from design development, improving design maintainability. Moreover, we improve design portability by introducing *portability* aspects that map legacy source code to our run-time management system. Finally, to improve productivity we automate the generation of code with optimisation strategies that can be applied automatically and use systematic design space exploration to identify maximum performance configurations, subject to platform specific constraints.

The proposed aspect-driven design-flow infrastructure is illustrated in Figure 3.2. To support **legacy applications**, our design flow would work as follows:

1. The inputs to the design flow are: (i) C99 source code and (ii) LARA portability aspects.

2. The source-level weaver, guided by the portability aspects, automatically transforms the input source code to interface the run-time management system. This includes identifying tasks in the application that can be accelerated, and substituting this code with calls to managed tasks and adding the necessary headers. The identification of tasks in the application can be done though different methods. The simplest method would include identifying names of functions (e.g. sort) that follow a specific interface and substituting them with calls to managed tasks. More complicated porting aspects include targeting applications that contain platform-specific APIs and identifying code idioms.

3. The end-result of this weaving process is an automatically generated application source code that interfaces with the run-time management system, and thus benefiting from acceleration using specialised resources.

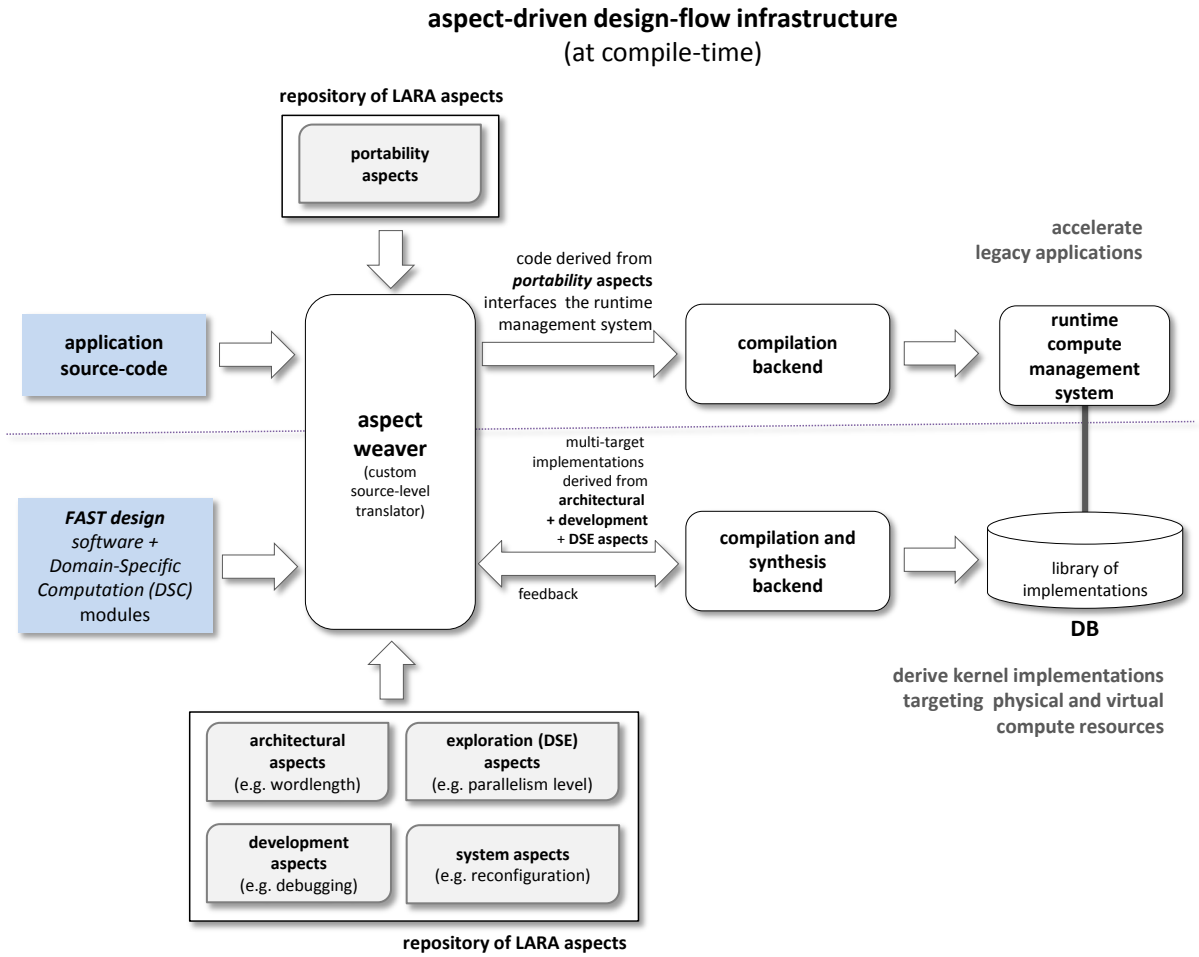


Figure 3.2: Our aspect-driven design-flow includes a source-level translator weaver [25], which uses a repository of LARA aspects developed in HARNESS to: (i) translate DSC modules to multi-target implementations that are made available to the run-time management system and (ii) port legacy application source code to exploit the run-time management system.

The key advantage of the compilation approach is that it allows code written for conventional CPU processors to exploit specialised resources with little or no effort from developers. To support the generation of **multi-target variant implementations**, the design flow presented in Figure 3.2 works as follows:

1. The inputs to the design flow are: (i) FAST source code (based on C99 syntax) containing software and DSC modules capturing the functionality of the design and (ii) LARA aspects that satisfy non-functional concerns, such as architectural, development and design-space exploration.
2. DSC modules are transformed by architectural and design-space exploration aspects to generate new variant designs (e.g. with multiple word-length configurations).
3. DSC modules are transformed by development aspects to facilitate design processes, such as automatically instrumenting the code to monitor specific variables.
4. The generated configurations are compiled using back ends according to the model of computation used. Currently, DSC modules supporting dataflow designs are synthesised with MaxCompiler tools for hardware synthesis.
5. The feedback from the compilation process is used to drive the design-space exploration, repeating the weaving and compilation process until user-specified requirements are met.

The following requirement from Deliverable D2.1, relating to the programming infrastructure, is addressed as follows:

R10. The compile-time infrastructure includes a source-to-source weaver to support the generation of multiple variant implementations using aspects. In our approach, these aspect descriptions capture optimising methods and design-space exploration strategies. The systematic and automated generation of multi-target variant implementations allow cloud providers (and potentially cloud tenants) to generate optimised kernels for various hardware platforms. The process of deriving variant implementations involves traversing a large design space and taking into account different application parameters and architectural features. Advanced strategies for design-space exploration also involve identifying Pareto-optimal implementations that can maximise, for instance, the trade off between performance and energy consumption. In this context, our aspect-oriented technology can automate the generation of performance models as part of the resource characterisation process.

Further details of our programming approach are provided in sections 3.6, 4.3, and 5.2.1, which cover the uniform heterogeneous programming model, its implementation and preliminary results.

3.3.3 Platform integration

As discussed in Section 3.3.1, the cloud platform is responsible for provisioning a set of compute resources to complete a given job. The run-time computation management system, on the other hand, is responsible for exploiting provisioned resources the best way possible to complete the job. The scope and the frequency in which decisions are made by both management systems are different. The cloud platform has a global view of all compute, storage and communication resources that are part of the data centre. And thus, the platform can make mapping decisions that have a larger impact on how resources are utilised in the data centre, and equally important, to reason about the implications of those decisions. This includes detecting and resolving conflicts between local management processes. The compute management system, on the other hand, makes mapping decisions at a local level and at higher throughput taking into account provisioned compute resources. The executive keeps track of resource utilisation and

informs the cloud platform whether resources are being under-utilised or under-provisioned, allowing the platform to refine and improve its budgeting decisions in subsequent iterations with the run-time computation management system.

The interactions between the run-time management system and the cloud platform (related to resource reservation, application deployment and monitoring) will be performed via the *infrastructure resource scheduler* component described in Deliverable D6.3.1 [36]. The following two requirements defined in Deliverable D2.1, related to platform integration, are addressed as follows:

R9. Our approach supports a generic interface supporting the discovery of compute resources from each machine node, and to report this information to the cloud platform.

R13. The executive component consolidates the monitoring information from different compute resources, and this information is reported back to the cloud platform. For instance, it can report which physical compute resources are being under-utilised.

More information about platform integration can be found in Section 6.2.

3.4 Managing Heterogeneity

In current computational systems, heterogeneity is largely invisible and only minimal "management" functionality is provided. Accelerators such as GPGPUs or DFEs are often accessed as I/O devices via library-call interfaces. These accelerators must be manually managed by the application programmer, including not just execution of code but also in many cases tasks that are traditionally performed by an operating system such as allocation, de-allocation, load balancing, and context switching. If resources are shared between several hosts (such as Maxeler's MPC-X series dataflow appliances, where DFEs are dynamically allocated to CPU processes across the network), issues of contention, fairness and security become further pronounced.

Since heterogeneous compute elements are generally outside the control of the operating system or other system software, global optimisation of resources for example for energy efficiency is not practical. Instead, local management and optimisation of different resources may be possible. For example, MaxelerOS manages the allocation of DFEs, while a graphics driver manages the use of GPGPUs in a system. Accelerators may provide diagnostic information such as temperature and power consumption, but these will generally be made available via proprietary interfaces. Control interfaces may include the ability to overclock or under-clock resources but again these interfaces are proprietary and usually not leveraged at a system level.

To address these problems, we have developed a compute management system to support cross-optimisation of multiple kinds of heterogeneous resources. This system, introduced in Section 3.3.1, has two management tiers (Figure 3.1). In the first tier we have the *executive* component. The executive makes decisions about which *compute resources* to employ to complete a given job, and to collect and process diagnostic information from these resources, and make them available to the platform. In the second management tier we have the *compute resource*, which is a virtual compute element that can manage one or more tightly coupled devices.

Since heterogeneous devices support proprietary interfaces or require specialised drivers (as is the case of OpenCL), we designed a uniform interface that all compute resources must adhere to in order to used by the executive. The compute resource interface provides the following features:

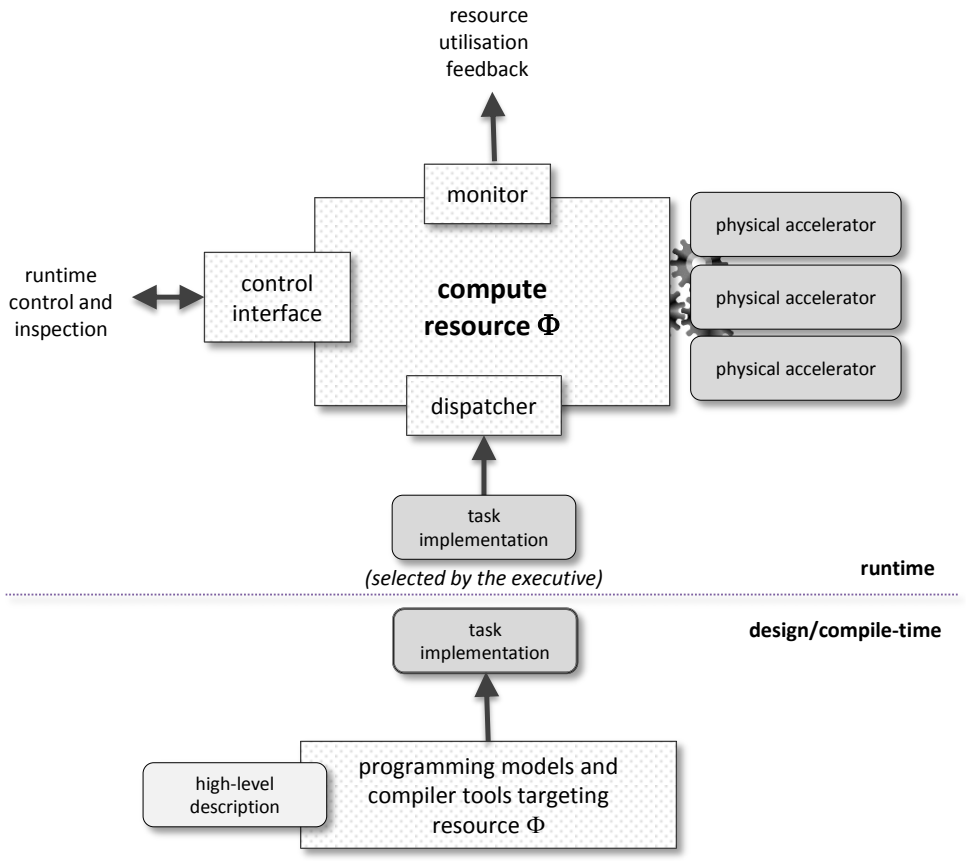


Figure 3.3: A compute resource, which is a virtual compute element that manages one or more physical compute elements such as FPGAs and GPGPUs. Compute resources wrap the complexity and heterogeneity of specialised resources, allowing the executive component to deal with resources using a standard interface that enables low-level control, task execution and monitoring. Each compute resource may also support specific programming models and compiler tools to develop optimised implementations for this target.

- **Dispatcher.** The dispatcher allows the executive to forward tasks to be executed on managed physical processing elements such as DFEs.
- **Monitor.** This interfaces allows the executive to acquire information about key resource utilisation metrics, such as information about load and temperature.
- **Control.** This interface provides fine-grained control over the compute resource node. For instance, at a given point in time, the executive may use this interface to reduce the number of physical processing elements if they are being under-utilised. Furthermore, it can be used to inspect the status of the resource.

Hence, compute resources hide the complexity of managing physical processing elements, offer a single interface that facilitates the integration of current and future specialised hardware platforms, and allow compute resources to be shared across multiple jobs. Compute resources can also support programming models and tools that help derive efficient implementations for these targets (Figure 3.3).

3.5 Managing Scalability

At the beginning of this chapter we mention two types of scaling: horizontal and vertical scaling. With horizontal scaling, the application's workload is allocated across a number of machines. With vertical scaling, on the other hand, computation is dynamically scaled within the processing elements of a machine. The definition of what constitutes a machine or a compute node in a cluster can be subjective, and thus, the boundary between these two types of scaling can be best expressed by their distinct set of concerns:

- In **horizontal scaling**, we find large-scale compute nodes connected through network resources (switches, routers). These compute nodes, while supporting different hardware configurations and supporting specialised resources, present themselves as *homogeneous* with support for *general-purpose compute* on a number of standard platforms, such as *Java virtual machine* (JVM) or Web programming frameworks like Google AppEngine [40]. Furthermore, these compute nodes, even with the same hardware, may take an unpredictable amount of time to complete a job. When a large number of these nodes are combined with a network infrastructure, there are concerns such as *jittering*, where small differences in the run time of each node can lead to a big difference in the overall execution of an application. Due to the limitations and reliability of the infrastructure, data locality, security and fault tolerance may also be prime concerns in this domain.
- In **vertical scaling**, we find *heterogeneous* processors tightly connected through system busses or fast network interfaces, and where the time to complete a job is more predictable. In particular, designs running on FPGAs take a specific number of cycles to complete a job, even when considering concurrent computations. However, since FPGAs interact with software in the co-processor model, the overall performance may not be totally predictable. The prime concern when scaling a computation across heterogeneous processors in this domain is *hardware/software partitioning* (Section 2.1.3), where parts of the application are selected and optimised to suit specific hardware resources. For instance, loops with a large iteration space that are computationally bound are prime candidates to be offloaded to an FPGA (Section 2.1.2). Hardware/software partitioning is critical as not all computations can be mapped to FPGAs or be efficiently mapped. Another concern

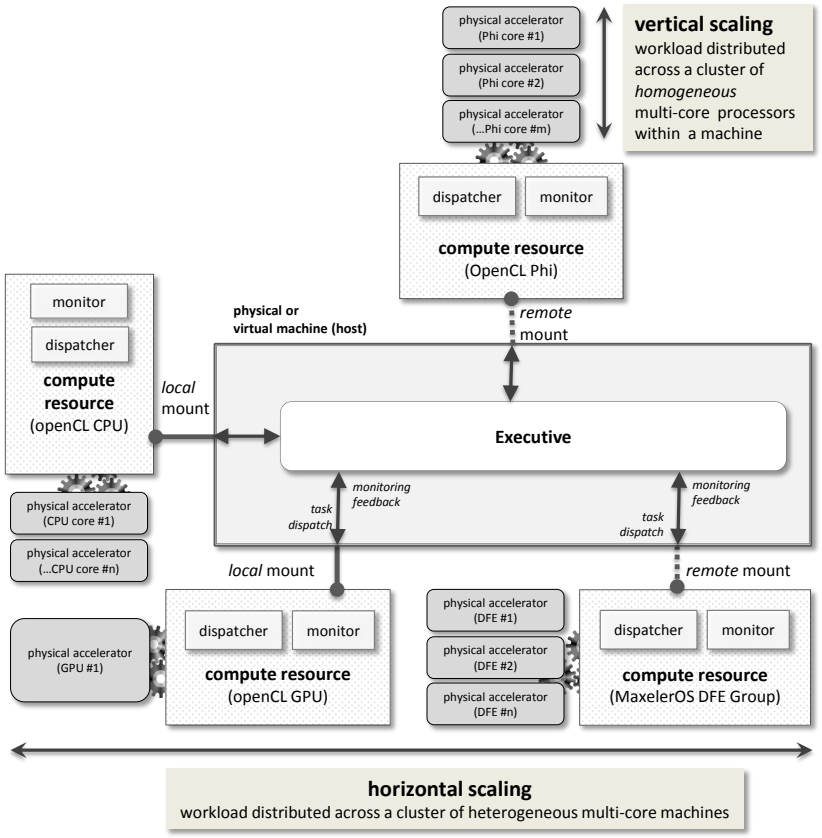


Figure 3.4: Our run-time management system supports two types of scalability. Horizontal scale is performed by the executive, which allocates the workload across compute resources that may be installed in different machines. Vertical scale is performed internally by compute resources that can manage multiple physical processing elements. The key difference is that horizontal scale deals with concerns such as communication latency, faults and unpredictable response times, while vertical scale must focus on exploiting specialised architectural features of physical processing elements.

related to hardware/software partitioning is the communication cost involving data transfers from the host processor (a CPU) to the accelerator and back.

Our **two-tier run-time management system** (Section 3.3.1) addresses the differences between these two types of scaling, as illustrated in Figure 3.4. The executive component, at the top level, allocates the workload to provisioned compute resources. Compute resources can be installed in different machines, and are made available through a socket-level interface to the host machine where the executive operates. Thus, the executive is capable of scaling the computation across multiple machines—that is, horizontally, and the scheduling algorithm (executive strategy) can take into account concerns such as communication latency and faults through diagnostic information provided by the compute resource. Compute resources, on the other hand, offer a low-level management tier of tightly coupled processors, allowing a computation to scale vertically at this level. Unlike the executive, which operates with tasks, a compute resource

operates with task implementations that exploit specialised architectural features. Table 3.2 summarises the key differences between the executive and the compute resource.

By combining both types of scaling, our approach provides the flexibility to utilise resources in the most efficient way possible: scaling vertically allows us to exploit tightly coupled specialised accelerators to maximise performance while reducing energy footprint; scaling horizontally enables us to use a vast amount of loosely coupled distributed compute resources. As far as we know, this multi-scaling capability is not available in any PaaS solution currently available, including Google AppEngine [40], Elastic Beanstalk [2], Microsoft Azure [62], OpenShift [69], Cloud Foundry [20] and *Contrail platform-as-a-service* (ConPaaS) [22]. We intend to integrate our approach with ConPaaS as part of the integration activities of WP6 in Year 2 (Section 6.2).

Executive	**Compute Resource**
scales computations horizontally	scales computations vertically
operates on tasks	operates on implementations
operates on (virtual) compute resources	operates on physical processing elements
concerns: fault tolerance, security, jittering	concerns: exploit specialised/dedicated features
monitoring info acquired from a standard compute resource interface	monitoring info acquired from a proprietary interface

Table 3.2: Comparison between two run-time management tiers: executive and compute resource.

3.6 A Unified Heterogeneous Programming Model

While multi-core heterogeneous systems can run computations orders of magnitude faster than single-core and many-core platforms, they are still difficult to program and maintain. Part of this problem is because mainstream software languages have limited ways to express a computation, which may lead to an inefficient mapping between the application and the target architecture. To develop an efficient implementation, developers need to choose the appropriate algorithm, use the appropriate model of computation, and exploit the appropriate architectural features. As an alternative to a single language approach that rules all heterogeneous devices (such as Xilinx Vivado HLS or IBM LIME), we can use different programming models and tools that are optimised to different processing elements in a heterogeneous platform: e.g. OpenCL for GPGPUs, MaxCompiler for FPGAs, and C++ for CPUs. This approach, however, is limited by the fact that optimisations are difficult to perform at the application level and, instead, must be performed locally at each partition. In addition, developers must usually contend with different languages and back ends.

We propose a novel unified programming approach, FAST, introduced in Section 3.3.2. It captures multiple programming semantics using the same language syntax, namely C99. With this approach, developers can embed C99 modules (called DSCs) acting as variants to a software function. These variants can represent (i) alternative software implementations, (ii) accelerator implementations, and (iii) hybrid software/hardware implementations. For instance, multiple matrix multiplier implementations may be available to cater to different sizes or requirements relating to numerical representation.

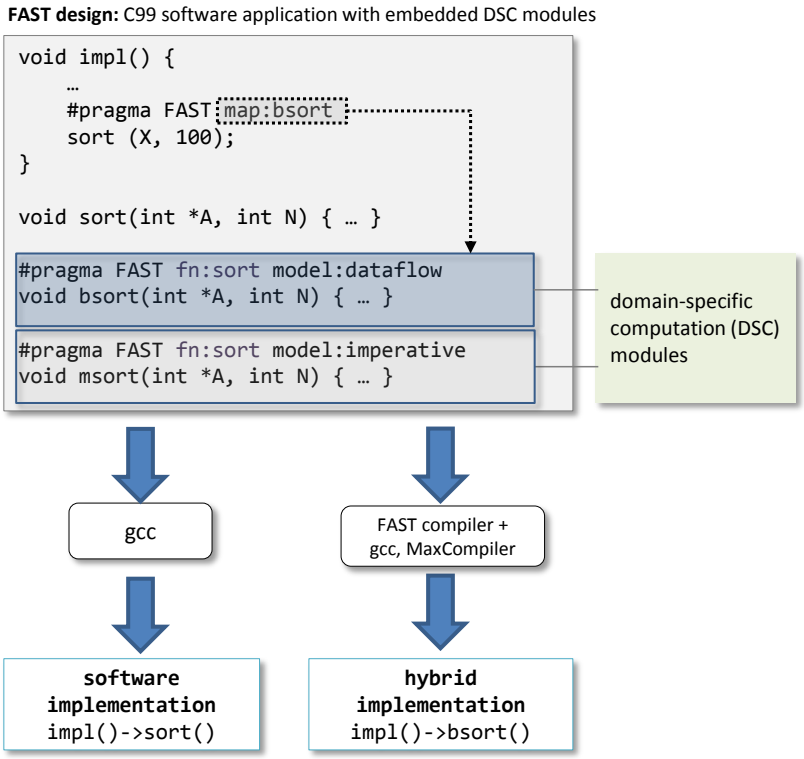


Figure 3.5: A FAST program. A FAST program contains two code-base layers, both described in C99. The first layer is the software code base capturing the functional description of the application. In this example, the software layer includes the `impl()` function definition, which invokes the `sort()` function. When a FAST program is compiled with GCC (bottom left) a pure software version is derived. The second layer is the variant layer, and consists of a set of C99 functions each decorated with a `#pragma FAST` annotation that identifies it as a DSC module and the software function with which it is associated. In addition, the pragma annotation captures the model of computation of the DSC module. In this example, we have two DSC modules defined that are associated with the `sort()` function: `bsort`, which adheres to dataflow compute semantics, and `msort`, which adheres to imperative compute semantics. In this design, we have a mapping pragma that associates the software function `sort` to variant `bsort`. An alternative design flow (which includes the FAST compiler, MaxCompiler and GCC) generates a hybrid design, linking the software layer to the variant layer.

A FAST program has two code layers. The first is the software layer. The second layer, which is known as the variant layer, is comprised of a set of DSC modules that are associated with functions from the software layer. To associate a DSC module to a particular software function, we add a C function definition in any part of the software code base, and then place on top of that definition a FAST `#pragma` annotation (see Figure 3.5). For instance:

```
#pragma FAST fn:sort model:dataflow
void fn_dfe(...) { }
```

The pragma above specifies that the DSC module `fn_dfe` is an implementation of the software function `sort()`, and it follows the semantics of dataflow computing.

Currently, we support two programming semantics for DSC modules: imperative and dataflow. The imperative DSC modules follow the traditional programming model of C, but we wish to make a distinction between software functional code (first layer) that will always run general-purpose processors, and DSC modules (second layer) that could target accelerators including multi-core CPUs.

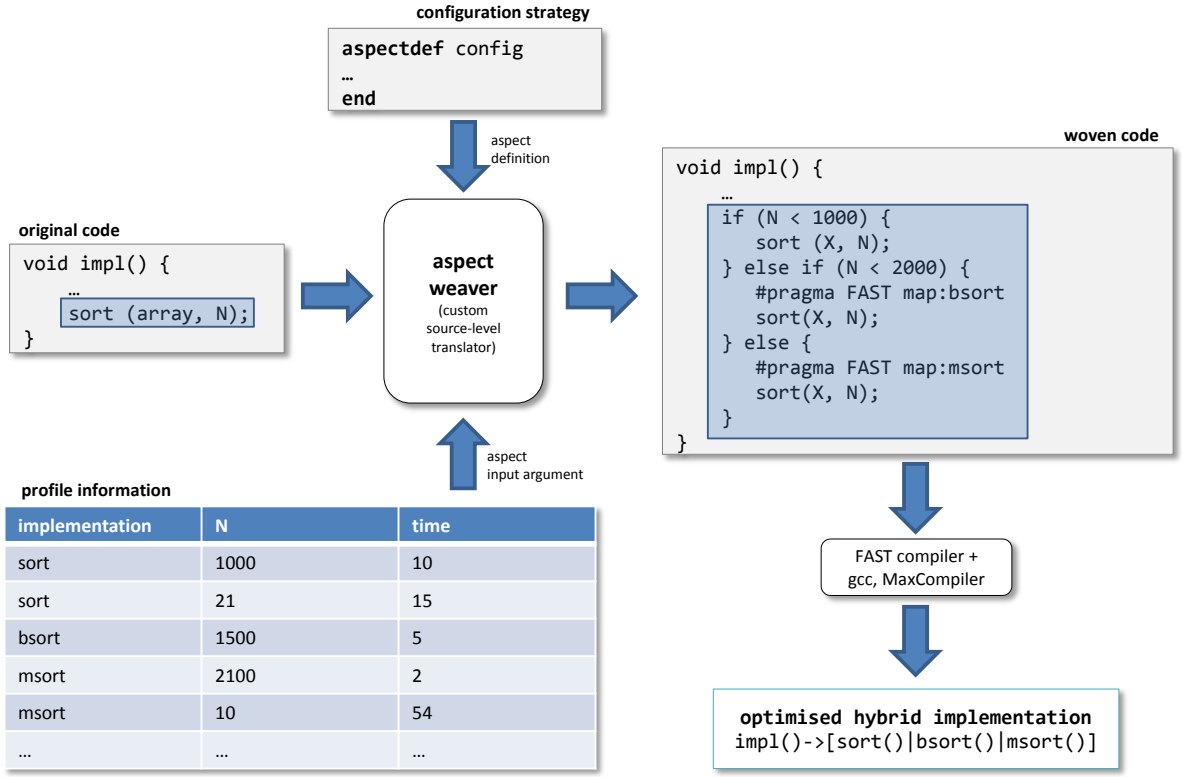
Functions from the software layer cannot invoke DSC modules directly. Instead, they must use a pragma on top of the software function call:

```
#pragma FAST map:bsort
sort();
```

This rule is enforced so that we can always compile a FAST program with GCC and get the software build. In addition, while DSC modules can be compiled with GCC and executed, GCC does not support the semantics of dataflow computing. To synthesise applications with DSC modules, we use an alternative design flow comprised of the FAST compiler, GCC and MaxCompiler. The FAST compiler scans for the `#pragma FAST map` pragmas, substituting software calls with calls to synthesised/compiled DSC modules. The FAST compiler converts C99 dataflow DSC modules to MaxJ, and uses the MaxCompiler tool chain to synthesise the design. Imperative DSC modules are compiled into library modules using GCC tools. DSC modules can invoke other DSC modules through normal C calls.

Our unified programming model thus supports a non-invasive mechanism that allows variants to be introduced to the software code base without affecting its original functionality. In addition, developers can express with the same language not only alternative algorithms to original software functions, but also different semantics, such as dataflow computing, which are more appropriate for targeting specialised resources such as FPGAs. Having a single programming environment prevents code fragmentation, making the application more portable and susceptible for optimisation at a global level.

Figure 3.6 illustrates an example of how our aspect-oriented design flow can be used to exploit this unified programming model. In this example, a configuration aspect codifies a strategy that, given a table with profiling results for available implementations of `sort`, computes the cutoff points and run-time scenarios in which specific implementations provide the most efficient performance. The result of this analysis can be used to transform the original code that invokes the `sort` function to code that invokes multiple DSC modules according to computed scenarios. While in this example we use aspects to codify when DSC modules are invoked from the host, we can also have specific strategies that modify and transform DSC modules directly. Section 4.3 presents a set of aspects that have been developed to improve dataflow design. We report the evaluation of this work in Section 5.2.1.



implementation	N	time
sort	1000	10
sort	21	15
bsort	1500	5
msort	2100	2
msort	10	54
…	…	…

Figure 3.6: Using a uniform programming model to capture software and variant layers enables us to use our aspect-oriented design flow to codify strategies that optimise either global application or local DSC modules using the same mechanism. In this example, we use an aspect to analyse the performance models of all implementations of `sort` available, to compute the cut-points where each implementation may offer best performance, and to automatically output (weave) code that calls the best implementations as result of that analysis.

Feature	Description	Method (see Fig. 3.7)
Input/Output	Declared in function header	C99 (line 2)
Control	Ternary op.	C99 (line 12)
	Stream mux (`mux()`)	FAST primitive
Computation	+, *, /, -	C99 (line 10)
	log, exp, sqrt, sin etc.	#include <math.h>
Streams	Declared as pointers	C99 (line 7)
	Accessed with array index	C99 (line 10)
Hardware Mapping	C pragmas	C99 (line 9)
Parameterisation	Constants, variables	C99

Table 3.3: Summary of the main features of the FAST language for the dataflow model.

Table 3.3 summarises the features of FAST, and Figure. 3.7 presents an example of two DSC modules that implement a 1D convolution using dataflow and imperative models, respectively. As can be seen, a dataflow DSC module (lines 1–13) is defined as a regular C function. Streams are represented as regular C99-style pointers (line 7). Normal array notation can be used to generate either previous or future values or de-reference the stream to obtain the current stream value. Negative indices are allowed for accessing previous stream values and supported offset expressions are linear expressions comprised of constants or variables (either loop induction variables, or normal variables but for which a compile-time range of values is specified, as a requirement for generating efficient hardware). Constructs such as loops are supported as long as their bounds are known at compilation time and are used to parameterise dataflow designs. FAST supports specific primitives and pragma notations for each DSC model. For instance, for dataflow modules, there is support for stream multiplexers (mux), and pragmas that map computations to hardware resources such as DSPs (line 9).

In this example, the dataflow DSC module (lines 1–13), while syntactically correct, does not produce correct results when compiled with GCC. This is due to the fact that GCC is a software compiler, and converts code into a list of instructions, in which operations are processed at different points in time by a fixed architecture; hence the computation is performed in a time domain. In a dataflow model, computation is translated into functional units that are laid out in space, where data are computed in parallel and forwarded to the next functional unit; hence dataflow computation is said to be performed in a spatial domain. Dataflow DSC modules are translated to MaxJ by the FAST compiler, and automatically linked with the software layer.

3.7 Summary

In this chapter we present the general methods for characterising, managing and developing applications to optimise workload allocation to heterogeneous multi-core compute nodes. We have divided our work

```
1     #pragma FAST fn:Convolution1D model:dataflow
2     void Convolution1D_dx(float *in, float *out, float c0,
3                           float cp, float cn, int n)
4
5    {
6       float i = count(n);
7       float* result; bool *up;
8
9       #pragma FAST DSPBalance:full
10      result[0] = in[0]*c0 + in[1]*cp + in[-1]*cn;
11      up[0] = (i >= 1) && (i < n - 1);
12      out[0] = up[0]? result[0]:in[0];
13    }
14
15    #pragma FAST fn:Convolution1D model:imperative
16    void Convolution1D_ix(float *in, float *out, float c0,
17                          float cp, float cn, int n)
18    {
19       for (int i = 1; i < n - 1; i++) {
20          out[i] = in[i]*c0 + in[i+1]*cp + in[i-1]*cn;
21       }
22    }
```

Figure 3.7: Two FAST DSC modules implementing 1D convolution using dataflow and imperative semantics, respectively.

into two systems that complement each other: (i) a run-time management system and (ii) a programming infrastructure targeting heterogeneous platforms.

The run-time management system is based on a novel two-tier scheme (Figure 3.4) in which each management tier caters to different concerns when dynamically scaling computation across available resources. At the first tier, we have the *executive* component, which allocates workloads to available *compute resources* distributed across compute nodes in a data centre. At the second tier, we have *compute resources*, which are virtual processing elements that offer low-level management of physical processing elements, including (single or multiple) FPGAs and GPGPUs. In addition, they offer the executive a uniform interface to support task execution, resource sharing and to receive diagnostic information from a wide range of resources. Compute resources aim to offer optimised utilisation of physical resources while providing a simpler programming interface to developers. Moreover, compute resources can be employed to scale the computation vertically when managing multiple processing elements. By combining both types of scaling, our approach provides the ability to exploit resources more efficiently: scaling vertically enables elasticity on tightly coupled specialised accelerators to maximise performance while reducing energy footprint; scaling horizontally enables the use of vast amounts of loosely coupled distributed compute resources.

Our programming infrastructure supports two novel features. First, we provide a unified heterogeneous programming model that supports multiple models of computation using the same programming language (C99). This allows programmers to offer alternate descriptions of a software function, which we call variants, facilitating the generation of efficient mappings, in particular to specialised resources.

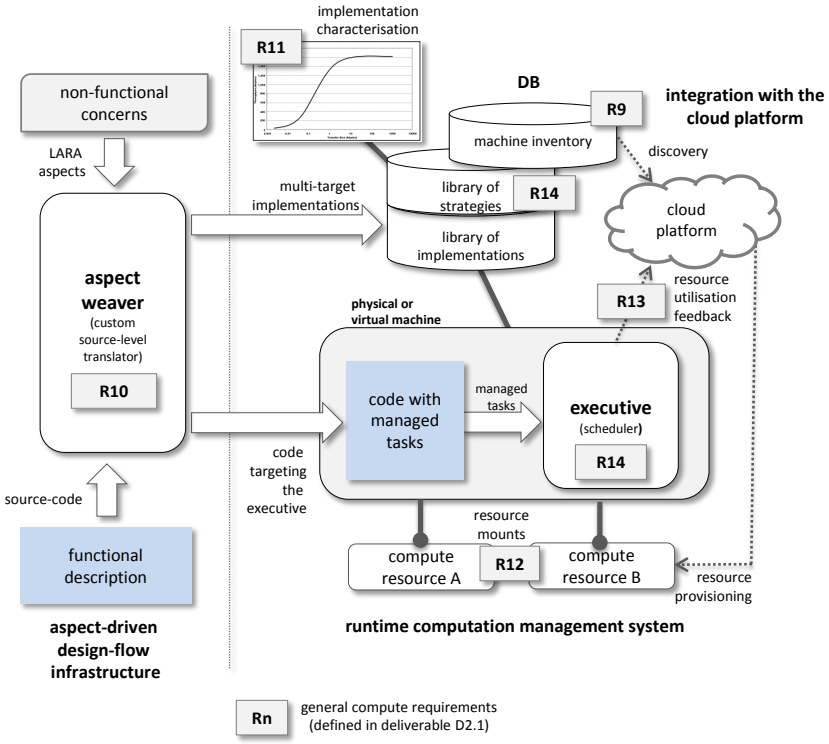


Figure 3.8: Integration between different components of our proposed approach: (a) aspect-driven design-flow (compile time); (b) run-time management infrastructure; and (c) cloud platform. This figure also illustrates the general computation requirements defined in D2.1 [31] that must be addressed by these components.

Furthermore, because software and variant module descriptions are in the same code base, system optimisations can be performed with the same compiler infrastructure. Second, our unified programming model is supported by an aspect-oriented programming facility, in which we decouple functional and non-functional concerns. This way, functional descriptions are captured by the C99 language (software and variants), and non-functional concerns by the LARA language. Non-functional concerns include automatically deriving and characterising optimised variant implementations that can be used by the run-time computation management system to accelerate applications.

Our work in Year 1 has been mainly driven by the general requirements specified in Deliverable D2.1 (Figure 3.8). We expect that in Year 2, our work will be more focused on addressing the application requirements (Deliverable D2.2) and the validation plan (Deliverable D2.3). In this context, we believe that each HARNESS validation use case will provide different challenges in terms of how jobs, user requirements, implementations and resources are characterised.

4 Infrastructure

In this chapter we report the implementation of three infrastructures that realise the two-tier run-time management described in Section 3.3.1, and the programming infrastructure for DFE designs presented in Section 3.3.2:

- **SHEPARD run-time management infrastructure (page 41).** *Scheduling for Heterogeneous Platforms using Application Resource Demands* (SHEPARD) targets OpenCL devices. SHEPARD has the ability to adapt allocation and to share out workload, thus removing fixed or static allocation of tasks in application code. When allocating tasks to resources based on the column size in the Delta Merge validation use case [33], our managed approach can adequately choose the device that yields the lowest execution time. Concurrent tasks are implicitly shared between multiple devices based on expected execution time and current device load.

- **MaxelerOS run-time management infrastructure (page 44).** One of the key features of this prototype is the use of the *groups* abstraction, which allows a cluster of dataflow engines to be managed as a single compute entity, offering a low-level elastic platform that can automatically adapt to workload.

- **Aspect-oriented dataflow design infrastructure (page 47).** This programming infrastructure supports an aspect-driven design flow for deriving optimised DFE designs. More specifically, we focus on codifying aspects that minimise development effort, exploit DFE architectural features, and explore variant implementations and run-time reconfiguration.

4.1 SHEPARD Run-Time Management Prototype

The aim of the SHEPARD prototype (Figure 4.1) is to manage application task allocation to available OpenCL devices and remove the need to statically decide at design time which devices to use. The OpenCL prototype allows programmers to annotate OpenCL functions as managed and will handle insertion of concrete kernel implementations at compile time and allocation of tasks to OpenCL devices at run time.

The prototype is composed of a number of components that together create a framework for creating and managing applications that can utilise heterogeneous resources.

4.1.1 Management database

The management database (DB) stores the characterisation of devices and implementations to enable execution management of tasks for heterogeneous devices. The information used by the OpenCL prototype consists of:

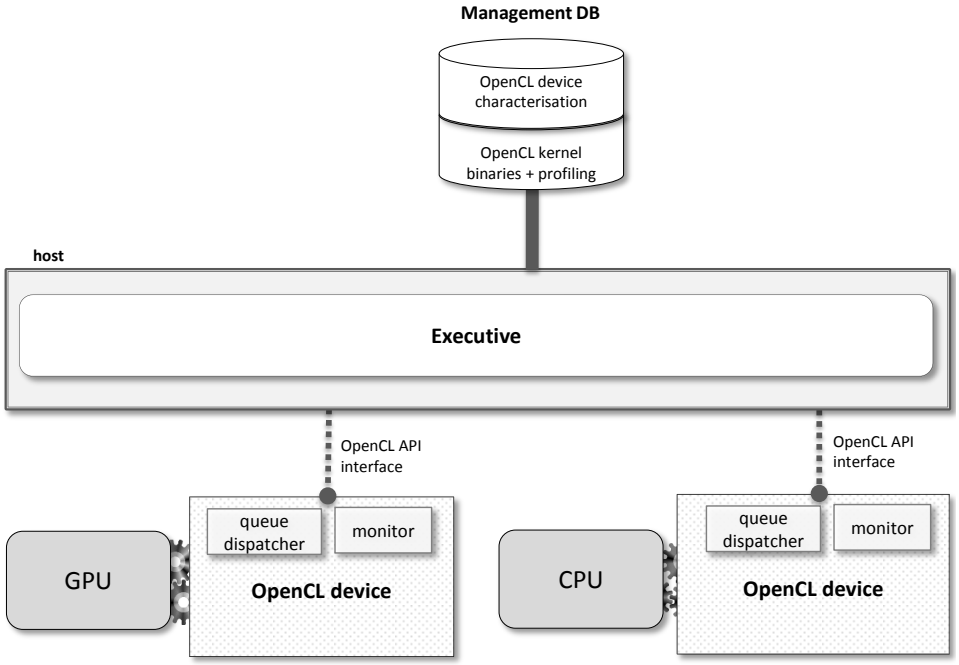


Figure 4.1: The SHEPARD prototype is an implementation of a two-tier management organisation supporting OpenCL devices.

- **Device Information.** Prior to running a managed application, a discovery tool catalogues all available devices into the management DB. This information can be used at compile time and at run time to inform managed applications of all available devices in the system. Each logged device is given a unique ID that is used to associate it with other stored items in the database such a kernel implementations and memory performance measurements.
- **Memory performance.** At the discovery stage, the memory of each device is benchmarked in terms of transfer times from the host. This is done through measurement of actual transfers of varying sized memory arrays. The results of these benchmark transfers are logged to the management DB and linear equations are created to describe the observations. This is done as an offline activity, as part of the platform setup. The transfer equations help to characterise memory transfers within a platform in terms of execution time for a given transfer size.
- **Kernel implementations.** In order to allow applications to perform tasks on OpenCL devices, implementations of the required functions, also known as *kernels* in OpenCL terminology, are required to be stored in the management DB. These implementations are then retrieved from the DB at compile time, ready to be executed at run time.
- **Kernel performance.** Two further pieces of information associated with kernel implementations are stored in the DB. The first is kernel execution time observations, which are logged from actual executions of the kernel on each supported device. When an application is run with logging enabled, the execution time of each kernel, along with associated parameters, is logged to the management

DB. This information is then used to create the second piece of information associated with a kernel, the performance estimation equations. Using the observed execution runs over a representative set of inputs, simple linear equations are created to describe the execution time of a kernel for a given input. As a performance equation is stored for each device implementation, it is possible to manually alter equations for individual kernels and devices, or to reuse equations derived for a given device on another platform if desired.

4.1.2 Task executive

Tasks are allocated to OpenCL devices at run time. The decision as to which device to use is calculated in three simple steps:

1. **Generate task time estimate.** An estimation of the task execution time is generated for each device that has been allocated to the application. The estimate is a combination of the approximate execution time for each kernel within a task, plus the approximate time to transfer memory to and from the device. These estimations are generated from the performance equations that are stored in the management DB. These estimations are then passed to a new instance of an executive object.

2. **Executive allocation decision.** To be able to adequately allocate tasks to devices, an estimation of the demand upon a device is needed to gain a true picture of the total execution time for a task. To accomplish this the executive object manages access to a shared memory space which holds queue lengths for each device in the form of a map. Each entry in the map contains the current execution time estimate for all tasks that have been assigned to a particular device. The executive object, given a set of task execution time estimates, coupled with current device queue estimate, will return to the application its decision as to which device should yield the lowest overall execution time. By delaying this allocation decision to run time can help applications avoid bottlenecks due to contention for a single device and take advantage of under-utilised resources. Additionally, given other strategies, such as energy usage or result precision, the executive can alter the behaviour and allocation decisions of applications to meet various goals.

3. **Task allocation.** Once the executive makes its decision, the application executes the implementation of the task for the chosen device.

4.1.3 Resource management

Management of devices is governed by the executive object in each application. Each object uses the shared memory map to query and update device demand as tasks are allocated to a device. Each application is given a set of resources it is permitted to use and can only allocate tasks to those devices. Each application is able to freely allocate tasks to available devices in order to match its optimisation goal. As tasks are allocated the queue length for each device is updated and informs other applications of the demand on each device. This allows the executive in each application to alter its allocation decisions to appropriately allocate tasks to meet a given strategy, such as minimise execution time or energy consumption for example.

4.1.4 Resource monitoring

Monitoring is provided through task execution logging to the database and interrogation of queue length values. Logging is optional and can be enabled when further insight into task performance is required. As logging involves the overheads of gathering and transmitting data it should only be used as a means of knowledge discovery and disabled when not required. Logging is enabled on managed tasks via an argument when creating an instance of the management library:

```
1 OCLHelper oclHelper
2         (std::string appName, bool doLogging [= false], int debugLevel [= 0] )
```

`appName` is a convenient label to identify logged object associated with an application

`doLogging` if true, collects and stores information on memory transfer and kernel execution times

`debugLevel` 0 = none, 7 = highest level of debug output to console

Data are stored in a *queue item* table with the following structure:

Column Name	Description
item_type	Type of item recorded: memory read/write/copy or kernel execution
q_id	Unique ID of the queue the item was performed on
kernel_id	Unique ID of the kernel (if item_type is kernel execution)
dev_id	Unique ID of the device used to complete the item
glb_work_size	Work size value used when executing the kernel
queued	Time in ns when the item was placed on the queue
submit	Time in ns when the item was submitted to be executed
qstart	Time in ns when the item began execution
qend	Time in ns when the item finished execution
duration	Time in ns for item to complete execution

4.2 MaxelerOS Run-Time Management Prototype

In this section we describe the MaxelerOS run-time management infrastructure. This system supports the *groups* abstraction, which allows multiple DFEs to be managed automatically according to the workload (Figure 4.2).

4.2.1 Elasticity of DFE compute resources

One challenge of the HARNESS platform is to utilise resources in a manner that will both satisfy client requests and give the application and cloud provider flexibility. As a result, the allocation of DFEs to a particular job can be made in a number of ways. The DFEs allocated to a job can either be situated on the local node that is executing the corresponding host software or can be a networked MPC-X device that has advertised its DFE content to the platform management. MaxCompiler's SLiC interface provides mechanisms to reserve and use DFEs; the mechanism used will depend on the use case and all can be used on either local DFEs or networked MPC-X DFEs.

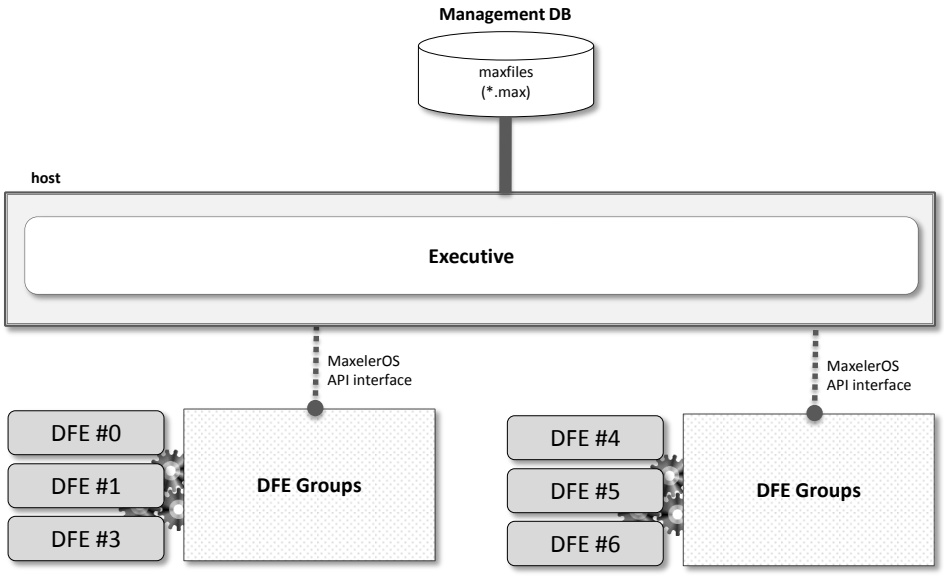


Figure 4.2: The MaxelerOS infrastructure managing DFEs using *groups*.

Individual engine allocation: DFEs on a node can be reserved individually. The engine is configured, with a *max file*, for the appropriate task when it is reserved. Tasks can then be submitted to the DFE and are run in sequence.

Group allocation: A group of DFEs is a collection of engines configured for a particular task. After configuration the engines are all identical.

By default, a group is a static size determined at creation time. The group can be a private (*EXCLUSIVE*) type, meaning that the group creator receives a handle to the group and provides the handle to any threads of execution that need to use the group. Alternatively the group can be a shared (*SHARED*) type, which will allow more processes to attach to the group. The group is identified by a string token used when creating the group, and all processes must use the same configuration.

When using the group, tasks are submitted to the group as a whole, but maybe executed on any DFE within the group. The scheduling of the tasks onto DFEs is performed using a *post office queue* algorithm, where the least utilised DFE will receive the task. Using a group removes the need for an executive to make complex scheduling decisions. With the group making scheduling decisions, the hardware optimises the pipelining of tasks to the DFEs. Also an engine can be reserved from a group and used exclusively if desired, and when finished can be returned to the group.

Dynamic group allocation: An extension to the group mechanism is the ability for the group to accumulate or shed DFEs as the utilisation of the group changes over time. A dynamic group is of type *SHARED_DYNAMIC* and is initialised to a size specified by the creator. The initial size of the group may be zero.

The utilisation of dynamic groups on the node is monitored by the node's *governor*. The governor will attach unused DFEs to the most utilised dynamic group. The governor will also take DFEs from under-utilised dynamic groups and attach them to dynamic groups with a higher utilisation. The governor will configure the DFE with the appropriate max file; this process requires no intervention from the host process.

Dynamic groups can always be shared, as it is the intention that the varying utilisation is due to the variation in the number of processes using the group. All other properties of the dynamic group are the same as the static group defined above.

Dynamic groups provide elasticity of compute resources. This is of particular benefit to the platform provider because resources are placed where they are most needed and are used when they otherwise would be idle. The infrastructure for scheduling within the group and governing group size is all performed at the MPC-X, which reduces the load on cloud management. Alternatively, if predictable throughput of tasks is required, then static groups, or individual engines, can be used.

4.2.2 Resource monitoring

MaxelerOS supports the *Maxtop* utility, which can provide per-DFE information for a node. All DFEs in a group can be recognised by the max file ID and the process ID of the user. By using this information, the run-time layers can find the instantaneous busy information for the DFE and also the load average for the entire node.

In the following example an MPC-X has eight Maia DFEs. Four of them are being used by a group and, hence, they are configured with the same max file, and have the same user, process ID, command and host. Each DFE displays how busy it is at a particular instant, and the load average for the entire MPC-X is displayed with time constants of one, five and fifteen minutes.

```
MaxTop Tool 2013.2.2
Found 8 Maxeler card(s) running MaxelerOS 2013.2.2
Card 0: Maia (P/N: 4848) S/N: 2422301010053 Mem: 48GB
Card 1: Maia (P/N: 4848) S/N: 2422301010018 Mem: 48GB
Card 2: Maia (P/N: 4848) S/N: 2422301010029 Mem: 48GB
Card 3: Maia (P/N: 4848) S/N: 2422301010030 Mem: 48GB
Card 4: Maia (P/N: 4848) S/N: 2422301010022 Mem: 48GB
Card 5: Maia (P/N: 4848) S/N: 2422301010033 Mem: 48GB
Card 6: Maia (P/N: 4848) S/N: 2422301010019 Mem: 48GB
Card 7: Maia (P/N: 4848) S/N: 2422301010059 Mem: 48GB

Load average: 0.73, 0.33, 0.16

DFE  %BUSY  MAXFILE      HOST         PID    USER       TIME
    COMMAND
 0   37.5%  fdc624cc...  host.cluster 1064   username   00:00:15  test
 1   48.9%  fdc624cc...  host.cluster 1064   username   00:00:15  test
 2   65.0%  fdc624cc...  host.cluster 1064   username   00:00:15  test
 3   66.4%  fdc624cc...  host.cluster 1064   username   00:00:15  test
 4   0.0%   IDLE (r7)    -            -      -          -         -
 5   0.0%   IDLE (r7)    -            -      -          -         -
```

```
21   6    0.0%    IDLE (r7)     -              -       -           -           -
22   7    0.0%    IDLE (r7)     -              -       -           -           -
```

Maxtop is essentially a view of the system from the perspective of the DFEs.

4.2.3 Task executive

While the groups mechanism, described in Section 4.2.1, provides elasticity of a homogeneous compute resource, the executive gives the opportunity to use another compute resource when the DFE group is fully occupied. The result is elasticity across heterogeneous resources.

The prototype implements an executive that keeps track of the number of tasks that are in progress within the DFE group. If the number of in-progress DFE tasks exceeds a threshold, then tasks are executed on the CPU until DFE tasks have been completed.

The executive assumes that the threshold is set such that the gain in execution speed by running on the DFE group is diminished because of the queue within the run-time infrastructure before the task is executed.

If all the tasks have the same execution time, the threshold N_{DFETasks} should be set such that:

$$t_{\text{CPU}} = \frac{N_{\text{DFE Tasks}}}{N_{\text{DFEs}}} \cdot t_{\text{DFE}}$$

where t_{CPU} is the time taken for the task to be executed on the CPU, N_{DFEs} is the number of DFEs in the group and t_{DFE} is the time taken for the task to be executed on the CPU.

However this executive implementation is naive because it makes several assumptions:

- That the number of DFEs in the group(N_{DFEs}) does not vary. This is true for a static group, but for a dynamic group the number varied depending on the load. This change is made by the governor and a mechanism is required to obtain the instantaneous number of DFEs in a group.
- The group is not shared. If the group is shared then the tasks submitted from the other host will be interleaved. Hence the effective queue is longer due to the other user.
- All tasks have the same execution time (t_{DFE}). It is expected that most of the time this will be the case, but if an application produces task that are different lengths then the calculation will be skewed.

4.3 Aspect-Oriented Design Flow for Dataflow Computing

This section describes the design and implementation of a hardware compilation approach targeting **dataflow engines**. This work builds on top of the LARA aspect-oriented programming approach developed in the FP7 REFLECT project[76], and presented in Section 2.2. This work is part of our initial effort to: (i) implement a generalised and customisable design flow (Section 3.3.2) that supports the automatic and systematic production of variant multi-target implementations that can be employed by cloud applications and (ii) to develop a unified heterogeneous programming model that captures multiple programming semantics using the same language and operated by the aforementioned aspect-oriented design flow (Section 3.6).

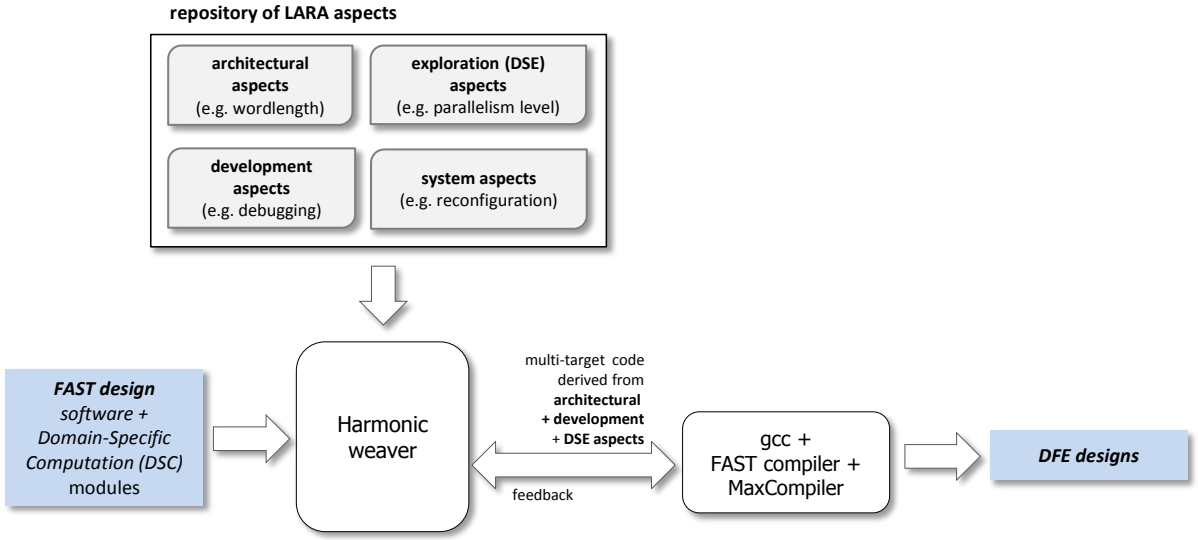


Figure 4.3: Aspect-oriented compilation tool chain, consisting of: the Harmonic source-level weaver and the FAST compiler and MaxCompiler, and a repository of aspects that address dataflow computing. Figure 3.2 illustrates the general design flow targeting heterogeneous compute resources.

We implemented an initial prototype of the uniform heterogeneous programming tools supporting FAST designs, in which we combine software C99 descriptions with variant functional modules, called DSCs. Our initial version supports two semantics: imperative (for CPUs) and dataflow (for DFEs). This resulted in the implementation of the FAST compiler, which translates C99 DSC modules using dataflow model into MaxJ code, and a set of aspects that address specific concerns of dataflow computing.

Figure 4.3 presents the implemented design flow, which operates on FAST designs and targets DFE platforms, typically consisting of an arbitrary number of CPUs and DFEs. The input of this design flow is a FAST design and an optional set of user arguments, and the output is one or more DFE designs that combine software and hardware code. The design flow is driven by an aspect repository that support different types of concerns. We report four types of aspect concerns (Table 4.1) for dataflow computing that we have successfully applied to FAST designs:

- **System aspects.** These capture transformation or optimisation strategies that affect the whole description, such as those concerning hardware/software partitioning and run-time reconfiguration capabilities. The goal of hardware/software partitioning is to improve the overall execution time by identifying parts of the code to be offloaded to hardware. Run-time reconfiguration can be used to remove idle functions from the accelerator at specific points in time, so that additional resources can be dedicated to functions that are active [65].

- **Architectural aspects.** These focus on low-level design optimisations that can be applied to designs in FAST to improve timing, resource usage or exploit specialised architectural features. For instance, operator optimisation aspects (Section 4.3.2) can be used to map operators in the program to dedicated hardware resources. Word-length aspects specify the numerical representation of variables and expressions in the design.

Aspect Type	Aspect Name	Description
system	■ hardware/software partitioning ■ reconfiguration	capture mapping between application modules and CPU/FPGA accelerators
implementation	■ operator optimisation ■ word-length spec	capture low-level hardware optimisations
exploration	■ iterative ■ meta-heuristic	generate multiple implementations based on design space exploration strategies
development	■ simulation ■ monitoring ■ debugging ■ compilation	improve developer productivity

Table 4.1: Types of aspects used in FAST.

- **Exploration aspects.** These deal with strategies that generate multiple designs to find an optimal implementation based on user requirements. Exploration aspects can act on any level of the FAST design. They enable systematic exploration of trade offs and optimisation opportunities. Examples of exploration aspects include iterative aspects (Section 4.3.3) that generate a sequence of solutions until a termination criterion is satisfied, and meta-heuristic-based aspects that find optimal solutions in a very large design space.

- **Development aspects.** These relate to concerns that have an impact on the design process, such as debugging (Section 4.3.5), and simulating kernels or improving compilation speed. Monitoring aspects instrument the application code to extract run-time behaviour and uncover opportunities for optimisation (Section 4.3.4). Separating these concerns makes the original code easier to maintain and enables the automatic application of the transformations to a wide range of designs, improving developer productivity. Simulation aspects can be applied to *dataflow* DSCs to translate them to *imperative* DSC modules, thus enabling pure software simulation. Compilation aspects, on the other hand, can be applied during the development process to create versions of the dataflow design that compile faster by reducing the operating frequency, removing debug blocks or applying design-level optimisations that can resolve timing constraints. Naturally, reducing the compilation time would increase developer productivity.

In the remainder of this section we give concrete examples of aspects of each type.

4.3.1 Reconfiguration aspect (System)

To support run-time reconfiguration, we specify the configuration associated with the function call using a FAST pragma annotation. For instance:

```
#pragma FAST map:fast_f0 cfg:c0
x = f(0);
#pragma FAST map:fast_f1 cfg:c1
```

```
y = f(x);
#pragma FAST map:fast_g cfg:c1
z = g(x);
```

With the code annotations above, our design flow can generate multiple configurations, each containing a set of FAST implementations that can be executed in parallel. If the configuration name is not specified using the FAST pragma, then we assume a default configuration.

Having a single configuration can lead to situations where at any point in time and due to data dependencies, part of the functions are idle. With run-time reconfiguration, we can exploit unused resources to support active functions. In particular, during the execution of an application, we select various configurations at different points in time to maximise the utilisation of FPGA resources. Within this context, we use a hardware partition, which contains a set of configurations that are used to support reconfiguration during the life cycle of an application. In the example above, configuration `c0` contains a single implementation of `f` (`fast_f0`), and thus can potentially use more resources and be faster than the `fast_f1` version which must share the same configuration (`c1`) with `fast_g`.

We have proposed an approach for extracting valid and efficient hardware partitions [65]. To realise run-time reconfiguration without modifying the original code we use the aspect shown in Figure 4.4. The input to the aspect is a hardware partition (lines 2–4). The partition is implemented as a hash table that maps a function call (`key`) to a hardware implementation, represented as a tuple containing the hardware implementation name (`hw`) and the associated configuration (`cfg`).

```
1  aspectdef AspReconfig
2  input
3     partition
4  end
5  select function.call end
6  apply
7     if ($call.key in partition) {
8        var cfg = partition[$call.key].cfg;
9        var map = partition[$call.key].map;
10       $call.insert before %{
11          #pragma FAST map:[[map]] cfg:[[cfg]]
12       }%;
13    }
14 end
15 end
```

Figure 4.4: A reconfiguration aspect.

Figure 4.5 shows an example of a hash table representing a hardware partition. The key (e.g. `main:f:1`) identifies a function call in the application, and is formed by concatenating the name of the caller function (`main`), the name of the invoked function (`f`) and a unique number (`1`). Line 5 in the aspect shown in Figure 4.4 selects all function calls and, for each call found in the input partition (line 7), we set the appropriate pragma on top of the call statement (lines 10–12). We can now realise and experiment with different reconfiguration designs by invoking this aspect with different hardware partitions.

partition		
$call.key	**map**	**cfg**
main:f:1	fast_f0	c0
main:f:2	fast_f1	c1
main:g:3	fast_g	c1

Figure 4.5: An example of a hardware partition, represented as a hash table, used as argument to the reconfiguration aspect (Figure 4.4).

4.3.2 Operator optimisation aspect (Architectural)

To provide architectural details to FAST designs, such as mapping operators to DSP blocks, we can use the FAST pragma shown in Figure 4.6 on top of a statement (including code blocks). The balancing parameter corresponds to the degree of utilisation of DSP blocks (dedicated hardware resources in an FPGA) in a statement, as in MaxCompiler.

```
1 #pragma FAST DspFactor:0.5
2 {
3   x = x * y;
4   x++;
5 }
```

Figure 4.6: The FAST balancing pragma provides fine-grained control over the mapping of computations to either DSPs or LUT/flip-flop pairs.

The aspect shown in Figure 4.7 captures a strategy for balancing DSP blocks in every statement of an application. Instead of adding the pragma above manually, we provide a set of rules (lines 3–5) that define where to place the `balanceDSP` pragma. In this example, we established the rule that full DSP block utilisation is applied to any statement that has five or more multipliers and adders, balanced if three or more multipliers, and no DSP utilisation otherwise.

4.3.3 Iterative aspect (Exploration)

With LARA we can implement and combine aspects to enable systematic design-space exploration of all the optimisation options exposed by the FAST back end, resulting in the generation of a large number of designs. The feedback-directed compilation process of LARA can be used to capture and extract feedback from the back-end reports pertaining to resource usage or timing information, and automatically adjust the compilation process.

An example of a LARA aspect for design-space exploration is shown in Figure 4.8. It highlights the feedback capabilities of the design flow: the aspect will generate and build the FAST designs until the resource usage passes a specified LUT threshold, and at each step increasing a particular design attribute, such as exponent, mantissa or the parallelism of the design by replicating the computational pipeline.

```
1  aspectdef DspBalancing
2  input
3     op_granularity =
4       [{DspFactor: 1,MultiplyOp: 5,AddOp: 5 },
5       {DspFactor:0.5,MultiplyOp:3}];
6  end
7  select function.statement end
8  apply
9     for (var i in op_granularity) {
10       var gprofile = op_granularity[i];
11       var match = true;
12       for (var k in gprofile) {
13          if (k != 'DspFactor') {
14             match &= ($statement.num_construct(k)
15                       >= gprofile[k]);}}
16       if (match) {
17          var pragma = '#pragma␣FAST␣:'
18                       + gprofile.DspFactor;
19          $statement.insert before "[[pragma]]";
20          break;}}
21    end
22 end
```

Figure 4.7: An operator optimisation aspect for exploring mapping of computation to DSP blocks.

```
1   aspectdef DesignExploration
2   input
3      attribute,
4      start, step,
5      lut_threshold,
6      config
7   end
8   config[attribute] = start;
9   var i = 0;
10  do {
11    var designName = genName(config);
12    call genFAST(designName, config);
13    buildFAST(designName);
14    config[attribute] += step; i++;
15  } while (@hw[designName].lut < lut_threshold
16           && i < LIMIT);
17  end
```

Figure 4.8: An exploration aspect that generates multiple FAST designs by varying a design attribute until a LUT threshold is reached.

4.3.4 Monitoring aspect (Development)

To find potential hot spots in an application and provide compensation for them (e.g. to perform hardware/software partitioning), we can use the aspect in Figure 4.9. In this example, the weaver can

automatically instrument any C application to self-monitor its innermost loops at run time, as they are natural candidates for dataflow-based acceleration. In particular, this monitoring aspect can compute the following information for every innermost loop:

1. the average number of times it has been executed;
2. the average number of iterations;
3. the loop average time; and
4. the loop iteration average time.

For this purpose, we use a monitoring API composed of four functions to mark the beginning and end of the loop (`monitor_instanceI` and `monitor_instanceE`, respectively), and to mark the beginning and end of an iteration (`monitor_iterI` and `monitor_iterE`, respectively). These monitoring functions keep an account of the frequency of execution and the time to complete the whole loop and a single iteration.

```
1  aspectdef LoopMonitor
2  select function.loop{is_innermost} end
3  apply
4      $loop.insert before
5         %{monitor_instanceI("[[$loop.key]]");}%;
6      $loop.insert after
7         %{monitor_instanceE("[[$loop.key]]");}%;
8  end
9
10 select function.loop{is_innermost}.entry end
11 apply $begin.insert after
12        %{monitor_iterI("[[$loop.key]]");}%;
13 end
14 select function.loop{is_innermost}.exit end
15 apply $begin.insert before
16        %{monitor_iterE("[[$loop.key]]");}%;
17 end
18 end
```

Figure 4.9: A monitoring aspect that instruments the application to reveal loop activity. The information generated can be used to identify hot spots.

The aspect code is as follows: line 2 selects all loops in the application that are innermost (loops with no other loops enclosed); lines 3–8 place an instance monitor call before and after each selected loop; lines 10–13 select all entry points inside the loop and insert a monitoring call to mark the beginning of each iteration; and lines 14–17 place an instance monitor call to mark the end of each iteration. Figure 4.10 shows an example of automatically applying the aspect from Figure 4.9 on a C-style function containing a loop.

Each monitoring call in the woven code receives as a parameter, the *loop key*, that uniquely identifies the loop within the application. The loop key is generated by concatenating the function name with the

original code	woven code

void f() {
 	
  while (i < N) {
 	
  i++;
 	
  }
 	
}
	void f() {
	  monitor_instanceI("f:1");
	  while (i < N) {
	  monitor_iterI("f:1");
	  i++;
	  monitor_iterE("f:1");
	  }
	  monitor_instanceE("f:1");
	}

Figure 4.10: Original and woven codes after applying the aspect shown in Figure 4.9.

hierarchical position of the loop within the abstract syntax tree. For instance, `f:2:1` corresponds to the first loop inside the second outermost loop of function `f`.

The profiling information generated by running the woven code can be fed to a hardware/software partitioning aspect (not shown) that determines which loop or function is most profitable to be offloaded to hardware.

4.3.5 Debugging aspect (Development)

Because the current DFE execution model does not provide run-time debugging of hardware designs, the easiest solution to debug designs is to log the values of various streams during execution. The insertion of debug statements can be encapsulated in aspects. It is particularly important to separate debug aspects from the original application code, since debug blocks can influence the compilation time and timing constraints as well as the behaviour of the design.

As an example, the aspect in Figure 4.11 instruments the code to log every change in scalar float variables, logging the before and the after values. In particular, the aspect finds all variable references in the code (line 2), and filters out all variable references that are not defined (line 9). For the remaining pool of variables, we instrument the corresponding statements by placing a `log` function before and after the statement. This aspect is generic, and can be applied to any C code to log changes in scalar float variables.

```
1  aspectdef WatchVar
2  select function.vref end
3  apply
4     $vref.parent_stmt.insert before
5      %{ log("[[$vref.name]]", [[$vref.name]]); }%
6     $vref.parent_stmt.insert after
7      %{ log("[[$vref.name]]", [[$vref.name]]); }%
8  end
9  condition $vref.is_out && $vref.is_scalar && $vref.is_float end
10 end
```

Figure 4.11: A development aspect for automatically instrumenting the code to watch any change in the value of a program variable.

5 HARNESS Use Cases

5.1 Overview

In this chapter, we report the research experiments performed in WP3 during Year 1, addressing general and application requirements specified in deliverables D2.1 [31] and D2.2 [32]. These experiments deal with programming and managing a wide range of heterogeneous devices such as CPUs, DFEs and GPGPUs across all three validation use cases [33]. They already show promising results that we expect to exploit in the context of the integrated HARNESS platform (Chapter 6) in Year 2.

Section	Demonstrator	General Requirements (D2.1)	Industrial Requirements (D2.2)
5.2.1 (page 56)	**RTM**	**R10, R11**	**R27**
Topic: Aspect-oriented design for dataflow computing			

In this section we evaluate the effectiveness of our aspect-oriented design flow in dataflow computing using the HARNESS RTM use case. We cover specific concerns related to dataflow design, including minimising development effort, exploiting architectural features, exploring variant implementations and run-time reconfiguration. This work has been published [42] (nominated for best paper award) and was the topic of an MEng thesis at Imperial College London (awarded the ARM prize for best project in computer systems by the Department of Computing in August 2013).

5.2.2 (page 60)	**RTM**	**R11, R12, R14**	**R27**
Topic: Dynamic stencil computations on reconfigurable (DFE-based) clusters			

In this section we present the *dynamic stencil* approach, which optimises stencil applications by constructing scalable designs that can adapt to available run-time resources in a reconfigurable cluster. Experimental results with RTM show that high throughput and significant resource utilisation can be achieved, which can dynamically scale into reconfigurable computing nodes as they become available during their execution. This work has been published [67].

5.3.3 (page 76)	**Delta Merge**	**R11, R12, R14**	**R31, R32, R33**
Topic: Managing Column-Based Merges on Heterogeneous Platforms			

In this section we report the work on accelerating the delta merge operation by exploiting heterogeneous processing elements such as CPUs and GPGPUs. The delta merge operation is an important HARNESS validation use case as it can have a large impact on the performance of a SAP HANA system. The results of this work include characterising the tasks involving the delta merge operation, and demonstrating how this task characterisation can be used to: (i) make run-time allocation decisions about which heterogeneous devices to use and (ii) performing load balancing using all available processing elements. This work has been submitted for publication under the title: *"SHEPARD: Scheduling on Heterogeneous Platforms using Application Resource Demands"*.

5.4 (page 81)	**AdPredictor**	**R11, R14**	**R35**
Topic: Accelerating AdPredictor's training process on reconfigurable platforms			

In this section we describe the work on mapping the AdPredictor training process onto a DFE and comparing this design against an optimised single-threaded and a multithreaded CPU version. AdPredictor is an open source HARNESS validation use case that uses a Bayesian machine-learning system to provide advertisement recommendations.

5.2 Reverse Time Migration (RTM)

RTM is a high-performance HARNESS industrial validation use case for seismic imaging, in which geological structures are detected based on the Earth's response to injected acoustic waves. The technique models the propagation of injected waves using isotropic acoustic wave equation [6], which is solved with a fifth-order stencil in space with three dimensions:

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

This application is described in detail in Chapter 2 of Deliverable D2.2 [32].

In Section 5.2.1 we use this application to illustrate how our aspect-oriented design flow for dataflow engines can support specific concerns, such as minimising development effort, exploit architectural features, explore variant implementations and exploit run-time reconfiguration. In Section 5.2.2, we cover a design and run-time method for accelerating scalable stencil computations with reconfigurable hardware that can adapt dynamically to different resource allocations during the life cycle of the application.

5.2.1 Aspect-oriented design flow for dataflow engines

We use FAST to implement the complete RTM application. The software layer (Section 3.6) is augmented with four variant DSC modules: two modules that are used to control the memory command read and write streams (`CmdRead`, `CmdWrite`), and two versions of the RTM computation kernel referred to as RTM Static and RTM RTR. In the static version we fit all the RTM hardware functionality into the same DFE configuration, while in the run-time reconfiguration version we use multiple DFE configuration so that each configuration optimises active functions using resources that would otherwise be occupied by idle functions in a static version.

Development effort

To illustrate the potential benefits of our aspect-oriented approach, we first analyse the effects on development effort. Table 5.1 compares the number of lines of code of four FAST kernels against the corresponding MaxCompiler code. The FAST kernels have been automatically instrumented using the **debugging aspect** presented in Section 4.3.5 to automatically monitor floating-point variables in the application. The MaxCompiler kernels, on the other hand, were manually instrumented for the same purpose. It can be seen, for instance, without considering the instrumentation effort, that the FAST run-time reconfigurable design is 42% smaller than the corresponding MaxCompiler code while using 67% fewer API calls, which translates to increased productivity. Although the FAST dataflow model is largely based on features of MaxCompiler, the discrepancy in number of lines and API calls can be explained by the fact that MaxCompiler models dataflow computations through a set of Java libraries, while FAST describes dataflow computations using language constructs. Extensions to FAST to support dataflow computation include adding a limited number of built-in functions that work as language operators (similar to `sizeof`) and pragma annotations.

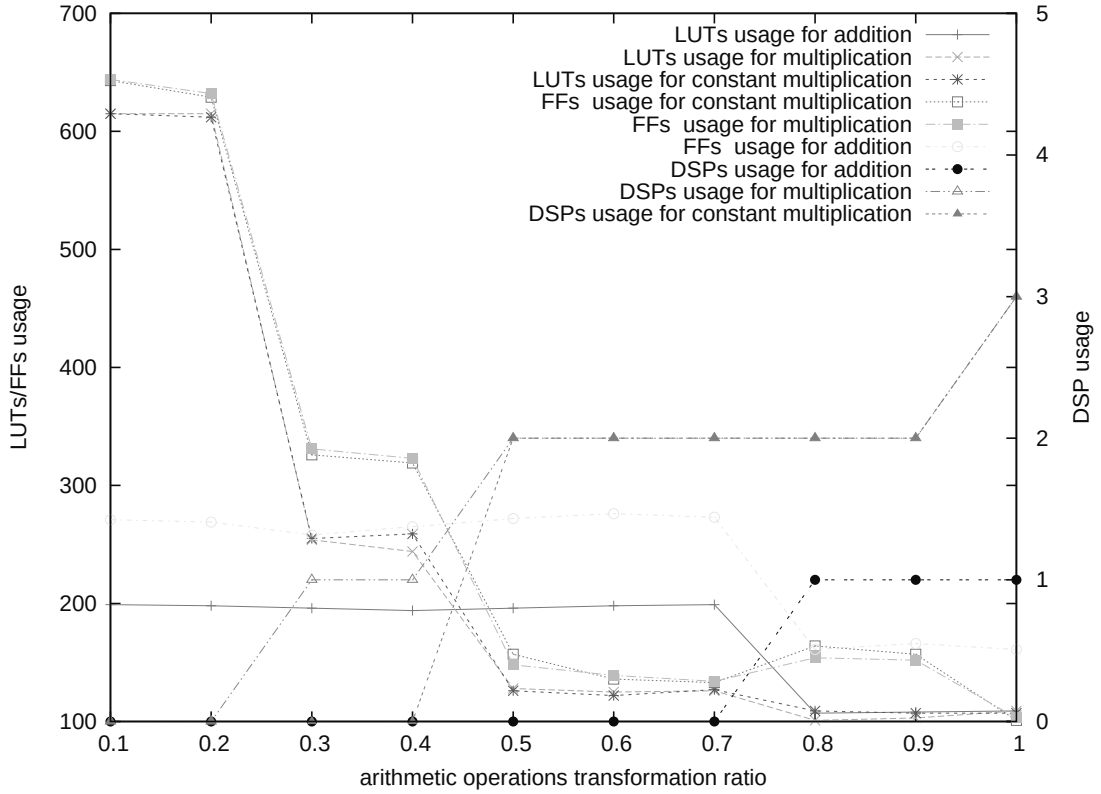


Figure 5.1: Exploration of DSP and LUT/flip-flop balancing for functional units implemented using the *operator optimisation aspect* presented in Figure 4.7.

Kernel	Aspect	FAST		MaxCompiler	
	LOC	LOC	# API calls	LOC	#API Calls
CmdRead	12	26	6	59	39
CmdWrite	12	28	39	79	56
RTM Static	12	246	43	403	175
RTM RTR	12	377	91	669	275

Table 5.1: Code measures for the RTM kernels comparing FAST and MaxCompiler.

Architectural features

The **operator optimisation aspect**, shown in Figure 4.7, captures an empirical strategy that automatically instruments the code to define the level of DSP utilisation on code statements, without having developers manually instrument the code. By specifying different arithmetic operation thresholds and DSP factors as arguments to the aspect description, we derived different designs, as shown in Figure 5.1, that exhibit different levels of DSP, LUT and flip-flop utilisation. It can be seen that as we increase the DSP factor (x-axis) for specific operations, we also increase the DSP utilisation and reduce LUT/flip-flop usage, thus providing more resource space for other computations. This aspect can be combined with an **exploration aspect** to find designs that maximise DSP utilisation without over-mapping in arithmetic-intensive applications.

Exploration of variant implementations

Results of the design-space **exploration aspect**, presented in Figure 4.8, with the RTM Static kernel allow us to explore the effects of varying mantissa with resource area (Figure 5.2). We observe irregular, large variations when decreasing the mantissa from 18 to 16 and 24 to 22, which is the effect of the back-end tools mapping arithmetic to a combination of both DSPs and LUT/flip-flop elements. The mantissa boundaries at which this optimisation occurs are platform specific, depending on the architecture of the DSPs. Hence, automating this optimisation via aspects and decoupling it from the original source code makes the application more portable and facilitates discovery of interesting trade-off opportunities using design space exploration.

We also use the same aspect to increase the parallelism level to investigate design scalability. For example, for the described RTM implementation, Figure 5.3 shows that performance scales linearly with the number of parallel pipelines and that significant speedups can be obtained by the FAST dataflow design compared to the CPU-only implementation. Depending on the problem size, our approach can be used to achieve a significant speedup over software only versions which is comparable with the best published FPGA results for static designs [6, 65].

Reconfiguration

Figure 5.3 also shows a model of the performance benefits of using a run-time reconfigurable implementation generated using the proposed aspect-oriented approach. Two configurations were created for the RTM FAST kernel. Since, in our model, during the first half of the execution time, the backward propagation and imaging functions are idle, the first configuration requires only half the resources. Hence,

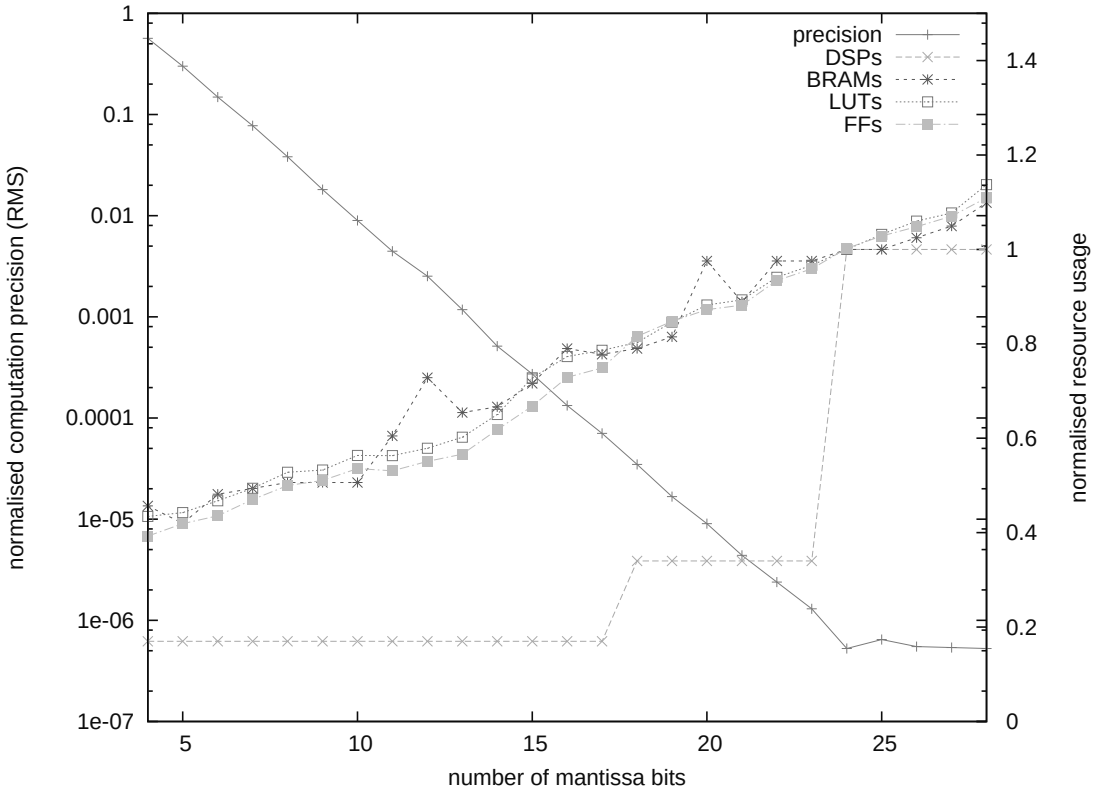


Figure 5.2: Automatically generated RTM designs that show a trade off between accuracy and resource usage using the *exploration aspect* presented in Figure 4.8 with variable mantissa.

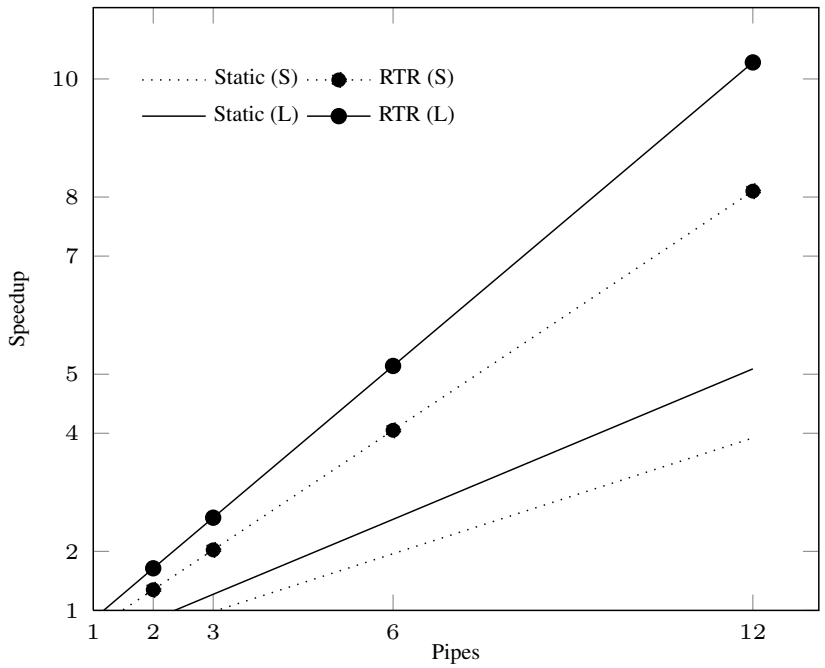


Figure 5.3: Scalability of the RTM dataflow design explored using the aspect shown in Figure 4.8.

the number of parallel pipelines can be doubled, halving the execution time of the first configuration. The speedup obtained is comparable [65], but the partitioning and optimisation exploration process is automated via **system aspects** (Section 4.3.1), which increases developer productivity. The automated process also improves portability of the design, allowing optimisations based on design-space exploration to be carried out on various platforms and, hence, subject to varying resource constraints without manual intervention.

5.2.2 Mapping dynamic stencil computations onto DFEs

We now explore the problem of mapping stencil computations efficiently on a reconfigurable cluster, to allow computationally intensive applications, such as RTM, to dynamically adapt to given resources at run time. A reconfigurable cluster is a system with a CPU acting as host and an arbitrary number of compute nodes (FPGAs/DFEs) acting as accelerators.

Sharing resources in a cluster where applications can be launched adds complexity to 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. In a reconfigurable cluster with nodes consisting of different FPGAs, the heterogeneous FPGA nodes are used and released by various computational tasks at different points in time. More specifically, for a given design, throughput can be potentially increased if more resources are available to perform its computation. However, the effectiveness of current static design methods is limited by unpredictable run-time conditions. Due to nondeterministic starting points of

applications, node availability and the amount of computational resources in available nodes are unknown during compile time.

The basic idea of our work is illustrated with the following example (Figure 5.4). In a reconfigurable cluster, four 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 one resource unit and process one data unit per second, while node C can process two data units per second. An application with eight data units to process is then launched into the cluster. Linear scalability is assumed for executed tasks, i.e., execution time is halved if the number of utilised resource units doubles. In this scenario, two static designs are illustrated in Figure 5.4. The OneNode Design will make use of only one node, so would take eight seconds to complete. The FourNode Design will take all four nodes when all of them become available at time 4, and would take two seconds to complete. Only half of the computational capacity in node C is utilised, as the FourNode Design predefines that one resource unit is used in each run-time 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 units, node B becomes available too, so both nodes process another two data units in the next second. At time 3 node A, B and C are available, completing the processing of the four remaining data units.

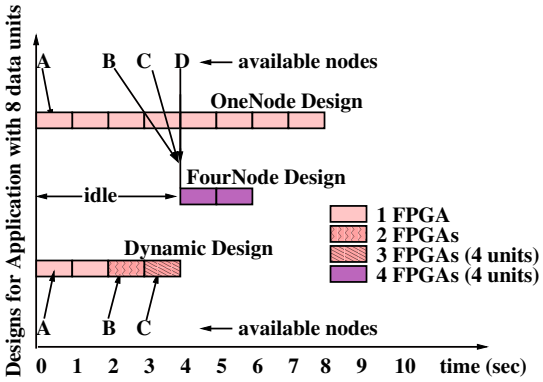


Figure 5.4: Execution of various designs in the cluster when nodes A, B, C, and D are released. The execution time of three designs (OneNode, FourNode and Dynamic) for the same application is shown.

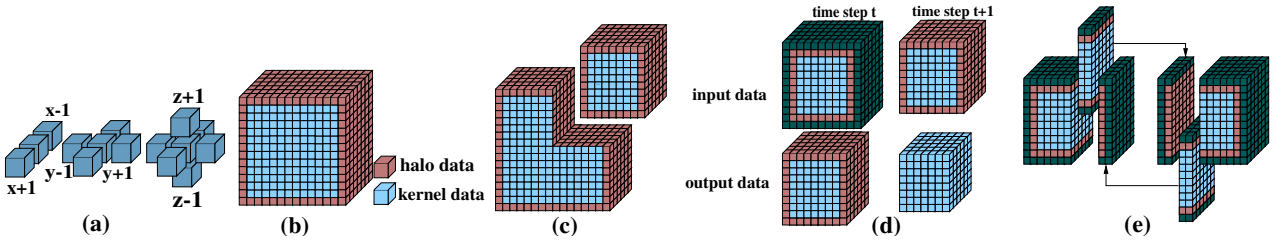


Figure 5.5: (a) 1D, 2D and 3D stencil in space; (b) halo data within space; (c) stencil data after spatial blocking [75]; (d) input and output results for time steps t and $t+1$ [64]; and (e) data exchange between neighbouring blocks [78].

In a reconfigurable cluster with various computing nodes, challenges for developing such dynamic designs include: (i) generating optimised designs that exploit intra-node resources in the cluster;

(ii) constructing initial designs when applications are mapped into the cluster, which ensures scalable performance for utilised nodes and correct functionality for the applications; and (iii) adapting designs to run-time resource variations.

For parallel applications without inter-processor communication, such as Monte-Carlo simulation, the run-time solution is easy to generate as there is no communication operation. For communication-intensive applications, the challenges described above become tightly coupled: communication is cycle-accurately scheduled to overlap with computation operations; variations in device processing capacity, device-level parallelism and distributed workload bring deviation in the scheduled timing; and if design configurations are not properly updated, incorrect results can be generated after the dynamic design scales into new nodes. In this work, stencil computation, known to be communication intensive and difficult to parallelise, is used as a case study for the proposed approach. Contributions of this work include:

- A novel design approach, *dynamic stencil*, that exploits various intra-node resources with compile-time optimisation, and utilises run-time resources with run-time initialisation and scaling. These three design stages cooperate with each other to ensure high resource utilisation for stencil applications in reconfigurable clusters.
- An asynchronous communication model that schedules communication operations to eliminate communication overhead based on: intra-node computational capacity, inter-node communication bandwidth, and available nodes. The communication model can be dynamically updated to ensure linear scalability as well as correct functionality when a Dynamic Stencil design scales.
- A run-time performance model that provides rapid evaluation of benefits and overhead for scaling current dynamic stencil designs into FPGAs provisioned during run time.

Stencil computation

Stencil computation refers to a class of iterative operations to update array data with a fixed pattern, called 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 equation*s (PDEs). As an example, to capture dynamic properties within target systems, a PDE can be formulated as follows:

$$A\frac{\partial^2 f(s,t)}{\partial t^2} = B\frac{\partial^2 f(s,t)}{\partial s^2} + C\frac{\partial f(s,t)}{\partial s}$$

where A, B and C are PDE parameters, and $f(s,t)$ denotes simulated properties at space s and time t. Finite difference is a numerical method to approximate derivative expressions. In this example, the target space s includes three dimensions x, y, z and derivatives are replaced with first-order finite difference expressions.

The system status can be propagated as shown in Figure 5.6. α, β and γ are constant coefficients calculated by the finite-difference method. The corresponding 3D stencil is shown in Figure 5.5a. As neighbouring data are required to support the calculation, as shown in Figure 5.5b, boundary data are not updated during computation; these are called *halo data*. In each time step t, the constructed stencil sweeps over kernel data to propagate $f(s,t)$ in the time dimension. The number of arithmetic operations in Figure 5.6 can be calculated as $nt \cdot nz \cdot ny \cdot nx \cdot N_{ar}$, where N_{ar} is the number of arithmetic operations for each data point.

```
1:  for t ∈ 0 → nt do
2:      for z ∈ 1 → nz − 1 do
3:          for y ∈ 1 → ny − 1 do
4:              for x ∈ 1 → nx − 1 do
5:                  f[t+1][z][y][x] = (f[t][z][y][x−1] + f[t][z][y][x+1]) * α
6:                                  + (f[t][z][y−1][x] + f[t][z][y+1][x]) * β
7:                                  + (f[t][z−1][y][x] + f[t][z+1][y][x]) * γ
8:                                  - f[t−1][z][y][x];
9:              end for
10:         end for
11:     end for
12: end for
```

Figure 5.6: An example of a stencil code pattern supported by dynamic stencil.

As neighbouring data at multiple dimensions are required for each computation, spatial locality reduces as the dimension size and the number of dimensions increase. If the dimension size is 1024, data accessed in the stencil in Figure 5.6 span 8MB of data. Limited by the sparse data access patterns, performance of stencil computations is limited to 1.8 Gflops [68] on a four-core Intel i7-870 CPU for a fifth-order stencil. Propagating the stencil for 1000 time steps in $1024 \times 1024 \times 1024$ space requires 63.4 Tera floating-point operations, and takes 10 hours to complete. The high-performance requirements limit the usage of stencil computations in scientific research and industrial development.

Parallelism in stencil algorithms can be exploited with optimisation techniques such as design parallelisation, spatial blocking (loop tiling, domain decomposition), temporal blocking and communication scheduling. In general-purpose processors such as CPUs and GPGPUs, stencil designs are parallelised through spatial blocking, which refers to dividing involved data into multiple blocks to improve temporal data locality. As shown in Figure 5.5c, for a 3D 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 5.5d, 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. Communication operations, as shown in Figure 5.5e, are required to exchange halo data between neighbouring devices when stencil applications are mapped into multi-device clusters. If not properly scheduled, the communication overhead increases with the number of involved devices, which severely limits design scalability.

Methodology

Dynamic stencil starts with a C language description for stencil computations, as shown in Figure 5.6, and ends up with a reconfigurable design that can adapt to available resources at run time. The development process of a dynamic stencil design is demonstrated in Figure 5.7, which includes three steps: compile-time optimisation, run-time initialisation and run-time scaling.

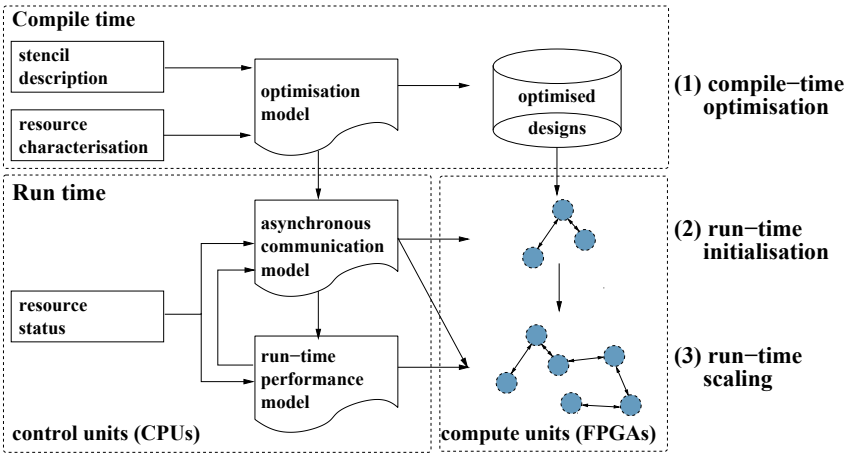


Figure 5.7: The dynamic stencil design approach has three steps: compile-time optimisation, run-time initialisation and run-time scaling.

variables		parameters	
optimisation model			
par_n	design parallelism	A	available resources
sk_i	spatial blocking ratio	BW	available bandwidth
tk	temporal blocking ratio	w_i	stencil size
sl	size of a data slice	D	stencil dimension
communication model			
dc_n	computation delay	F	number of FPGAs
$dm_{n,j}$	memory delay	par_n	design parallelism
$tr_{n,j}$	arrival time for halo data	sk_i,tk	blocking ratios
		dw_n	distributed workload
		w_i	stencil size
performance model			
rtb	run-time benefit	F	number of FPGAs
rov	reconfiguration overhead	par_n	design parallelism
		sk_i,tk	blocking ratios

Table 5.2: Variables and selected parameters in the dynamic stencil approach (indices: i=dimension, n=node, j={front,end})

The *compile-time optimisation* phase first translates a stencil kernel described in C into a data-flow graph. This data-flow graph captures all the kernel operators, the operator dependencies and memory access patterns. This intermediate kernel representation is used with the optimisation model to generate a stream-based architecture supporting multiple inter-connected FPGAs to form a dynamic stencil. Design parallelisation, spatial blocking and temporal blocking are integrated into our optimisation model to evaluate their impact on the optimised architectures. Differences in FPGA nodes are expressed as variations of available resources. Bounded by available resources, the basic hardware architecture is automatically optimised to achieve maximum throughput, and then synthesised with back-end vendor tools to generate executable bit streams.

Run-time initialisation refers to constructing the initial dynamic stencil design when the application is launched into clusters. Interconnections between FPGAs are required to support data exchange with neighbouring devices, as shown in Figure 5.5e. A topology where FPGA nodes are chained with point-to-point connections or connected to the CPU host is called an FPGA *path*. Synthesised designs for various nodes are loaded and linked to occupy the longest FPGA path among available FPGA nodes. An asynchronous communication model is developed based on properties of mapped designs, resource status and communication bandwidth. Communication operations among utilised nodes are scheduled to run in parallel with communication operations, with data dependency expressed as timing constraints in the model.

Run-time scaling is triggered if a dynamic stencil design finds available nodes to expand. Benefits and overhead for expanding a current design into the new nodes are evaluated with a run-time performance model. Once the benefits outweigh the overhead, a run-time scaling algorithm is executed to reconfigure new nodes, switch context into the scaled dynamic stencil design, and dynamically update design configurations to ensure correct functionality for the scaled dynamic stencil design.

Variables and parameters for a dynamic stencil design are presented in Table 5.2. Variables for the optimisation model are used as parameters in the communication model and the performance model.

Compile-time optimisation

The basic streaming architecture for stencil computations is shown in Figure 5.8a. A pipelined data path is mapped from arithmetic operations in stencil descriptions, and a memory architecture is built based on the extracted data access pattern (stencil shape). At each clock cycle, the stencil moves one step forward in the fastest (x) dimension, one data unit is streamed from off-chip memory, and $\sum_{i=0}^{D} w_i \cdot 2 + 1$ stencil data units are loaded from the on-chip memory, where w_i indicates the number of data units in a stencil at dimension i. In the example in Figure 5.8a, $w_z = 1$.

Design parallelisation variable par indicates the number of replicated data paths. For a streaming architecture with $par = 4$, as shown in Figure 5.8b, the replicated stencil moves four steps forward in the x dimension at each clock cycle. The same memory architecture is used to share accessed data, while four data paths are replicated to process data in parallel. Resources consumed by a pipelined data path can be estimated by accumulating resources consumed by each arithmetic operator, with $R_{op \in \odot = \{+,-,\bullet,\div\}}$ and S_{op} indicating resource utilisation for operator *op* and the number of operator *op* in the stencil description, respectively. Theoretically, n valid results can be generated per clock cycle for a streaming architecture configured as $par = n$ if the design can be accommodated in on-chip resources and if memory bandwidth permits.

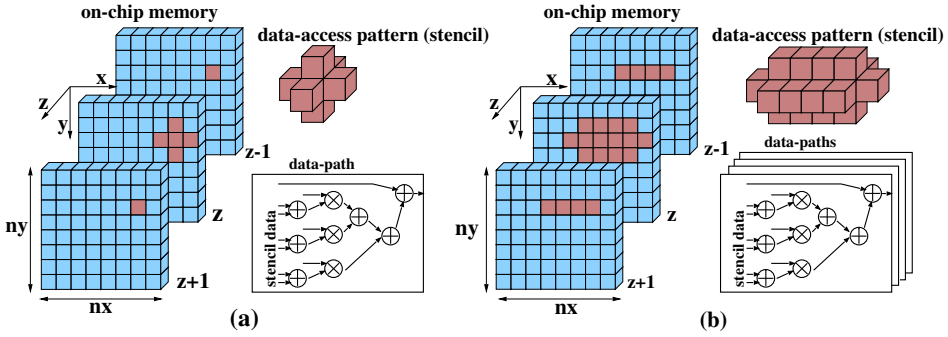


Figure 5.8: Data access patterns, memory architectures and data paths in streaming architectures for stencil computations with (a) a single data path (par = 1) and (b) four replicated data paths (par = 4).

Spatial blocking is applied to reduce memory resource consumption. 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 5.8). When the dimension size increases, memory resource consumption can easily exceed resource constraints. Blocking dimensions in memory slices regroups streaming patterns in the blocked dimensions, which effectively reduces slice size and memory resource consumption. As shown in Figure 5.5c, to protect data dependency of boundary data after blocking, one layer of halo data is distributed to blocked data. In a dimension i with n_i kernel data and blocking ratio sk_i, the size of blocked dimension can be expressed as $\frac{n_i}{sk_i} + 2 \cdot w_i$. Since halo data are distributed to each data block, spatial blocking increases the overall data size compared with unblocked designs.

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 loading and writing par data units per clock cycle. As shown in Figure 5.5d, 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, eliminating redundant memory access and 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 halo data of intermediate results can be properly updated without synchronising with neighbouring blocks. Therefore, the size of blocked dimension i with spatial blocking ratio sk_i and temporal blocking factor tk can be expressed as $\frac{n_i}{sk_i} + 2 \cdot w_i \cdot tk$, where tk layers of halo data are represented as $2 \cdot w_i \cdot tk$. The size of one slice data after spatial and temporal blocking is:

$$sl = \prod_{i=1}^{D-1} \left(\frac{n_i}{sk_i} + 2 \cdot w_i \cdot tk \right) \tag{5.1}$$

An optimisation model is built to configure design parallelism par, spatial blocking ratio sk and temporal blocking ratio tk to achieve minimum execution time, i.e., the ratio between overall data size and computational capacity. Overall data size is expressed as $nsteps \cdot sl \cdot \prod_{i=1}^{D-1} sk_i \cdot n_D$, where $nsteps$ is the number of time steps, $\prod_{i=1}^{D-1} sk_i$ indicates the number of data blocks, and n_D is the size of the slowest dimension corresponding to the number of data slices (n_z in Figure 5.8).

Computational capacity $par \cdot tk$ increases with design parallelism and temporal blocking ratio, while slice size and number of data blocks increase with temporal and spatial blocking ratios. Bounded by available on-chip resource A and off-chip bandwidth BW, the model is expressed as:

$$\textbf{minimise}: \frac{nsteps \cdot sl \cdot (\prod_{i=1}^{D-1} sk_i) \cdot n_D}{par \cdot tk} \tag{5.2}$$

$$\textbf{subject to}:$$

$$\sum_{op \in \odot} S_{op} \cdot R_{op} \cdot par \cdot tk + I_{LT/FF/DP} \leq A_{LT/FF/DP} \tag{5.3}$$

$$ws \cdot tk \cdot sl \cdot 2 \cdot w_D + I_{BR} \leq A_M \tag{5.4}$$

$$par \cdot ws \cdot \max(IO_{in}, IO_{out}) \cdot fq \leq BW \tag{5.5}$$

Infrastructure resource consumption $I_{LT/FF/DP/BR}$ indicates the LUTs, flip-flops, DSP and *block random-access memory* (BRAM) resources consumed by communication infrastructures. Equation 5.3 expresses the data-path resource consumption, which increases linearly with par and tk. The memory resource consumption is estimated in equation 5.4, with tk on-chip memories implemented, and $2w_D$ data slices stored in each on-chip memory. Design parallelism par does not affect memory resource consumption, as the replicated data paths share the same memory architecture. ws is the width of each data unit (for single floating-point stencils: ws=32-bits). The impact of temporal blocking in memory bandwidth is expressed in equation 5.5, where an increase in tk does not contribute to memory requirements. IO indicates the number of input/output arrays, and fq is the operating frequency.

Run-time initialisation

When a stencil application is mapped into a reconfigurable cluster, optimised designs for available FPGA nodes are loaded and connected to form an initial dynamic stencil design. During available FPGA nodes, the longest FPGA path is detected to accommodate the initial dynamic stencil design. Data distribution for a dynamic stencil design is shown in Figure 5.9a, with halo data at the front region and the end region of the allocated data, respectively, called *front halo data* and *end halo data*. Design properties of an occupied FPGA node n are abstracted with its computational capacity par_n, i.e., the number of data units processed per clock cycle. For F available FPGA nodes, the slowest dimension with size n_D is decomposed to balance the distributed workload dw_n based on the computational capacity of involved nodes:

$$dw_n = (\prod_{i=1}^{D-1} sk_i) \cdot sl \cdot (\frac{n_D}{\sum_{k=1}^{F} par_k} \cdot par_n + 2 \cdot w_D) \tag{5.6}$$

As shown in Figure 5.9b, the decomposed data are processed simultaneously in the three involved FPGAs. In an FPGA node, the distributed workload are processed with par data per clock cycle, propagating a time step with $\frac{dw_n}{par_n}$ cycles. Results for kernel data are transmitted to neighbouring nodes to update the corresponding halo data used in next time step.

Communication operations between involved FPGAs are performed in parallel with the computation operations to eliminate communication overhead. To satisfy data dependencies in a stencil computation, halo data from remote devices must: (i) arrive after the halo data in current time step are consumed and

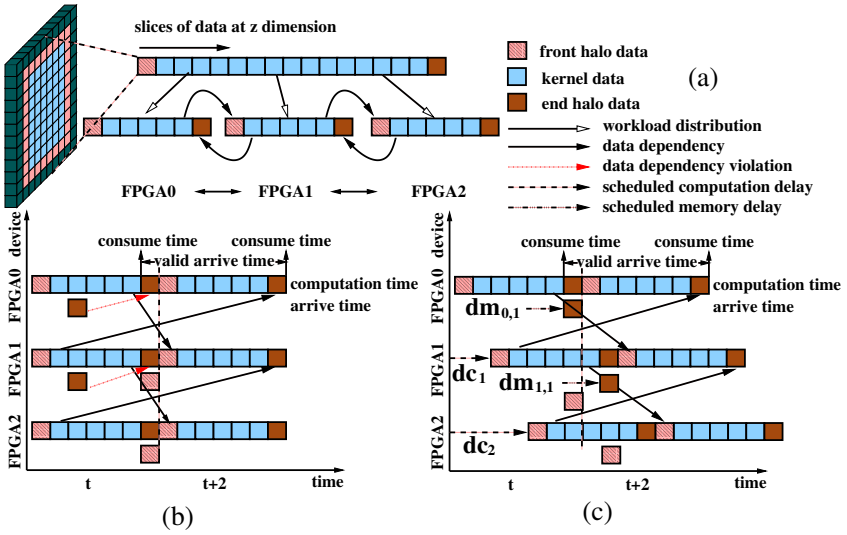


Figure 5.9: (a) Decomposed data for an FPGA path with three interconnected FPGAs, and the corresponding communication and computation operations in time dimension if (b) unscheduled and (c) scheduled. Each grid in the figure represents one data slice. Data dependencies, valid times and scheduled delays are labelled in this figure. The three FPGAs process three data units per clock cycle, and communication channels can transmit one data unit per clock cycle.

(ii) before the halo data in the next time step are used. In the unscheduled design in Figure 5.9b, the end halo data for `FPGA0` and `FPGA1` arrive too early and overwrite the existing end halo data before they are used, rendering all subsequent computations incorrect. An asynchronous communication model is built to translate data dependencies into timing constraints, and express arrival times with computational capacity and communication bandwidth. Variable and parameters for this model are presented in Table 5.2.

Timing constraints can be expressed with the consume time of halo data in the current time step and in the next time step, as shown in Figure 5.9b. For the halo data j in node n, the arrival times $tr_{n,j}$ are bounded as follows, with $j = front$ indicating the front halo data and $j = end$ indicating the end halo data:

$$\begin{cases} 0 & \leq tr_{n,front} & \leq \frac{dw_n}{par_n} \\ \frac{dw_n - sl \cdot w_D}{par_n} & \leq tr_{n,end} & \leq \frac{2 \cdot dw_n - sl \cdot w_D}{par_n} \end{cases} \tag{5.7}$$

where $\frac{dw_n}{par_n}$ is the computation time for one time step, and $dw_n - sl \cdot w_D$ indicates the distance between front halo data and end halo data.

Arrival times of halo data are determined by computational capacity in neighbouring nodes and the communication time. The front halo data at node n is derived from the kernel data $(dw_{n-1} - 2 \cdot w_D \cdot sl, dw_{n-1} - w_D \cdot sl)$ at node $n-1$. Similarly, end halo data at node n is generated from the kernel data $(sl \cdot w_D, 2 \cdot w_D \cdot sl)$ at node $n+1$. As there are $\frac{w_D \cdot sl}{par_n}$ cycles delay between when the kernel data are

loaded and when the correspondingly results are generated, unscheduled arrival times of halo data j in node n can be expressed as:

$$tr_{n,j} = \begin{cases} \frac{dw_{n-1}-w_D \cdot sl}{par_{n-1}} + \frac{w_D \cdot sl}{bw_n} \cdot m & j = front \\ \frac{2 \cdot w_D \cdot sl}{par_{n+1}} + \frac{w_D \cdot sl}{bw_{n+1}} \cdot m & j = end \end{cases} \tag{5.8}$$

where bw_n indicates the communication bandwidth between node $n-1$ and node n, m is the margin factor for communication operations, and $\frac{w_D \cdot sl}{bw_n} \cdot m$ is the communication time.

Communication scheduling refers to configuring memory delay dm and computation delay dc to satisfy the timing constraints. If halo data arrive too early, a memory delay dm is inserted to postpone the update time of the halo data in local memory. On the other hand, if halo data arrive too late, the halo data cannot be scheduled to arrive earlier. Instead, the starting time of the communication operations is delayed to postpone the latest timing constraints. The actual arrival times after scheduling are $tr_{n,j} + dm_{n,j}$. Since the inserted computation delay postpones the timing constraints, if we keep the timing constraints the same as in equation 5.7, the arrival times can be expressed as $tr_{n,j} + dm_{n,j} - dc_n$.

Run-time scaling

Once initialised, a dynamic stencil design investigates run-time status variations to utilise FPGA nodes provisioned during its lifetime. The mapped FPGA path can be expanded if FPGA available nodes can be connected to either the first node or the last node in the current FPGA path. Scaling a dynamic stencil design involves run-time evaluation, context switching, run-time reconfiguration and configuration update.

Context switching refers to redistributing the intermediate results from the current dynamic stencil design into the FPGAs of the new scaled dynamic stencil design: input and output arrays of the current FPGAs are loaded from off-chip memories back to the host memories; corresponding bit streams are configured into the new FPGAs; and the intermediate arrays are redistributed into the FPGAs of the expanded dynamic stencil design, thus ensuring that the context of the current stencil computation is preserved in each FPGA. A control unit for the dynamic stencil design is implemented on the host CPU, which executes run-time evaluation, design scaling and configuration update.

Run-time benefit rtb refers to the reduction in execution time for the remaining stencil computation when a dynamic stencil design expands into more FPGAs. The remaining workload for a stencil application is calculated with its remaining time steps $nsteps$ and distributed workload dw_n. If available nodes are employed, the distributed workload is reduced to dw'_n, and the saving in execution time is expressed as:

$$rtb = \frac{nsteps}{tk} \cdot \frac{(dw_n - dw'_n)}{par_n \cdot fq} \tag{5.9}$$

where $dw_n - dw'_n$ indicates the reduction in workload. As distributed workload is proportional to par_n (equation 5.6), rtb is the same for each node.

The scaling overhead refers to time consumed to reconfigure devices and redistribute data, and can be estimated as:

$$rov = \max\left(\frac{R \cdot \phi}{\theta}, \frac{dw_n}{bw_{pci}}\right) + \frac{dw'_n}{bw_{pci}} \tag{5.10}$$

where $\frac{dw_n}{bw_{pci}}$ and $\frac{dw'_n}{bw_{pci}}$ respectively indicate the time to load and redistribute memory data, through PCIe channels with bandwidth bw_{pci}.

The reconfiguration time can be estimated with bit stream size and throughput of reconfiguration interface θ. The bit stream size is calculated with resource consumption R and bit stream size per resource unit ϕ. 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 run-time reconfiguration is finished. The loading of context data, on the other hand, is executed in parallel with reconfiguration operations.

The scaling algorithm for a dynamic stencil design coordinates the communication model and the run-time performance model. Resource status is monitored after a dynamic stencil design is initialised at time zero, with run-time benefits and overhead evaluated for detected available resources. If run-time benefit rtb outweighs the scaling overhead rov, the scaling algorithm stalls computation for next time step in FPGA nodes, reconfigures available nodes and switches context data into the new nodes. Parameters of the communication model are updated, and the communication variables are rescheduled to respond to the design variations. Computation operations are then resumed for the scaled dynamic stencil design, and the scaling algorithm goes back to the monitoring phase. The algorithm is executed iteratively to adapt a dynamic stencil design to run-time resource variations.

5.2.3 Evaluation

We now report the results of our dynamic stencil approach with the RTM application to evaluate: (i) resource exploitation, (ii) design scalability and (iii) run-time adaptivity, which respectively reflect how available resources are exploited for the optimised single-node design, the initially constructed dynamic stencil design and the dynamically scaled dynamic stencil design. Hardware designs are described with MaxCompiler version 2012.1, operating at 100 MHz, and implemented on Xilinx Virtex-6 SX475T FPGAs, each hosted by one of four MAX3424A systems in an MPC-C500 computing node from Maxeler Technologies.

Resource exploitation in an FPGA is evaluated in terms of resource consumption and achieved design throughput. Resource consumption and design throughput of the optimised design are presented in Figure 5.10. The resource consumption is normalised against available resources, and the resource consumption when design parallelism is zero indicates the resources consumed by communication infrastructures. Before off-chip memory channels are saturated by par = 16, each replicated data path generates one result per clock, with design throughput and data-path resource consumption scaled linearly. Temporal blocking ratio tk is increased to two 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 consumption. 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 flip-flops, 952 DSPs and 989@BRAMs, with the optimisation model estimating the design to consume 255936 LUTs, 357120 flip-flops, 806 DSPs and 947 BRAMs. The optimisation model can capture variation in resource consumption with more than 90% accuracy, which can automatically optimise translated streaming architectures for FPGAs with various characteristics.

Design performance is given in Table 5.3. Reference single-device designs include a parallelised CPU design executed on a 4-core Intel i7-870 CPU, Blue Gene/P [73] and Blue Gene/Q designs [58],

single-device performance					
System	Freq (GHz)	TH(Gflops)[1]	P(Watt)[1]	E(Gflops/w)[1]	Speedup[1]
CPU	2.93	1.8	183	0.01	72×
GPU [61]	1.15	58.8	369	0.159	2.2×
MaxGenFD [71]	0.1	71.3	137	0.52	1.8×
Dynamic Stencil	0.1	130.67	142	0.92	–

multi-device performance (GFlops)						
System/number of devices	2	4	8[2]	16[2]	32[2]	Speedup
Blue Gene/P [73]	2.98	5.96	11.92	23.84	47.68	87.8×
Blue Gene/Q [58]	38.4	76.8	153.6	307.2	614.4	6.8×
Cray XK6 [77]	181	362	524	1048	2096	2.00×
MaxGenFD[3] [71]	128.3	196.8	n/a	n/a	n/a	2.66×
Dynamic Stencil	261.3	523	1045	2091	4184	–

[1] TH, P, and E respectively stand for throughput, power consumption, power efficiency. Speedup relates to performance compared with the dynamic stencil design.
[2] Limited by available resources, performance for more than four FPGAs is simulated. When 1–4 FPGAs are involved, measured performance confirms the simulated results.
[3] MaxGenFD supports up to 8 FPGAs, performance cannot be simulated due to lack of optimisation details. Measured scalability for four FPGAs is 0.69.

Table 5.3: Single-device and multi-device performance comparison.

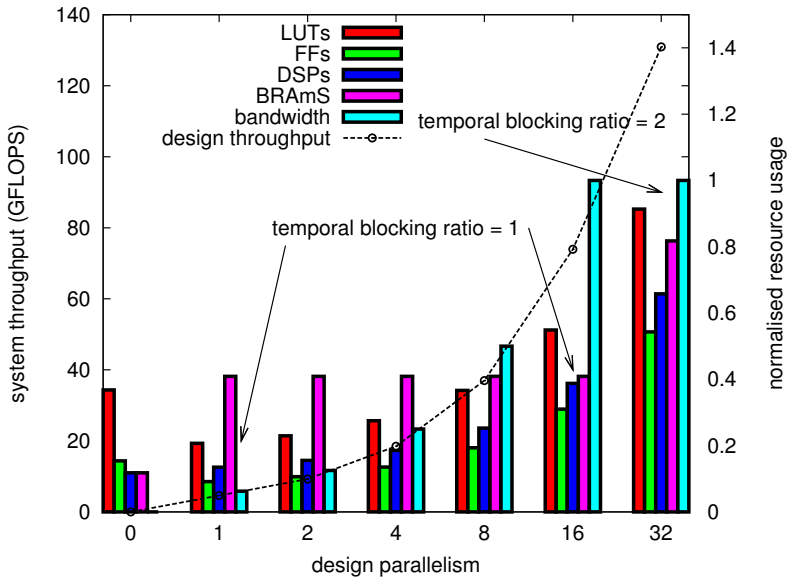


Figure 5.10: Design throughput and resource consumption of the RTM design. The optimisation increases design parallelism to 16 until the bandwidth bottleneck is hit, and increases the temporal blocking ratio to utilise leftover resources.

a GPGPU design optimised by NVIDIA [61] and customised for NVIDIA Tesla C2070, and an FPGA design developed with MaxGenFD [71].

Overall 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. Performance of CPU and GPGPU designs is limited by their general-purpose memory system. A run-time profiling shows that the optimised GPGPU design can only achieve 35% memory efficiency, i.e., loading one data unit takes three clock cycles. Performance of MaxGenFD design is limited by the memory bandwidth due to lack of temporal blocking in its optimisation configurations. The optimised design for RTM is up to 1.8 to 72 times faster and 1.7 to 92 times more power efficient than the reference designs.

Design scalability reflects the effectiveness of the asynchronous communication model. For the current platform, inter-FPGA communication operations are either through point-to-point channels with 3.2GB/s bandwidth or through 10Gb/s Ethernet connections between CPUs, with data moved between CPUs and FPGAs through 1GB/s PCIe channels. The lower bound of inter-node bandwidth bw is 1GB/s, while on-chip results are streamed with 48 bytes per clock cycle.

In the asynchronous communication model, the computation delay dc in involved FPGAs is scheduled to be $5w_D = 25$ data slices sl to reduce the bandwidth requirement to 0.8GB/s, with margin factor $m = 1.25$. Memory delay dm is scheduled to ensure local halo data are consumed before being overwritten. Limited by available FPGAs in our platform, our current design scales up to four FPGAs. Based on computation throughput of utilised FPGAs and available bandwidth, performance of the dynamic stencil design when more FPGAs are involved is simulated.

The simulated and measured results are presented in Table 5.3, which shows that linear scalability has been achieved for the initialised dynamic stencil design. Previous large-scale designs on Blue

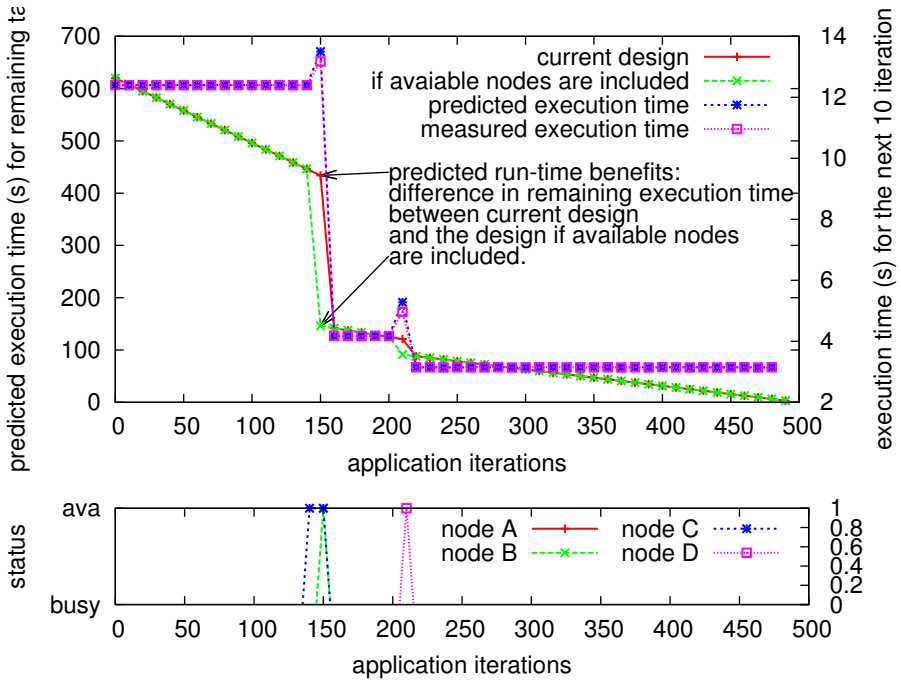


Figure 5.11: Evaluation and prediction of the run-time performance model during one of the test cases, at the application iteration (time step) dimension. The resource status is measured from target cluster. ("ava" stands for available.)

Gene/P [73], Blue Gene/Q [58] and Cray XK6 [77] are also introduced to provide a comparison. The measured results confirm the simulated performance, and overall design throughput reaches 4.09 Tflops when 32 FPGAs are involved, outperforming the reference designs by 2 to 88 times. Besides throughput, power consumption in large-scale clusters determines the maintenance cost such as cooling infrastructures and electricity, and plays an important role in large-scale designs. Power consumption numbers are not provided in previous work [73, 58, 77]. If we make a conservative assumption that a Tesla X2090 GPGPU in Cray XK6 consume the same power as the Tesla C2070 design in Table 5.3, the dynamic stencil design is 5.2 times more efficient than the stencil design running on Cray XK6 when including all host and accelerator power consumption.

Run-time adaptivity of the developed design is evaluated with design performance and device-level resource utilisation ratio, when the RTM design is mapped into the reconfigurable cluster. For the available four FPGAs, static designs with 1, 2, 3 and 4 device-level parallelism are developed and executed to provide comparison. Run-time status during 10 separated time periods is measured and used as 10 test cases in this experiment.

The evaluation process of the run-time performance model for one of the test cases is demonstrated in Figure 5.11. The run-time performance model predicts the execution time for the remaining tasks of the current design as well as the scaled design. When new nodes become available, the difference between the two predictions indicates the run-time benefits.

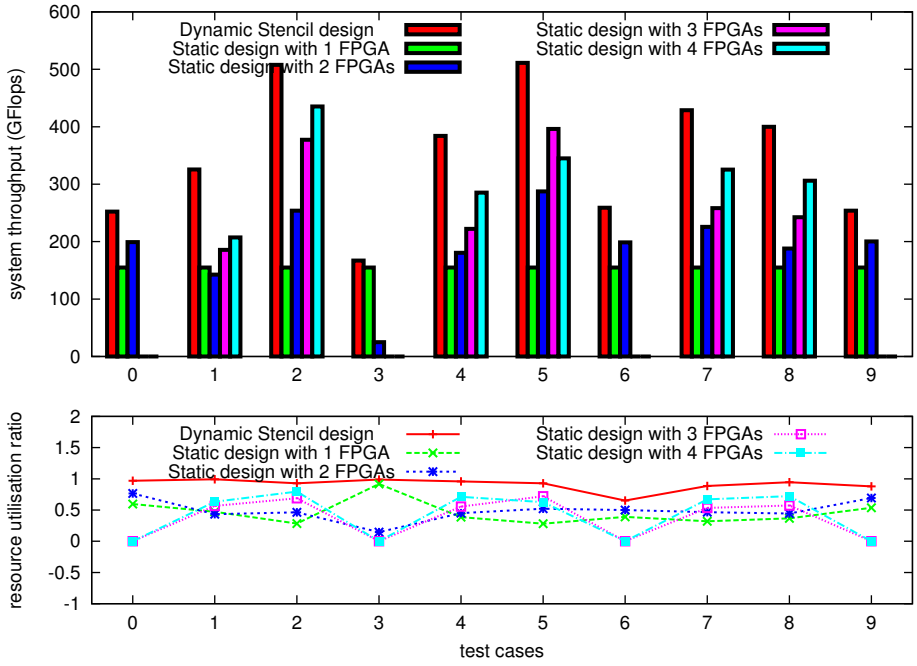


Figure 5.12: Design performance and resource utilisation for 10 test cases.

FPGA node A is available when the application is launched, and nodes B, C and D are released by other computational tasks at 150, 142 and 209 iterations, respectively. Although node C becomes available earlier than node B, the detected FPGA path first expands when node B is released due to the lack of communication channels between node A and node C. If nodes B and C are included in the dynamic stencil design, execution time for the following tasks is reduced by 357.4s, with 0.71s run-time overhead introduced. As the benefit outweighs the overhead, node B and node C are reconfigured to cooperate with the existing node A. Context data are redistributed, and design variables are rescheduled to ensure linear scalability and correct functionality when the dynamic stencil design is expanded. Similarly, node D is employed by the dynamic stencil when it becomes available.

As shown in Figure 5.11, the measured performance aligns with predicted execution time for the remaining tasks showing high accuracy of the performance model. Corresponding results in the test case are shown in Figure 5.13. Device-level parallelism for the static design using one FPGA is limited to one, while the static designs using more FPGAs need to wait for released nodes to start. The dynamic stencil design finishes 490 time steps in 297 seconds, outperforming the static designs by 1.67 to 2.72 times.

The resource utilisation ratio is calculated with measured performance and the theoretical performance upper bound. The theoretical performance is calculated as the overall performance if FPGAs are fully utilised once released by other applications. The measured performance and resource utilisation for the 10 test cases are shown in Figure 5.12. The average resource utilisation ratio for the dynamic stencil design is 91%. The gap between the achieved resource utilisation ratio and the full utilisation ratio (100%) is introduced by the reconfiguration overhead and communication infrastructure. As shown in the test case in Figure 5.13, node C remains idle until the dynamic stencil design expands into node B, as there is no

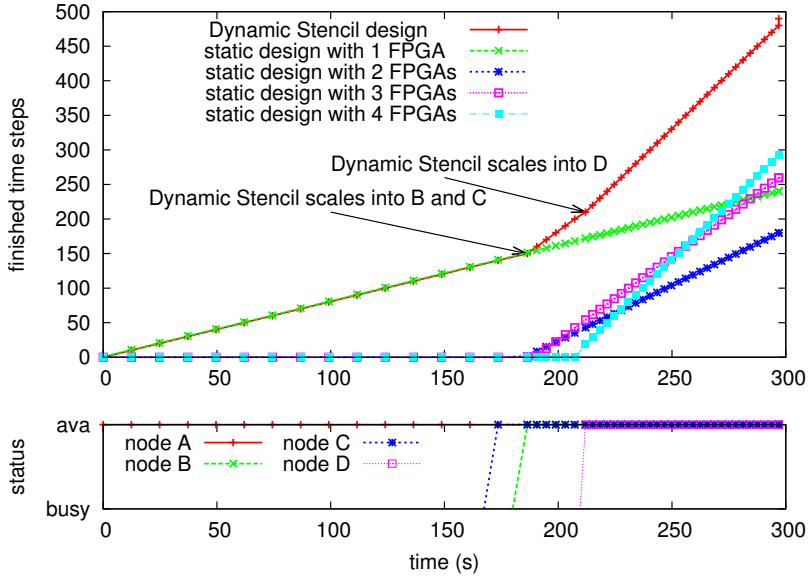


Figure 5.13: Performance of a dynamic stencil design and a static design with 1, 2, 3 and 4 FPGAs at the time dimension.

communication channels between node A and node C. Resource utilisation for static designs is limited between 40% and 49%. In other words, it is limited by predefined communication and computation patterns, and half of the resources in the cluster remain idle. On average, the high resource utilisation of the dynamic designs enables them to run 1.8 to 2.3 times faster than the static designs.

5.2.4 Summary

For large-scale reconfigurable clusters, the effectiveness of conventional static design methods that predefine communication patterns and hardware configurations are limited by unpredictable run-time conditions. We introduce *dynamic stencil*, a novel approach that statically optimises target applications for various FPGA nodes, and dynamically constructs an executable design that automatically adapts to resources available at run time. In particular, we achieve a high resource utilisation ratio and concrete speedup over reference designs at each stage of the approach for computationally intensive stencil applications.

Our current approach is limited to single-tenancy considerations: a dynamic stencil design tends to occupy all available resources during its execution, which may not be the optimal solution if targeting maximum overall performance of multiple tasks. Idle nodes due to lack of communication channels in existing dynamic stencil designs can be occupied by other computational tasks, which can further increase resource utilisation. We intend to extend our work in Year 2 to support dynamic design methods for multi-task and multi-user environments, which will be built on top of the current dynamic stencil approach, to exploit more complex run-time scenarios.

5.3 Delta Merge

The column store utilised in the SAP HANA database uses dictionary encoding and other compression algorithms to ensure that all relevant data is accessible in main memory. Figure 5.14 illustrates dictionary compression for a column of names. Dictionary encoding has the effect that a single write operation into this compressed data would require resorting and recoding the compressed structure, and as such, a *delta* storage is used to maintain the list of write accesses to the data. Figure 5.15 illustrates the data structures utilised to maintain a table within HANA.

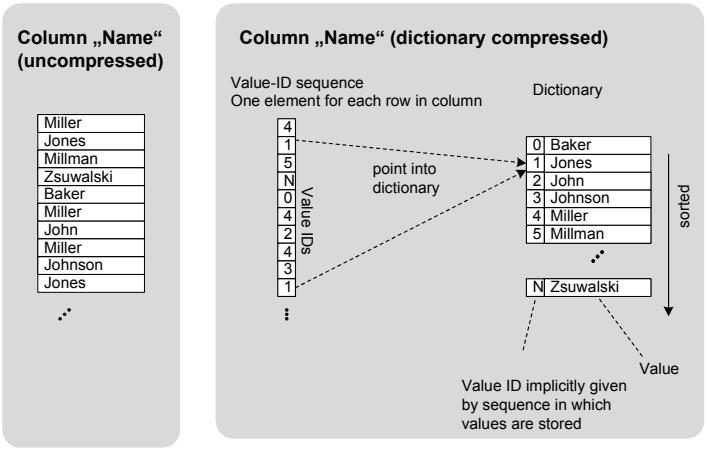


Figure 5.14: Column store dictionary compression.

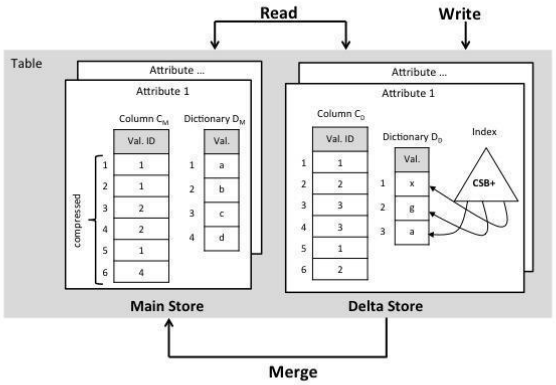


Figure 5.15: SAP HANA table data structures.

The Main Store contains two data structures: (i) the Main Dictionary, D_m and (ii) the Main Column, C_m, which is a vector of Value IDs. These Value IDs are defined by the value position in D_m. The Delta Store consists of an unsorted dictionary, the Delta Dictionary, D_d, the Delta Column, C_d, a vector of Value IDs defined by value position in D_d, and the Delta Map, K_d, which maps the values with their positions in D_d.

As each read of the database requires a search of both the Main Store and Delta Store, the Delta Store needs to be organised in such a way that it does not limit the read performance of the system. This is accomplished using M_d, enabling a Value ID entry to be found in $\log(|D_d|)$. The map is implemented as a cache-sensitive B+ tree (CSB+ tree) and is therefore ordered by the dictionary value that maps to its respective Value ID (its position in D_d). As D_m and D_d are not related, there can be repetition between values in each; however, each individual dictionary will only contain unique values.

Each update of the Delta Store requires a look up for the value in M_d. If the value does not exist in the map, then the dictionary value is appended to the end of D_d and the associated index value is appended to the end of C_d. If the value does occur in M_d, then the associated Value ID, from M_d, is appended to C_d, and no operation occurs on D_d.

The consequence of maintaining two data structures is the need to occasionally *merge* them. This preserves the read and write performance by ensuring that the Delta Store does not grow overly large, but requires additional overhead to perform the merge. The delta-merge process requires two main steps, firstly merging D_M and D_d, and secondly updating C_m and C_d. Once C_m and C_d are updated, C_d is simply appended to the end of C_m to create the new Main Store column, C_n. As the HANA system must continue to operate while the delta-merge process executes, it is necessary to permit reading of the old Main Store and Delta Store during a merge, as well as support writing and reading to a new Delta Store structure that is generated after a merge operation begins, as illustrated in Figure 5.16.

The delta-merge operation is an important validation case for HARNESS as it can have a large impact on the performance of a HANA system. The delta merge can operate over a diverse set of inputs, from columns with a handful of entries, to those containing millions. The delta merge, while a regular operation within the HANA system, is not performed on a fixed schedule. There are a number of triggers that govern when it is performed, and it is even possible to manually trigger a delta merge. The ability to perform this task over a set of resources allows for the impact of this operation to be effectively managed.

There are a number of scenarios in which the delta merge can be run, and depending on the scenario, the ideal allocation of processing resources may change. For instance, on very small tables with a limited number of columns, the delta merge has negligible performance impact. Allocating accelerators and transferring memory in this case is unnecessary. However, for large columns, that can contain millions of entries, accelerators can provide measurable performance gains. There is also the option to load balance individual column tasks within a delta-merge operation. As each column is merged separately, there is the opportunity to perform multiple column tasks in parallel over a number of processing resources.

There is also the issue of system demand and resource contention. Usage levels of the HANA system as a whole may vary. If the system is not under significant load, then it may not be necessary to require any acceleration resources to perform the delta merge, as transaction levels will not be adversely impacted to the point where they fall below customer requirements. However, during periods of high system usage, it may be prudent to offload all delta merge tasks to accelerators to enable the host CPUs to continue servicing requests and mitigate all impact of the delta merge.

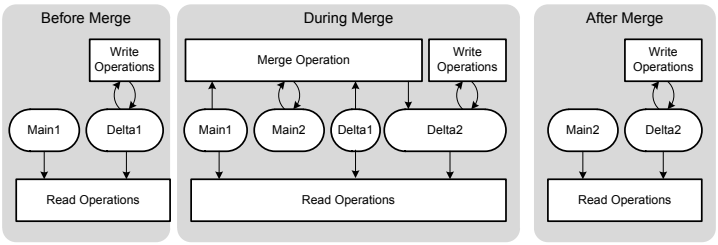


Figure 5.16: SAP HANA delta-merge operation.

SAP HANA provides a rich environment for validation of the HARNESS project, and in particular the delta-merge process due to the number of scenarios under which it operates and its importance within the HANA system. The scenarios listed above provide a real challenge even on static, known platforms. Being able to effectively manage these tasks in a dynamic cloud environment will provide a genuine challenge to HARNESS.

5.3.1 Observation and characterisation of tasks

The delta-merge operation is highly sensitive to input data and, therefore, the placement of delta-merge execution is dependent on the inputs provided. As a consequence, it is important that any scheduling framework be able to adequately assess the availability of processing resources and the suitability of those resources to process a given task. In the instance of the delta merge, it could be the case that very small column sizes are best performed on the host CPU, while large column delta merges can gain a measurable speed up on local accelerator cards. Thus, there is a need for the run-time management system to understand the input to a given task and allocate devices appropriately. Of course, this can change from platform to platform as the mix of processing resources may be different.

To address this problem, the SHEPARD run-time management system (Section 4.1) maintains a library of implementations along with associated performance models. These are used to imbue an application with extra non-functional information that is useful when making decisions. The presence of the performance models for each implementation allows SHEPARD to better understand the costs associated with choosing to perform a task on a certain processing resource.

To create these performance models, exploration executions for each implementation on each device are needed. Inputs to the exploration observations should be representative of those likely to be encountered at run time. These are logged to a management database (DB), where performance models can then be generated from the observed data. Figure 5.17 represents the scaling observed over single column delta-merge tasks of increasing input size. The results depict a near linear increase in execution time over the set of inputs. The inputs are sized according to the main dictionary with the delta dictionary size being assumed to be five percent of the main dictionary size.

5.3.2 Adaptive allocation of tasks based on input

With performance models, applications can be enhanced with extra information on the tasks they perform. This allows the HARNESS platform to make informed decisions when deciding which device should perform any given task. By making these decisions at run time, the executive can use actual run-time

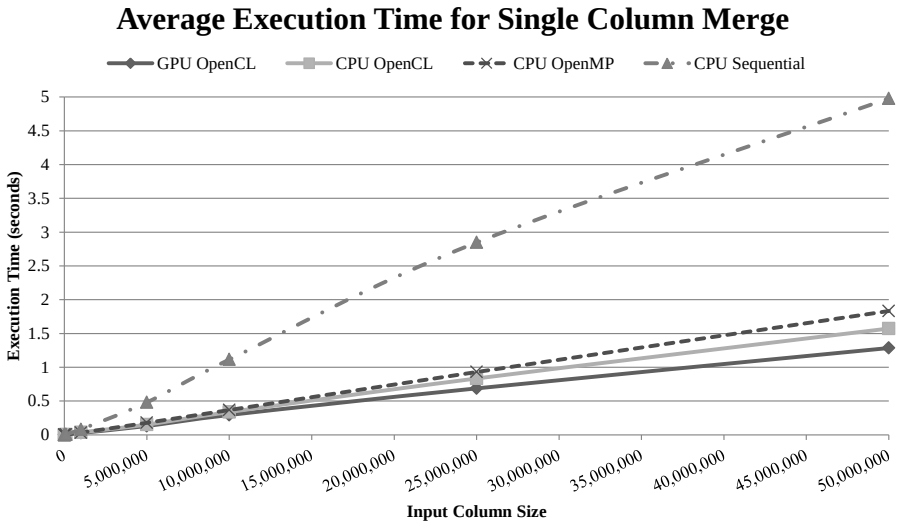


Figure 5.17: Delta merge scaling with input.

values of inputs to each task to parameterise the performance models and generate execution time estimations. Given these estimations the platform can then choose the most appropriate device to process a task for a given set of inputs. This means that for tasks, such as the delta merge, where input size can determine the performance of the operation, the platform is equipped to intelligently decide which device should execute a task, removing the need to hard-code explicit scheduling rules where the target hardware must be known at design time. Since the delta merge can be performed on tables containing very small columns, the decision process should place minimal overhead on the overall performance per task. This is stipulated by Requirement R31 (“the HARNESS platform shall efficiently schedule small compute jobs [for column based merges]”) [32].

Figure 5.18 represents the decision over a single-column delta-merge task with inputs ranging from one thousand to fifty million values. The graph illustrates the ability to choose the device that yields the lowest execution time, while imposing negligible overhead.

5.3.3 Task-parallel load balancing over heterogeneous resources

The delta-merge operation is performed on a per-table basis. Each operation performs a delta-merge task on each column independently. This affords the ability to perform these column-oriented delta-merge tasks in parallel over a set of devices to minimise average execution time per task and per table.

Effectively allocating column delta-merge tasks to processing resources requires also assessing the current demand on a given resource. If current processing resource demand is not taken into account, then there is the potential that all tasks are allocated to the same resource resulting in over-utilisation of that resource. By estimating the current demand on a processing resource, the workload can be more effectively balanced. This should result in an increase in average task performance and an overall reduction in the total time taken to complete a full table delta merge operation.

Considering current resource demand also allows the executive to avoid using the host CPU when the HANA system is under high load. In this case, the execution of the delta merge on the host CPU

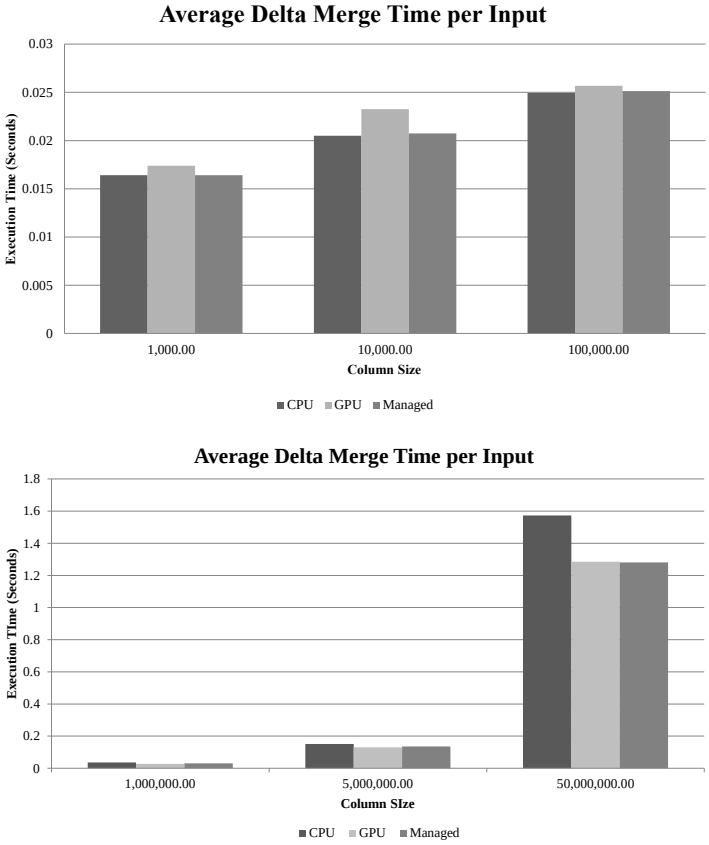


Figure 5.18: Allocation decisions over input.

would be adversely impacted so the executive is likely to allocate delta merge tasks to accelerators. Under low-load situations, the CPU is not as busy and, therefore, the likelihood of its utilisation in delta-merge tasks is increased. This scenario addresses requirement R32 (“the HARNESS platform shall intelligently schedule [column-merge] jobs to resources). Requirement R33 (“the HARNESS platform shall support elastic resource allocation [for column-merge jobs]”), is also addressed in this scenario due to allocation being influenced by resources that are currently available and their current demands.

Figure 5.19 depicts the average execution time per delta-merge column task over tables of an increasing number of columns. In the test platform the GPGPU is capable of performing faster delta-merge operations. The managed allocation of tasks demonstrates the potential performance gain achievable through the ability to dynamically offload tasks to multiple processing resources. The speed up achieved is a direct result of task sharing between processors and the resultant reduction in contention on each device. Figure 5.20 represents the total execution time required to complete all column delta merge tasks within each table, demonstrating the ability to lower total execution time by sharing tasks over multiple heterogeneous processors.

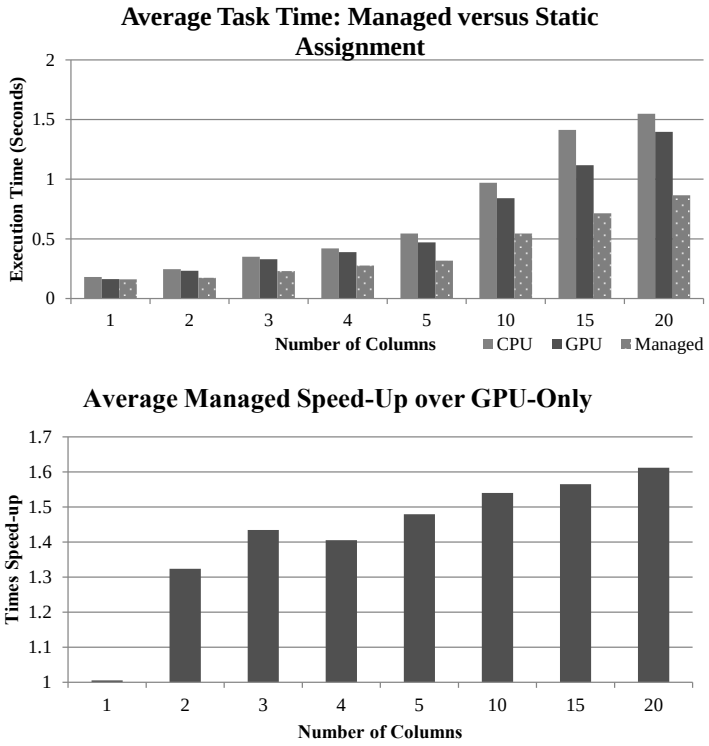


Figure 5.19: Average delta-merge task performance with column number scaling.

5.4 AdPredictor

In this section we describe our experiments with AdPredictor [41], a Bayesian machine-learning system that is used by Microsoft's Bing search engine [63] for advertisement ("ad") recommendations. Our initial effort with AdPredictor has been to study how fast we can accelerate the **training** process using a single dataflow engine compared to conventional CPUs.

The training module processes observations about ads, called *ad impressions*. Ad impressions are tuples X that consist of attributes x_i related to a particular user search session. Example attributes include the user's identifier, the IP address, the query's terms, and the ad's relative ranking on the page. For each ad impression, the system observes whether the ad was clicked or not clicked. This information is stored as part of a single ad impression, resulting in several ad impressions being logged for training. The goal of the training process is to update prior belief with a set of new observations to improve ranking of ads that will be most likely clicked.

The training process is as follows. Given an observation (X, y), where $y = 1$ when an ad was clicked and $y = -1$ otherwise, and prior probability distribution $p(\theta)$, the system must infer the new posterior $p(\theta|X, y)$. For every feature value x_i, AdPredictor maintains a Gaussian belief over parameter θ_i. In practice, these prior beliefs can be viewed as vectors:

$$\boldsymbol{\mu} := (\mu_1, \mu_2, \ldots, \mu_n)^{\mathrm{T}}$$

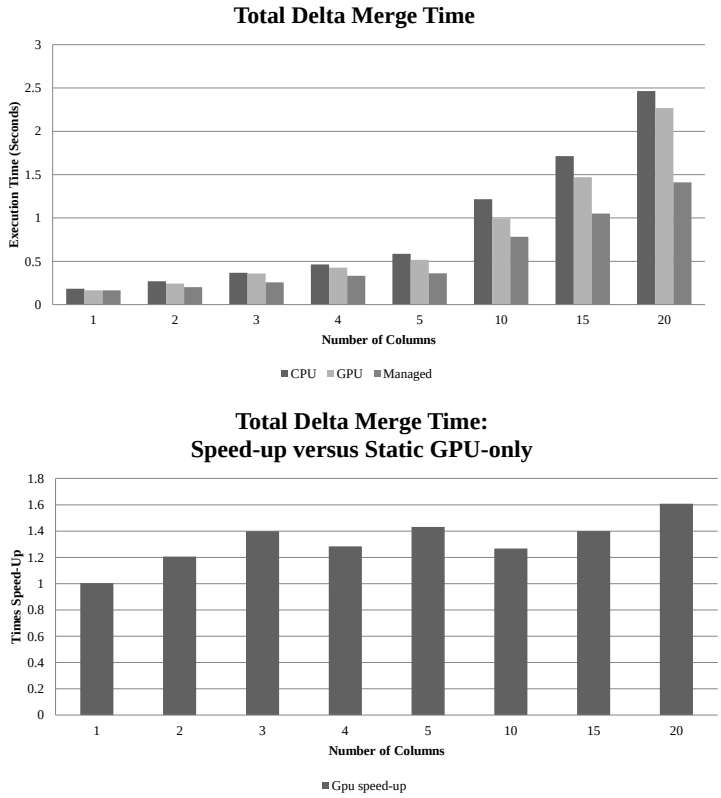


Figure 5.20: Total delta-merge operation performance with column number scaling.

and

$$\boldsymbol{\sigma^2} := (\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2)^{\mathrm{T}}$$

The likelihood function $p(y|X, \theta)$ is then approximated using expectation propagation on factor graphs in order to derive the following closed-form update equations for the mean and variance of the posterior distributions $\mathcal{N}(\theta_i; \widetilde{\mu}_i, \widetilde{\sigma}_i^2)$:

$$\widetilde{\mu}_i = \mu_i + yx_i \cdot \frac{\sigma_i^2}{\mathcal{S}} \cdot v\left(\frac{y \cdot \sum_{j=1}^{n} x_j \mu_j}{\mathcal{S}}\right)$$

and

$$\widetilde{\sigma}_i^2 = \sigma_i^2 \cdot \left[1 - x_i \cdot \frac{\sigma_i^2}{\mathcal{S}^2} \cdot w\left(\frac{y \cdot \sum_{j=1}^{n} x_j \mu_j}{\mathcal{S}}\right)\right]$$

In the equations above, $\mathcal{S}^2 = \sum_{j=1}^{n} x_j \sigma_j^2 + \beta^2$ is the sum of variances of the independent and identically distributed parameters involved in the training instance X, plus the variance β^2 of the Gaussian noise process that generated it, and $v(\cdot)$ and $w(\cdot)$ are functions that control the magnitude of the mean and variance update, as follows:

$$v(t) = \frac{\mathcal{N}(t; 0, 1)}{\Phi(t; 0, 1)}$$

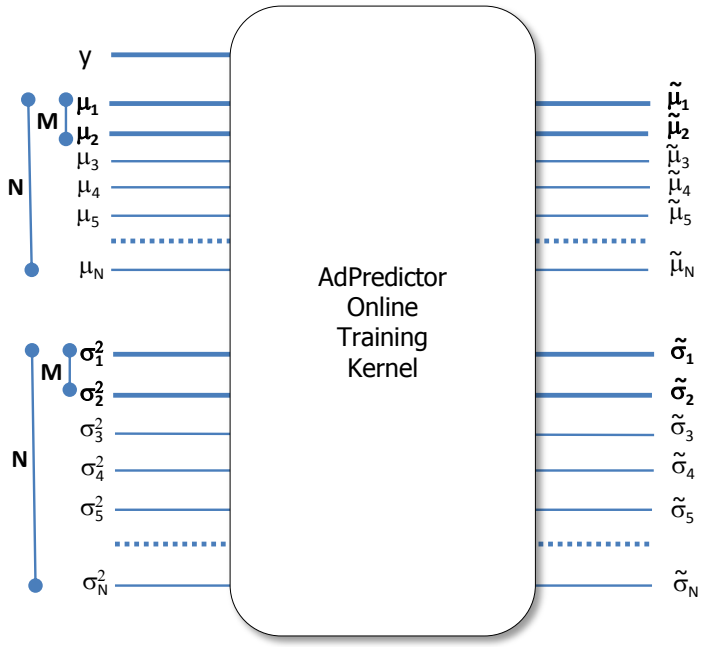


Figure 5.21: The training kernels of CPU and DFE implementations.

and

$$w(t) = v(t) \cdot [v(t) + t]$$

where $\Phi(t) = \int_{-\infty}^{t} \mathcal{N}(s; 0, 1)ds$ is the standardised cumulative Gaussian density function.

We developed kernel implementations for the CPU and DFE that implement the update equations described above (Figure 5.21). A kernel receives as input a stream of ad impressions. Each impression is represented by y (-1: not clicked, 1: clicked) and the prior distributions of each of its features. The output is a stream of corresponding new posterior distributions. Our DFE architecture can be parameterised by N and M, which represent the total number of features and the number of features read per DFE tick, respectively. While the N value is part of the application specification, the M value allows us to trade off the kernel parallelism and resource utilisation: decreasing the level of parallelism provides more opportunities to share resources and decrease the number of required resources. Thus, we require N/M DFE ticks until all N features are loaded into the DFE for the kernel to fully process one impression, but we also only need M parallel computation units ("threads"). In general, when $M == N$ then we derive a fully parallelised design, when $M = 1$ we derive a resource-efficient sequential design. For instance, if $N = 10$ and $M = 2$, then it takes give DFE ticks until all 10 features have been loaded into the DFE, requiring only two parallel computation units.

The 2012 KDD Cup data set (a realistic data set used for evaluating AdPredictor and described in Section 5.2.1 of Deliverable D2.3 [33]) employs 10 features per impression and single floating-point precision to handle Gaussian distributions. Our initial fully pipelined DFE design for the training module did not fit the Xilinx Virtex-6 SX475T FPGA inside a Maxeler MAX3424A card. Given that we can only stream around 2GB/s from the CPU to the DFE, we can only receive 4–5 single-precision values per cycle

at 100MHz. Hence, our initial design, which processes data at a rate of one impression per DFE tick, performs faster than the communication rate. This gives us the opportunity to reduce the rate of the kernel to match the communication rate, and at the same time reduce resource utilisation by sharing resources.

Our second DFE implementation is now parameterised by N and M, which represent the total number of features and the number of features read per DFE tick, respectively (see Figure 5.21). Taking into account the resource utilisation level, the communication rate and padding restrictions, we found the appropriate values for N and M to be 12 and 4, respectively. While the KDDCup training set requires 10 features, this DFE kernel can be used with this data set by adjusting the input arguments. For the CPU version, we developed a fully multithreaded version of the same specification using OpenMP with the full (`-O4`) compiler optimisation. In our experiments we compared two CPU versions using one thread and 20 threads running at 2.67Ghz for each CPU core, against the aforementioned DFE version running at 100Mhz. All DFE results are compared against CPU versions to verify for correctness.

# Impressions	CPU-1 @2.67Ghz	CPU-20 @2.67Ghz	DFE @100Mhz	DFE vs CPU-1	DFE vs CPU-20
10	0.12 ms	0.04 ms	0.33 ms	3 × slower	8 × slower
50	0.52 ms	0.04 ms	0.37 ms	1.4 × faster	9 × slower
100	0.93 ms	0.07 ms	0.34 ms	3 × faster	5 × slower
500	4.7 ms	0.30 ms	0.36 ms	12 × faster	1.2 × slower
1,000	9.3 ms	0.58 ms	0.37 ms	25 × faster	1.6 × faster
5,000	46 ms	2.9 ms	0.60 ms	77 × faster	5 × faster
10,000	93 ms	5.6 ms	0.92 ms	101 × faster	6 × faster
50,000	465 ms	28 ms	2.9 ms	160 × faster	9 × faster
100,000	931 ms	56 ms	6.3 ms	148 × faster	9 × faster
500,000	4660 ms	282 ms	33 ms	141 × faster	9 × faster
1,000,000	9312 ms	560 ms	64 ms	145 × faster	9 × faster
5,000,000	46522 ms	2796 ms	321 ms	145 × faster	9 × faster
10,000,000	92975 ms	5599 ms	624 ms	149 × faster	9 × faster

Table 5.4: The performance of three designs implementing the training module shown in Figure 5.21.

The results are summarised in Table 5.4 and Figure 5.22. The table shows the performance of three designs implementing the training module shown in Figure 5.21. The values shown correspond to the total time to stream in and out a group of ad impressions with sizes that range from 10 to 10 million. We compared the performance using two processing elements: a dual Intel Xeon X5650 (6-core per CPU) and our DFE design (N=12,M=4, single precision floating-point). We developed two versions targeting the Intel Xeon X5650 running at 2.67Ghz: a single threaded version (CPU-1) and a parallelised version using 20 threads. The DFE version runs at 100Mhz. It can be seen that the DFE version is very efficient for large tasks: starting with 5000 impressions, the 100Mhz DFE architecture beats the single-threaded Xeon by 77 times and the 20-thread version by six times. At 10M impressions, the DFE version runs at two and one orders of magnitude faster than the single-threaded version and multithreaded version respectively. Figure 5.22 gives the overall result. We can clearly see that the task size influences the relative performance of the three designs, with smaller task sizes performing better with CPU designs and very large tasks performing better with the DFE.

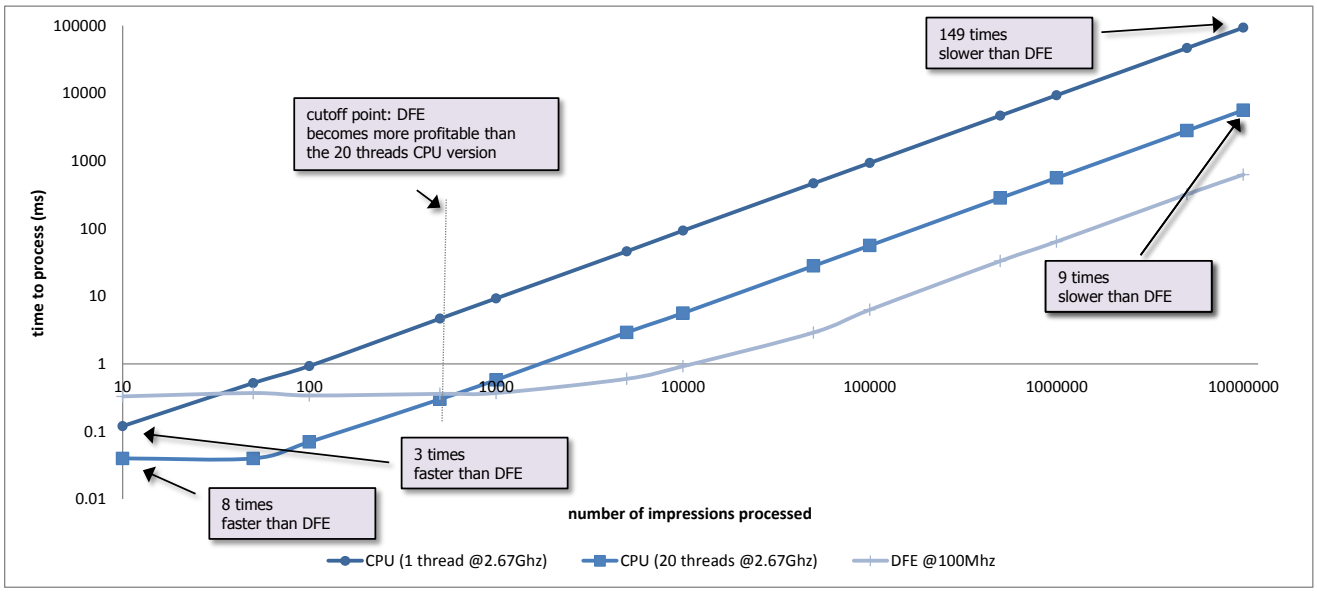


Figure 5.22: Results comparing single-threaded (CPU-1) and 20-threaded (CPU-20) 2.66Ghz Xeon against a 100Mhz DFE (see Table 5.4). Task size (number of impressions to process) influences the relative performance between these 3 designs, with smaller task sizes performing better with CPU designs, while very large tasks performing better with the DFE. These performance profiles can be used by the runtime management system to determine the best implementation depending on task size.

By characterising the AdPredictor tasks by the number of impressions, the HARNESS run-time management system can select the best design: using a sequential CPU version for very small tasks (<= 10), a multithreaded CPU version for small to medium tasks (<= 1000), and a DFE for large tasks (>= 1000).

6 Future Work

6.1 Support for Virtualisation and Horizontal Allocation

In Year 1 we developed two independent infrastructures that realise the two-tier run-time management approach described in Section 3.3.1. Both infrastructures operate on a single machine, and manage OpenCL and DFE compute resources, respectively. More specifically, these two infrastructures support vertical scaling, in which the workload is adapted to multiple heterogeneous processors within the context of a compute node.

In Year 2, we plan to extend this work in the context of tasks T3.2 and T3.3, to implement virtualised compute resources (Section 3.4). This way, a single run-time management system can support different types of compute devices, including those supported by OpenCL and MaxelerOS, that are available across multiple machines. Our new infrastructure will allow us to: (i) experiment with different scheduling algorithms (executive strategies) that may depend on the type of job; (ii) find effective implementation and resource characterisations; and (iii) experiment different trade offs in horizontal and vertical scaling in terms of efficiency, energy consumption and pricing models.

In order to implement the new run-time management system, we identify five components, illustrated in Figure 6.1:

(a) a compute resource API to virtualise heterogeneous resources;

(b) a database to store implementations, implementation characterisations and executive scheduling strategies;

(c) the executive component;

(d) a socket interface/protocol for communication between the executive and resources; and

(e) compute resource wrappers for OpenCL and MaxelerOS devices.

These components will be integrated, and we will use the experiments and results obtained in Year 1 (Chapter 4) as reference for our new run-time management system in Year 2.

In the context of this work, we plan to continue the development of following three types of virtual compute resources:

- **OpenCL devices.** *SAP AG* (SAP) is developing a class of compute resource that revolves around OpenCL devices, offering the possibility to run optimised kernels on a wide variety of heterogeneous hardware, including CPUs, GPGPUs and the Intel Phi.
- **DFE groups.** *Maxeler Techologies* (MAX) will continue to improve the *groups* mechanism, which manages a collection of dataflow engines configured identically to complete a sequence of incoming tasks. DFE groups employ a device-queue mechanism that dispatches incoming tasks to available DFEs in a transparent way.

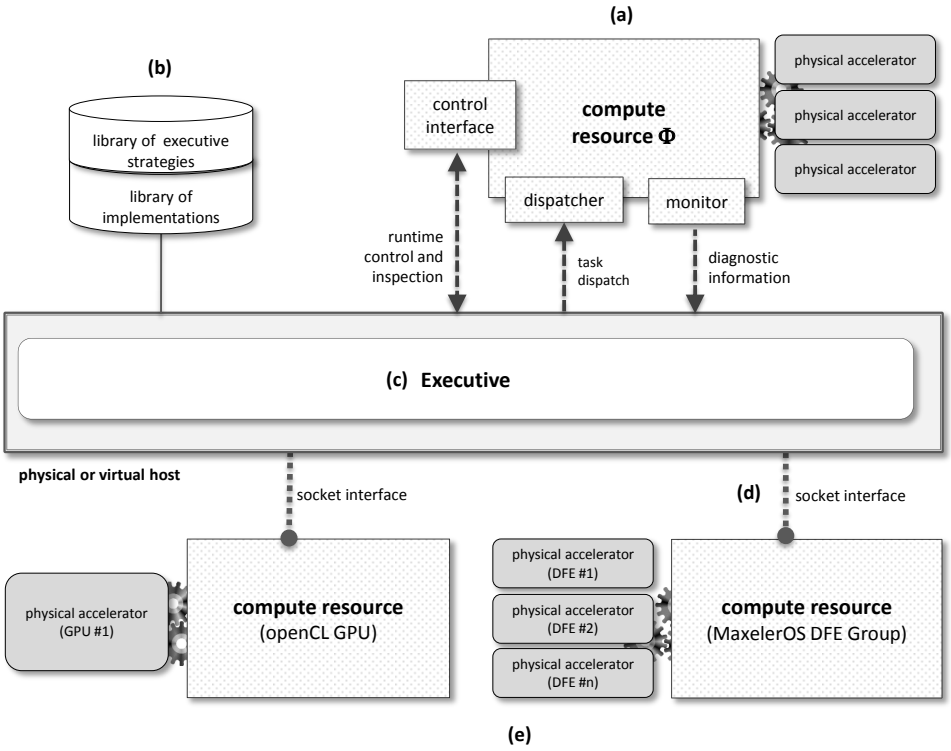


Figure 6.1: Technologies to be developed and integrated in Year 2: (a) a standard *compute resource API* that virtualises physical processing elements and allows the executive to control and inspect the resource, acquire diagnostic information and execute tasks; (b) a *database* to store implementations (binaries and characterisation) and executive (scheduling) strategies; (c) the *executive*; (d) the *socket interface and protocol* that allows the executive to communicate with local and remote compute resources; and (e) *wrappers* for OpenCL and MaxelerOS DFE devices using the compute resource API.

- **DFE peers.** *Imperial College London* (IMP) is developing an experimental type of compute resource that, like DFE groups, manages a collection of dataflow engines. However, each dataflow engine must be connected with each other in a chain-like topology, with the first and last element of the chain connected to the CPU. Section 5.2.2 presents the initial experiments with mapping stencil computations onto this type of compute resource.

DFE groups, as part of the Maxeler infrastructure, provides an allocation and scheduling mechanism that has been developed to satisfy the needs of many different applications. However, for optimal utilisation within a cloud platform environment further functionality would be required.

One improvement would be enhanced monitoring features. Offering a group view of DFE hardware allocation and use, as mentioned in Section 4.2.2, would allow the cloud platform manager to clearly see group usage rather then having to accumulate this information from the DFE view. Also providing fast monitoring information to the executive, as mentioned in Section 4.2.3, would mean a better decision could be made about the appropriate resource for a task. Additional information, such as current group size, average task execution time, estimated group pipeline length and details of group sharers, could all be used to calculate the time taken to execute a task on the MPC-X resource.

Other improvements to DFE groups would include allowing the governor (i.e., DFE group manager) behaviour to be influenced by the cloud platform manager. This would enable the platform to guide the governor's decisions. This would be beneficial as the governor uses predominantly a utilisation model to decide DFE allocation and has no knowledge of the platform requirements of the groups. Another improvement would be to advertise the increase in a group's size to allow the using program to pre-load some state into the DFE. Static groups have the advantage that state can be pre-loaded into each DFE, performed by getting a handle to all the engines in the group and loading them individually. This allows the DFE initial state to consist of two parts: the configured max file and the loaded state. One issue with dynamic groups is that when an engine is added to the group, the user has no knowledge of it. The governor will configure the DFE with the max file, but the user does not have the opportunity to load any further state.

6.2 Integration with the HARNESS Platform

In Year 2, once the run-time management system described in Section 6.1 is complete, our technical effort in WP3 will move towards integrating this work with the HARNESS platform, currently being developed in WP6, along with other heterogeneous management components being developed in WP4 and WP5, respectively.

In this integrated system, the HARNESS platform will provision computation, communication and storage resources to allow the run-time computation management system process dispatched jobs (Section 3.3.1). One key focus of our research will be to optimise the dynamics between the cloud platform and the computation management system to maximise the utilisation of heterogeneous compute resources and machines in a cluster. On the one hand, budgeting decisions made by the HARNESS platform restrict the choices made by the computation management system. On the other, low-level mapping decisions made by the computation management system and the resulting feedback may be used to improve future budgeting decisions.

The run-time computation management system also interfaces, albeit indirectly, with communication and storage management processes (Figure 3.1). Storage resources proposed in WP5 will be managed by

XtreemFS [34], with local and remote storage accessed transparently using POSIX I/O function calls on mounted volumes [34]. The platform also allocates communication resources (WP4) to allow the run-time computation management system to access remote compute and storage nodes.

Part of the integration effort in WP3 will involve developing the *infrastructure resource scheduler* component, which is part of the HARNESS platform and described in Deliverable D6.3.1 [36]. This component is the main mechanism by which the HARNESS platform and the run-time management system interact to request and provide: resource reservation, application deployment and monitoring. We envision the following steps (see Section 3.3 of D6.3.1 [36]):

1. Users submit an application manifest [35] describing the application and the *service-level objective* (SLO) to the HARNESS platform. The manifest contains the application organisation and the physical computation resources required for execution. The SLO, on the other hand, captures expected execution time or the budget to complete this execution.

2. The HARNESS platform proceeds with the following steps (using the application manager and cross-resource scheduler):

   (a) find a suitable resource configuration that satisfies the SLO;

   (b) compute an optimised schedule with discovered resources; and

   (c) reserve scheduled resources.

3. The HARNESS platform (through the resource dispatcher) deploys the run-time computation management system on a host machine, mounts compute and storage resources using allocated network resources, and launches the application. More details about the application deployment can be found in Section 3.3.1.

The integration effort between WP3 and WP6 will imply three management tiers, related to the cloud platform, the executive and the compute resource. Each of these tiers operate within their own respective concerns, scope and objectives (see Table 3.2). One key challenge in Year 2 will be to create synergies between these components in order to maximise the efficacy of the platform. This can be achieved through well-established separation of concerns, and effective feedback from low-level to top-level components.

6.3 Extend the Unified Heterogeneous Programming Approach

In Year 1, our compile-time work included the design and implementation of the uniform heterogeneous programming model (Section 3.6). The variant descriptions this approach enables can capture different algorithms or programming semantics that facilitate the efficient mapping of applications to specialised heterogeneous architectures. We also allow non-functional concerns to be codified in an aspect-oriented language called LARA, which captures optimising strategies that can be applied automatically and systematically to functional code to derive optimised descriptions of the application. In Year 1, we investigated a number of aspect-oriented strategies specific for dataflow computing (Section 4.3).

In Year 2, we plan to capture more complex design-space exploration strategies that exploit specialised features of resources such as DFEs and GPGPUs, including run-time reconfiguration and data-parallel computations, to generate efficient implementations. We also want to explore ways in which we can minimise the porting effort of software applications to the HARNESS platform.

Finally, we wish to generalise how to model and realise elastic applications in the context of heterogeneous resources. Elasticity is a property commonly associated with the platform. In this context, the platform dynamically provisions resources based on the needs of the application. The reverse problem, in which applications dynamically adapt to provisioned resources, gives the platform more control and flexibility to manage resources in a multi-tenancy environment.

7 Conclusion

In this deliverable we present the general methods in which we characterise and manage compute resources, and how heterogeneous compute resources can be programmed to leverage their potential. In particular, we are investigating (i) how applications can dynamically exploit heterogeneous compute resources at run time and (ii) how applications can be programmed to allow a more flexible and efficient mapping to heterogeneous platforms at compile time. In the context of this work, we are developing two novel and complementary approaches:

1. a **two-tier run-time computation management system** that allocates heterogeneous compute resources to adapt workloads *horizontally* across multiple compute nodes and *vertically* by exploiting multi-core heterogeneous resources available in a compute node and

2. a **unified heterogeneous programming approach** that decouples functional descriptions from non-functional concerns.

In our approach, functional descriptions can capture multiple semantics to describe a computation using the same language, such as imperative and dataflow, allowing a more efficient mapping to heterogeneous compute resources such as CPUs and DFEs. Non-functional descriptions, on the other hand, allow user knowledge and expertise to be codified with aspects. Aspects can then be applied and even reused to automatically optimise functional descriptions.

As a result of this research, we have developed three infrastructures covering the run-time and compile-time approaches, and have evaluated them with the HARNESS validation use cases, namely RTM, Delta Merge and AdPredictor:

- **SHEPARD run-time management infrastructure.** SHEPARD targets OpenCL devices. SHEPARD has the ability to adapt allocation and to share out workload, thus removing fixed or static allocation of tasks in application code. When allocating tasks to resources based on the column size in the Delta Merge validation use case [32], our managed approach can adequately choose the device that yields the lowest execution time. Concurrent tasks are implicitly shared between multiple devices based on expected execution time and current device load.

- **MaxelerOS run-time management infrastructure.** One of the key features of this prototype is the use of the *groups* abstraction, which allows a cluster of dataflow engines to be managed as a single compute entity, offering a low-level elastic platform that can automatically adapt to workload.

- **Aspect-oriented dataflow design infrastructure.** This programming infrastructure supports an aspect-driven design flow for deriving optimised DFE designs. More specifically, we focus on codifying aspects that minimise development effort, exploit DFE architectural features, and explore variant implementations and run-time reconfiguration.

In addition, we have developed two prototypes, targeting RTM and AdPredictor, to investigate how these applications scale on dataflow engines and to help us derive performance models for CPUs and DFEs:

- **Dynamic stencil computation prototype.** This prototype implements a *dynamic stencil*, which is a special type of design that implements stencil computations and that scales automatically and dynamically on a platform with multiple DFEs connected with each other. Experimental results with RTM show that high throughput and significant resource utilisation can be achieved with dynamic stencil designs, which can dynamically scale into reconfigurable computing nodes as they become available during their execution.
- **AdPredictor parallel training rrototype.** The AdPredictor prototype allows the evaluation of the multi-threaded CPU and DFE designs targeting the 2012 KDD Cup data set [32]. This tool can be used to derive performance models for CPUs using an arbitrary number of threads, and for the AdPredictor training DFE design also included with this prototype.

We intend to extend the work done in Year 1 and integrate our existing run-time management prototypes by virtualising MaxelerOS and OpenCL devices, as well as support horizontal allocation of heterogeneous compute nodes. In addition, we will integrate our run-time management system into the HARNESS platform. We also intend to broaden the scope of our unified heterogeneous programming approach to support data parallel computations to support GPGPUs and new generations of many-core architectures, such as the Intel Phi.

Bibliography

[1] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 363–368. European Design and Automation Association, 2006.

[2] Amazon, Inc. Elastic Beanstalk. Available at `http://aws.amazon.com/elasticbeanstalk/`.

[3] AMD Inc. AMD Heterogeneous Uniform Memory Access. Available at `http://goo.gl/bzKrlC`.

[4] AMD Inc. AMD Opteron™ 6300 Series Processors. Available at `http://goo.gl/cU61ct`.

[5] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–49, 2009.

[6] M. Araya-Polo et al. Assessing Accelerator-Based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 22:147–162, Jan. 2011.

[7] ARM Holdings plc. Introducing big.LITTLE technology. Available at `http://www.thinkbiglittle.com`.

[8] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. *ACM Sigplan Notices*, 45(10):89–108, 2010.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*. Springer-Verlag New York, Inc., 1993.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[12] J. Cardoso, T. Carvalho, J. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. LARA: An aspect-oriented programming language for embedded systems. In *Proceedings of the Annual International Conference on Aspect-Oriented Software Development*, pages 179–190, 2012.

[13] J. M. Cardoso. Programming strategies for runtime adaptability. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8. IEEE, 2012.

[14] J. M. Cardoso, T. Carvalho, J. G. Coutinho, P. C. Diniz, Z. Petrov, and W. Luk. Controlling hardware synthesis with aspects. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 226–233. IEEE, 2012.

[15] J. M. Cardoso, J. Teixeira, J. C. Alves, R. Nobre, P. C. Diniz, J. G. F. Coutinho, and W. Luk. Specifying compiler strategies for fpga-based systems. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 192–199. IEEE, 2012.

[16] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[17] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. The MIT Press, 2008.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[19] T. Chockalingam and S. Arunkumar. A randomized heuristics for the mapping problem: The genetic approach. *Parallel computing*, 18(10):1157–1165, 1992.

[20] CloudFoundry. Available at `http://www.cloudfoundry.com/`.

[21] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[22] ConPaaS Project. Available at `http://www.conpaas.eu/`.

[23] I. Corporation. Intel quickassist technology accelerator abstraction layer (aal). Available at `http://goo.gl/ixk8Px`.

[24] J. G. F. Coutinho, S. Bhattacharya, W. Luk, G. A. Constantinides, J. M. Cardoso, T. Carvalho, P. C. Diniz, and Z. Petrov. Resource-Efficient Designs using an Aspect-Oriented Approach. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 399–406. IEEE, 2012.

[25] J. G. F. Coutinho, J. M. Cardoso, T. Carvalho, S. Bhattacharya, W. Luk, G. Constantinides, P. C. Diniz, and Z. Petrov. Aspect-based source to source transformations. In *Compilation and Synthesis for Embedded Reconfigurable Systems*, pages 71–103. Springer New York, 2013.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters.

[27] A. DeHon. Compute models and system architectures. In *Reconfigurable Computing*, pages 91–127. Morgan Kaufmann, 2008.

[28] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[29] E. Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[30] R. A. Ferreira, W. Meira Jr, D. Guedes, L. M. d. A. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pages 159–166. IEEE, 2005.

[31] FP7 HARNESS Consortium. General requirements. Project Deliverable D2.1, 2013.

[32] FP7 HARNESS Consortium. Industrial requirements. Project Deliverable D2.2, 2013.

[33] FP7 HARNESS Consortium. Validation plan. Project Deliverable D2.3, 2013.

[34] FP7 HARNESS Consortium. Characterisation report. Project Deliverable D5.1, 2013.

[35] FP7 HARNESS Consortium. Application characterisation report. Project Deliverable D6.1, 2013.

[36] FP7 HARNESS Consortium. Heterogeneous platform implementation (initial). Project Deliverable D6.3.1, 2013.

[37] M. Frigo. Multithreaded programming in cilk. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 13–14. ACM, 2007.

[38] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the fft. In *ICASSP*, volume 3, pages 1381–1384, 1998.

[39] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[40] Google, Inc. App Engine. Available at `https://developers.google.com/appengine/`.

[41] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale Bayesian click-through rate prediction for sponsored search advertising in Microsoft's Bing search engine. In *Proceedings of the International Conference on Machine Learning*, 2010.

[42] P. Grigoras, X. Niu, J. G. F. Coutinho, W. Luk, J. Bower, and O. Pell. Aspect Driven Compilation for Dataflow Designs. In *Proceedings of the IEEE Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2013.

[43] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. the MIT Press, 1999.

[44] K. O. W. Group et al. The OpenCL specification. *A. Munshi, Ed*, 2008.

[45] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *Micro, IEEE*, 26(2):10–24, 2006.

[46] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[47] J. Hoberock and N. Bell. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems, Jade Edition*, pages 359–372, 2011.

[48] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP 2008–Object-Oriented Programming*, pages 76–103. Springer, 2008.

[49] IBM Corp. IBM Power Systems. Available at `http://goo.gl/cU1YR`.

[50] IBM Corp. SPU Instruction Set Architecture. Available at `http://goo.gl/7axGQg`.

[51] Intel Corp. An Introduction to the 4th Generation Intel Core™ Processor. Available at `http://goo.gl/tpiz2u`.

[52] Intel Corp. Intel Many Integrated Core Architecture. Available at `http://goo.gl/WbEIj8`.

[53] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[54] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP 2001Object-Oriented Programming*, pages 327–354. Springer, 2001.

[55] A. Kukanov and M. J. Voss. The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322, 2007.

[56] Y. M. Lam, J. Coutinho, and W. Luk. Integrated Hardware/Software Codesign for Heterogeneous Computing Systems. In *Programmable Logic, 2008 4th Southern Conference on*, pages 217–220. IEEE, 2008.

[57] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[58] L. Lu and K. Magerlein. Multi-Level Parallel Computing of Reverse Time Migration for Seismic Imaging on Blue Gene/Q. pages 291–292, 2013.

[59] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[60] W. Luk, J. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. W. Susanto, Q. Liu, and W. Wong. A high-level compilation toolchain for heterogeneous systems. In *SOC Conference, 2009. SOCC 2009. IEEE International*, pages 9–18. IEEE, 2009.

[61] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[62] Microsoft, Inc. Azure Services Platform. Available at `http://www.azure.net/`.

[63] Microsoft Bing. Available at `http://www.bing.com/`.

[64] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proc. of the Int. Conf. for SC*, pages 1–13, 2010.

[65] X. Niu, T. Chau, Q. Jin, W. Luk, and Q. Liu. Automating Elimination of Idle Functions by Run-Time Reconfiguration. In *Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[66] X. Niu, J. G. F. Coutinho, and W. Luk. A Scalable Design Approach for Stencil Computation on Reconfigurable Clusters. In *Proceedings of the IEEE on Field Programmable Logic and Applications (FPL)*, 2013.

[67] X. Niu, J. G. F. Coutinho, Y. Wang, and W. Luk. Dynamic Stencil: Effective Exploitation of Run-time Resources in Reconfigurable Clusters. In *Proceedings of the IEEE on Field-Programmable Technology (FPT)*, 2013.

[68] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell. Exploiting Run-Time Reconfiguration in Stencil Computation. In *IEEE Conference on Field Programmable Logic and Applications*, pages 173–180, 2012.

[69] OpenShift. Available at `https://www.openshift.com/`.

[70] Oracle Corp. SPARC T5 Processor. Available at `http://goo.gl/t2V0V`.

[71] O. Pell et al. Finite Difference Wave Propagation Modeling on Special Purpose Dataflow Machines. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):906–915, 2013.

[72] O. Pell, O. Mencer, H. T. Kuen, and W. Luk. Maximum performance computing with dataflow engines. In *High-Performance Computing Using FPGAs*, pages 747–774. Springer-Verlag, 2013.

[73] M. Perrone et al. Reducing Data Movement Costs: Scalable Seismic Imaging on Blue Gene. In *IPDPS*, pages 320–329, 2012.

[74] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[75] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems. *IEEE Trans. Computers*, 36(7):845–858, 1987.

[76] REFLECT Project. Available at `http://www.reflect-project.eu/`.

[77] M. Rietmann et al. Forward and Adjoint Simulations of Seismic Wave Propagation on Emerging Large-scale GPU Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 38, 2012.

[78] M. Ripeanu et al. Cactus Application: Performance Predictions in Grid Environments. pages 807–816, 2001.

[79] R. R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[80] L. Schubert and K. Jeffery. Advances in clouds—Research in future cloud computing. Expert Group Report, European Commission, Information Society and Media, 2012.

[81] C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, pages 47–63. Springer, 2011.

[82] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, Inc., 2002.

[83] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and its Applications*, pages 9–16. ACM, 2011.

[84] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(13):1411–1430, 2012.

[85] T. Todman and W. Luk. Reconfigurable design automation by high-level exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 185–188. IEEE, 2012.

[86] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk, and P. Y. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005.

[87] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 users guide. Available at `http://goo.gl/Qr8Fei`.

[88] K. H. Tsoi and W. Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124. ACM, 2010.

[89] M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. pages 163–170, 2000.

[90] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521, 2005.

[91] J. R. Wernsing and G. Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 115–124, 2010.

[92] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *CDROM*, pages 1–27, 1998.

[93] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC – First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.